Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

State-inferer: LLM-Driven State Machine Extraction and Formal Verification for Secure Protocol Implementations

MengCheng Ju

zhangph12138@163.com

PLA Information Engineering University

Xi Chen

PLA Information Engineering University

XieLi Zhang

PLA Information Engineering University

Mingliang Zhu

PLA Information Engineering University

Guanping Zhang

PLA Information Engineering University

Dongliang Zhao

PLA Information Engineering University

ChunXiang Gu

PLA Information Engineering University

Research Article

Keywords:

Posted Date: November 9th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11391337/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

State-inferer: LLM-Driven State Machine Extraction and Formal Verification for Secure Protocol Implementations

Abstract

The security of a network protocol implementation depends on the finite-state machine that its code realizes. Implementations often deviate from their RFC specifications, and these deviations lead to state-machine vulnerabilities such as authentication bypass. Static extraction of state machines from source code suffers from path explosion, dynamic inference is limited by the inputs it observes, and analyses based on large language models (LLMs) cannot fit a protocol codebase into the context window and miss transitions that depend on internal variables. This paper presents State-inferer, a white-box framework that combines program slicing, LLMs, and model checking to extract and verify protocol state machines. Instead of retrieving code by textual similarity, State-inferer slices the system dependence graph under depth bounds and gives the LLM one code fragment per state transition. From these fragments it builds a state transition system whose transitions carry guards over internal variables, and it specializes RFC security properties into linear temporal logic formulas over this system for checking with nuXmv. We evaluate State-inferer on seven open-source implementations of TLS, IKEv2, and SSH. State-inferer achieves a macro-average strict transition-level F-score of 95.2, compared with 78.9 for the strongest baseline, and uncovers four previously unknown vulnerabilities, all of which have been assigned CVE or CNVD identifiers.

Keywords: Large Language Model, Program Analysis, Protocol Reverse Engineering, State Machines, Software Security

1 Introduction

Network protocols are implemented as finite state machines (FSMs), and the security of a protocol stack depends on whether its implementation realizes the intended FSM (Pham et al. 2020). Real implementations such as TLS or IPsec stacks often deviate from their RFC specifications, through misread requirements or vendor-specific logic (De Ruiter and Poll 2015; Yuan et al. 2024), and such deviations have repeatedly produced state-machine vulnerabilities, including authentication bypass and leakage of sensitive data (Beurdouche et al. 2015; Bhargavan et al. 2014). Analyzing these

flaws requires the FSM that the code actually implements, not the one the specification describes (de Ruiter 2016).

Recovering this FSM from source code is hard. Protocol state is spread across conditional branches and combinations of internal variables. Static analysis runs into path explosion, and dynamic analysis only sees the behavior that its test inputs exercise. Large language models (LLMs) can read and reason about code (Zhang et al. 2023), which makes automated FSM extraction from source code a realistic goal. Using them for formal security analysis of protocol implementations, however, raises three problems.

C1: Abstract models miss implementation state. Existing methods (de Ruiter 2016; Fiterău-Broştean et al. 2017; Wei et al. 2025) build message-driven state machines that abstract away the implementation, whereas the state of a real protocol stack also depends on internal variables (e.g., state flags and counters) and on control-flow conditions (Beurdouche et al. 2015). When states and inputs are derived from external messages, these internal distinctions are merged into coarse states or dropped, and the resulting models cannot express the implementation-level behavior that verification needs.

C2: Code context exceeds the LLM window. Protocol codebases are far larger than the context window of current LLMs. Retrieval-augmented generation (RAG) (Lewis et al. 2020) reduces the amount of code passed to the model, but similarity-based retrieval does not return the cross-function and cross-file call chain that leads to a state update as a unit, so the LLM sees only parts of a transition path.

C3: Specification properties do not name implementation variables. RFCs state security properties without reference to any implementation. Since state-variable names and code structure differ widely across implementations, each property must be rewritten as a Linear Temporal Logic (LTL) (Pnueli 1977) formula over the variables of a particular code base before it can be model-checked, a step that existing extraction tools leave to the analyst.

ProtocolGPT (Wei et al. 2025) showed that an LLM can derive FSMs from protocol code, but it builds the code context from retrieved chunks that need not follow the dependencies of a transition, produces message-level models, and does not connect the extracted model to security properties.

We present State-inferer, a white-box framework for extracting and verifying protocol state machines, which addresses the three problems as follows. For C1, it builds a state transition system (STS) whose transitions carry guards over internal variables. For C2, it slices the system dependence graph (SDG) (Weiser 1984; Horwitz et al. 1990; Jackson and Rollins 1994) forward and backward under depth bounds, and gives the LLM one compact code fragment per state transition. For C3, it specializes each RFC security property into an LTL formula over the extracted model and checks it with nuXmv (Cavada et al. 2014).

The main contributions of this paper are as follows:

1. A slicing method for protocol state recovery: forward and backward search on the SDG, bounded separately along data-flow and control-flow edges, extracts one compact code fragment (gadget) per state transition.

2. A pipeline that turns the sliced code into an implementation-level STS with guarded transitions and specializes RFC security properties into LTL formulas over that STS for verification with nuXmv.
3. An evaluation on seven open-source implementations of TLS, IKEv2, and SSH. At the transition level, State-inferer is compared with ProtocolGPT and two coding agents; we further measure how many reference transitions a Mealy-style representation cannot express, compare dependency slicing with retrieval-based and agentic context construction, check the generated security properties against expert-written formulas, and vary the slicing depth and the model backend. State-inferer found four previously unknown vulnerabilities, which have been disclosed to the maintainers and assigned CVE or CNVD identifiers.

2 Related Work

Protocol state machines describe how communicating peers interact. Methods for inferring them fall into two groups: static and dynamic program analysis, and approaches based on natural language processing (NLP) and language models.

Static and dynamic analysis. Static approaches analyze source code with techniques such as symbolic execution and data-flow analysis. Shi et al. (2023) extract message formats and represent them as state machines through controlled static loop analysis. Yuan et al. (2024) infer the state machines of MQTT implementations from predefined state templates and symbolic execution, and use them to detect logic flaws. Static methods generalize and scale poorly (Fujiwara et al. 1991): they suffer from state explosion on large programs and require source code.

Dynamic approaches do not need source code. They infer state machines from the data-flow or control-flow behavior observed during message parsing, using network traces, statistical methods, or active learning. Kleber et al. (2018; 2020) classify and reverse-engineer binary protocol message types based on intrinsic message structure or segment similarity. Ye et al. (2021) propose a probabilistic method grounded in joint distributions to reverse-engineer network protocols from message traces. Active model learning treats the system as a black box and extracts its state machine through membership and equivalence queries. Angluin's L* algorithm (Angluin 1987) introduced the minimally adequate teacher framework, which was later adapted to Mealy machines and applied to many security protocols. De Ruiter et al. (2016; 2015) performed large-scale black-box and state-fuzzing analysis of TLS implementations. Fiterău-Broştean et al. (2016; 2017; 2017) combined model learning with model checking for SSH and TCP implementations. Howar et al. (2012) proposed register automata learning and applied it to TCP sliding-window behavior. Tappler et al. (2017), Aarts et al. (2013), Guo et al. (2019), Wang et al. (2021), and Chen et al. (2024) applied model learning or testing to IoT, IPsec, and MQTT protocols, among others. McMahon Stone et al. (2022) identified states from memory snapshots for grey-box learning of TLS and Wi-Fi implementations. All of these dynamic and learning-based methods can only model the behavior that their inputs reach (Ba et al. 2022; Natella 2022; Daniele et al. 2024).

Table 1 State and transition counts of the source-level (Code), specification (RFC), and learned I/O state machines.

Implementation	Protocol Version	Code		RFC		I/O	
		States	Trans.	States	Trans.	States	Trans.
OpenSSL	TLS 1.3	20	38	8	8	15	22
WolfSSL	TLS 1.3	10	16	8	8	8	11
TLSE	TLS 1.3	12	22	8	8	9	14
strongSwan	IKEv2	12	27	4	6	9	14
Libreswan	IKEv2	22	29	4	6	10	15
OpenSSH	SSH	14	21	6	5	7	10
tinySSH	SSH	9	13	6	5	5	7
Average		**14.14**	**23.71**	**6.29**	**6.57**	**9.00**	**13.29**

NLP and language-model-based approaches. Several works extract state machines from natural-language specifications. Al Ishtiaq et al. (2024) synthesize FSMs from cellular protocol specifications and use them to find vulnerabilities, and Pacheco et al. (2022) extract FSMs from RFC text for automated attack synthesis. LLMs have also been applied to code generation, program repair, vulnerability detection, and protocol fuzzing. Wei et al. (2025) use an LLM to infer state machines from protocol source code (ProtocolGPT). IRIS (Li et al. 2025) combines an LLM with static analysis: the LLM infers taint specifications and filters alerts for whole-repository vulnerability detection. State-inferer shares with IRIS the idea of letting a program analysis decide which code reaches the LLM, but it targets protocol state transitions rather than taint flows. Specification-level methods model what the RFC requires and black-box methods model what can be observed at the interface; State-inferer instead models the internal states and guards of the implementation, and its output can also serve as input to fuzzing and formal verification.

3 Motivation and Case Study

This section uses three case studies to show where existing approaches fall short when FSMs are inferred from protocol implementations.

3.1 Semantic Misalignment Between Protocol Implementations and State-Machine Models

Prior work usually describes protocol behavior by external input messages and coarse protocol states. Such I/O state machines are obtained by model learning (Angluin 1987; de Ruiter 2016; Fiterău-Broştean et al. 2017), i.e., by learning an automaton from message sequences or inferring transitions from observed input–output behavior. Because the input is a fixed set of observable messages, the learned model differs from the state machine in the source code in two ways. Some transitions are driven by internal logic, such as timer expiry or a failed resource allocation, and have no corresponding message. Many error-handling branches and boundary states are reached only by specially crafted malformed messages. Finite internal configurations could in principle be encoded as Mealy states, and internal events added to the input alphabet.

A model learned over a fixed message alphabet through an external interface, however, cannot separate configurations that produce the same observable behavior, and therefore does not reflect how the code actually executes.

To measure the difference, we took seven implementations of TLS, IKEv2, and SSH. For each one, we learned an I/O state machine by active model learning, and two domain experts extracted the source-level state machine from the code as the reference. Table 1 shows that the source-level machines have on average 14.1 states and 23.7 transitions, against 9.0 and 13.3 for the I/O machines and 6.3 and 6.6 in the RFCs. Implementations of the same standard also differ: strongSwan (StrongSwan Project 2026) and Libreswan (Libreswan Project 2026) both implement IKEv2, yet have 12 and 22 states. An inference method that relies on a fixed message set cannot capture these implementation-specific differences.

Finding 1: I/O state machines learned over a fixed message alphabet miss the states and transitions that depend on internal variables.

3.2 Incomplete Path Tracking Under LLM Context Constraints

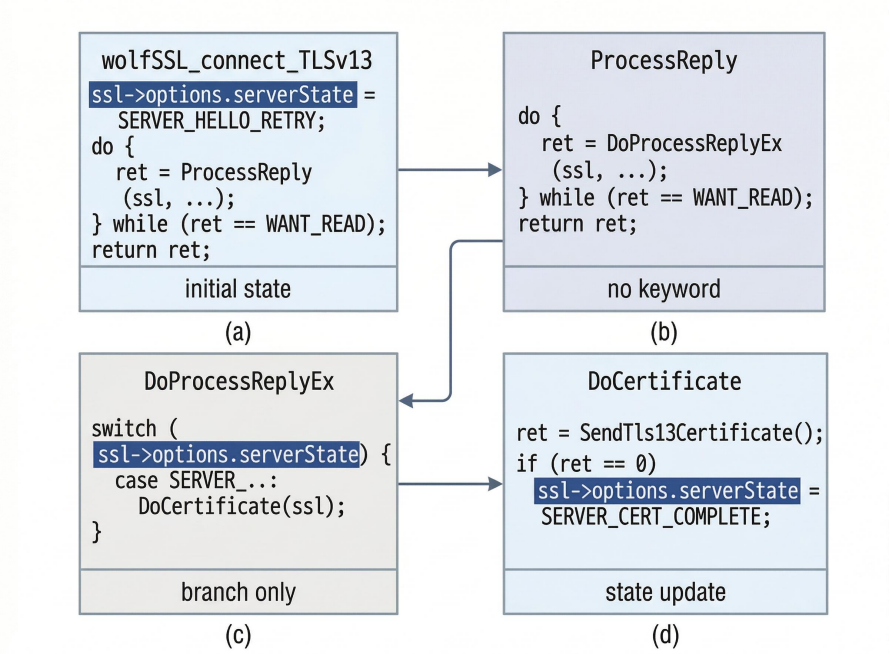


Fig. 1 WolfSSL TLS 1.3 handshake call chain.

Protocol codebases contain tens to hundreds of thousands of lines, and the logic that updates protocol state runs through call chains that cross functions and files. A whole codebase does not fit into the context window of current LLMs.

Existing work therefore uses retrieval-augmented generation (RAG) (Lewis et al. 2020): code snippets are retrieved by textual similarity or keywords and passed to the LLM. Retrieval returns code blocks that resemble the query, typically those that mention the target state variable, but not the control-flow and data-flow relations among them. In the WolfSSL TLS 1.3 handshake of Figure 1, the update of `serverState` spans four nested calls: `wolfSSL_connect_TLSv13()` → `ProcessReply()` → `DoProcessReplyEx()` → `DoCertificate()`. Retrieval finds the assignments to `serverState` in the first and last functions, but not the calls and parameter passing that connect them, since that code does not mention `serverState`. The preconditions of the transition sit in conditional branches of the two intermediate functions; these branches do not mention `serverState` either, so they are ranked as irrelevant and left out.

With such gaps in the context, the LLM labels some states as unreachable and reduces transition conditions to the received message, dropping the constraints on internal variables. The missing piece is a connected cross-function context, not a better reading of the code that is supplied.

Finding 2: Similarity-based retrieval does not supply the cross-function call chain of a transition, so LLM-based extraction omits transition conditions and misjudges reachability.

3.3 Difficulty of Converting Generic Security Properties to Implementation-Specific Model Properties

RFCs state security properties without reference to any implementation. Implementations differ in state-machine structure and variable naming, so a property template written once cannot be reused as is; it has to be instantiated for each implementation.

Take the TLS requirement that the handshake complete before application data is sent. In the RFC, this is a precedence relation: "key negotiation complete → application data transfer." Its formal version depends on the symbols of each model. OpenSSL TLS 1.2 marks handshake completion by `hand_state` taking the value `TLS_ST_OK`, whereas WolfSSL TLS 1.2 expresses it as `ssl->options.handShakeState == HANDSHAKE_DONE`. A single template applied to both either fails to instantiate or binds to the wrong variables.

Meta-properties are stated at the level of protocol semantics, whereas a checkable property must refer to the states and variables of the implementation-level FSM. Transferring a property from one implementation to another therefore requires a mapping from specification concepts to implementation variables.

Finding 3: RFC security properties must be mapped to the variables of each implementation-level FSM before they can be verified.

4 System Design and Implementation

4.1 Overview

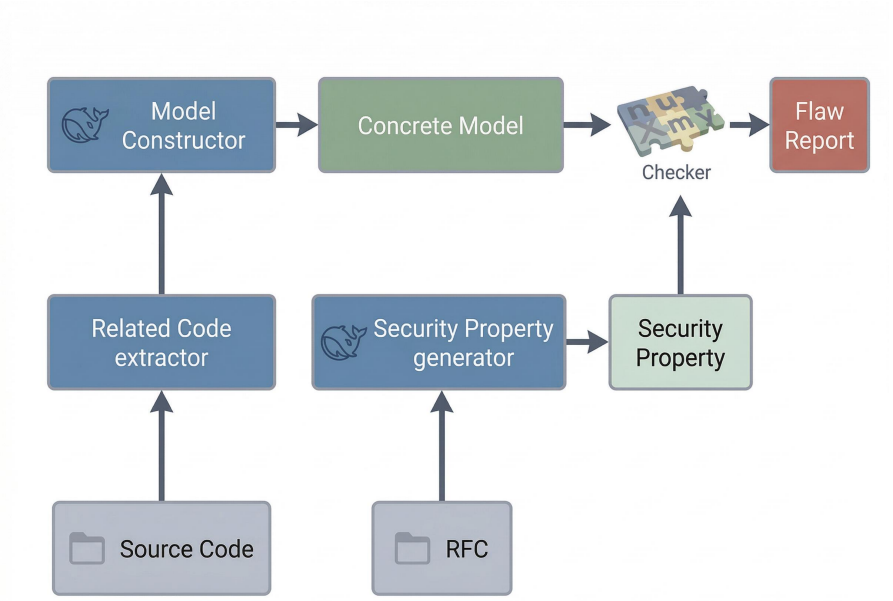


Fig. 2 The architecture of State-inferer.

State-inferer recovers the state machine of a protocol implementation from its source code and checks security properties on it with a model checker. It has three components: the Related Code Extractor, the Model Constructor, and the Security Property Generator.

Figure 2 shows the workflow. The Related Code Extractor extracts the code fragments relevant to state transitions from the target project. The Model Constructor identifies states in these fragments, extracts transition conditions, and assembles a global state-machine model.

The Security Property Generator combines the model with the RFC to produce security properties over the model's variables. The model and the properties are then

translated into the input language of nuXmv (Cavada et al. 2014). For each violated property, nuXmv returns a counterexample: the transition chain from the initial state to the violating state together with the triggering actions. Counterexamples are reviewed manually.

4.2 Related Code Extractor

Protocol codebases also contain configuration parsers, logging subsystems, test harnesses, and other code unrelated to the state machine. Passing raw source code to an LLM exhausts the context window (Lewis et al. 2020) and adds irrelevant code that can cause hallucination. The Related Code Extractor (RCE) reduces the codebase to the code needed for FSM construction, in two steps: state-variable identification and program slicing (Weiser 1984).

[Instruction]
You are a code analysis expert proficient in network protocol implementations. Your task is to identify the key variables representing the Finite State Machine of the `{protocol_name}` protocol from the provided codebase `{code_dir}`, along with their possible enumeration values.

[Output format]:

```
{"state_variables": [{"state_variable1": "value1, value2,
    ..."},{ "state_variable2": "..."}]}
```

Fig. 3 Prompt template for extracting protocol FSM state variables and their possible values.

4.2.1 State Variable Identification

The state-machine logic of a protocol usually revolves around a few designated state variables. Their naming conventions, however, vary considerably across implementations, so regular-expression matching is largely ineffective. We instead use structured prompts that guide the LLM to locate these variables. As illustrated in Figure 3, the LLM acts as a protocol code auditor and is instructed to scan header files and core source files for explicitly defined enumeration types.

4.2.2 Program Slicing

Identifying state variables alone does not characterize the state machine: transitions depend on contextual constraints such as input validation and handshake parameter negotiation, and the relevant logic often spans multiple functions. We therefore slice the System Dependence Graph (SDG) (Horwitz et al. 1990; Jackson and Rollins 1994), starting from the assignments to state variables, to extract a compact code

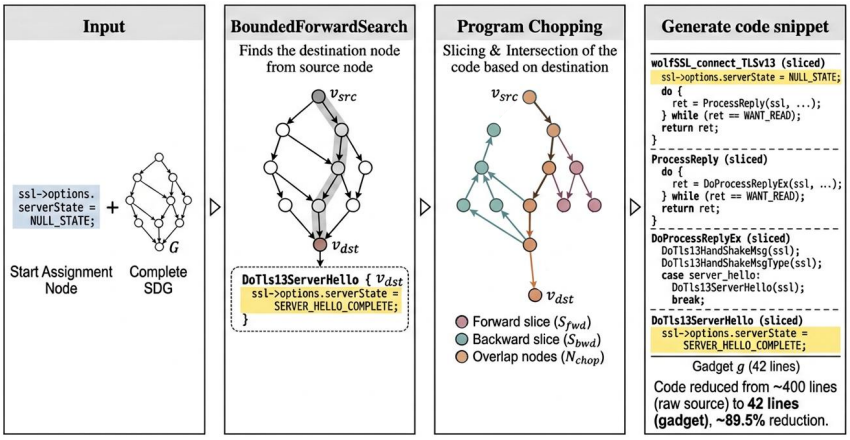


Fig. 4 Depth-bounded slicing on the SDG: forward and backward searches from state-assignment nodes yield compact gadgets for each state transition.

fragment for each state transition. The SDG is built with Joern (Yamaguchi et al. 2014), whose code property graph provides the interprocedural data-dependence and control-dependence edges traversed by the slicer.

Algorithm 1 details the slicing procedure. A state transition corresponds to two consecutive assignments to the same state variable. Starting from the current assignment node v_{src}, with the state variable as the slicing criterion, we perform a depth-bounded forward search on the SDG, where k_d and k_c bound the data-flow and control-flow traversal depths, respectively. The search stops at the next assignment node v_{dst} of the same variable, which prevents slice explosion and excludes unrelated low-level code. Because one v_{src} may lead to several transitions, the forward search yields a set of candidate endpoints $\mathcal{N}_{\text{next}}$. For each $v_{\text{dst}} \in \mathcal{N}_{\text{next}}$, a symmetric backward search recovers the preconditions of the state update. The intersection of the forward and backward slices, augmented with $\{v_{\text{src}}, v_{\text{dst}}\}$ and sorted by line number, forms a *gadget* that encodes one guarded transition.

Figure 4 illustrates this process on the WolfSSL TLS 1.3 handshake. With the assignment of `NULL_STATE` as v_{src}, the forward search reaches the assignment of `SERVER_HELLO_COMPLETE` as v_{dst} within the depth bounds. The forward and backward slices overlap at intermediate nodes such as `ProcessReply` and `DoProcessReplyEx`, so the resulting fragment spans function boundaries and keeps the routing and validation logic of this transition. Here, logic spread over 4 functions and about 400 lines of code is reduced to a 42-line gadget (an 89.5% reduction) that preserves the transition and its guard conditions and fits easily within the LLM context window.

4.3 Model Constructor

Formal model.

We model a security protocol as a state transition system $M = (\mathcal{V}, \mathcal{S}, s_0, \delta, \mathcal{A})$ (Yuan et al. 2024), where: $\mathcal{V}$ is a finite set of variables whose valuations determine when

Algorithm 1 Depth-Bounded Gadget Extraction for State Transitions

Require: SDG $G = (\mathcal{N}, \mathcal{E})$, start node v_{src},
state variable var, depth bounds k_d, k_c
Ensure: Set of gadgets $\mathcal{G}$

$\mathcal{G} \leftarrow \emptyset$
$(\mathcal{N}_{\text{fwd}}, \mathcal{N}_{\text{next}}) \leftarrow$
BoundedFwdSearch$(G, v_{\text{src}}, \text{var}, k_d, k_c)$
for each $v_{\text{dst}} \in \mathcal{N}_{\text{next}}$ **do**
  $\mathcal{N}_{\text{bwd}} \leftarrow$
  BoundedBwdSearch$(G, v_{\text{dst}}, k_d, k_c)$
  $\mathcal{N}_{\text{chop}} \leftarrow \mathcal{N}_{\text{fwd}} \cap \mathcal{N}_{\text{bwd}}$
  $\mathcal{N}_{\text{chop}} \leftarrow \mathcal{N}_{\text{chop}} \cup \{v_{\text{src}}, v_{\text{dst}}\}$
  $g \leftarrow$ SortByLineNumber$(\mathcal{N}_{\text{chop}})$
  $\mathcal{G} \leftarrow \mathcal{G} \cup \{g\}$
end for
return $\mathcal{G}$

function BoundedFwdSearch$(G, v_{\text{src}}, \text{var}, k_d, k_c)$
  $Q \leftarrow \{(v_{\text{src}}, 0, 0)\}$
  $\mathcal{N}_{\text{vis}} \leftarrow \{v_{\text{src}}\}$
  $\mathcal{N}_{\text{next}} \leftarrow \emptyset$
  while $Q \neq \emptyset$ **do**
    $(u, d_d, d_c) \leftarrow Q$.pop()
    for each $e = (u, v) \in \mathcal{E}$ **do**
      $d'_d \leftarrow d_d + \mathbb{I}[e \in \mathcal{E}_{\text{data}}]$
      $d'_c \leftarrow d_c + \mathbb{I}[e \in \mathcal{E}_{\text{ctrl}}]$
      if $d'_d \leq k_d \wedge d'_c \leq k_c \wedge v \notin \mathcal{N}_{\text{vis}}$ **then**
        $\mathcal{N}_{\text{vis}} \leftarrow \mathcal{N}_{\text{vis}} \cup \{v\}$
        if IsAssignment(v, var) **then**
          $\mathcal{N}_{\text{next}} \leftarrow \mathcal{N}_{\text{next}} \cup \{v\}$
        else
          Q.push$((v, d'_d, d'_c))$
        end if
      end if
    end for
  end while
  return $(\mathcal{N}_{\text{vis}}, \mathcal{N}_{\text{next}})$
end function

transitions fire; $\mathcal{S}$ is a finite set of states representing discrete phases of protocol execution; $s_0 \in \mathcal{S}$ is the initial state before any protocol interaction; $\delta : \mathcal{S} \times \mathcal{A} \times \text{Val}(\mathcal{V}) \to \mathcal{S}$ is the transition function, which determines the successor state from the current state, the triggering action, and the valuation of $\mathcal{V}$; and $\mathcal{A}$ is a finite set of actions that trigger protocol state transitions.

To map unstructured source code to M, State-inferer uses the LLM together with static code features, in two phases: state-space extraction and transition-logic formalization.

[Instruction]
You are an expert in protocol implementation and static code analysis. Your task is to infer and extract the initial state constant(s) of the target state variable(s) at system startup, based on the provided state variables and their enumeration values {state_variables}.

[Output format]:

```
{"state_variables": [{"state_variable1": "init_value1"}, {"state_variable2": "
    init_value2"}, ...]}
```

Fig. 5 Prompt template for extracting initial state constants of protocol state variables.

Construction of state set $\mathcal{S}$ and initial state s_0.

The discrete state space of a protocol is usually defined by enumeration types, structures, or macros in core header files or at global scope. To extract it without interference from unrelated code, the system first applies the state-extraction prompt in Figure 3. Given the codebase path and the protocol name, the LLM identifies the state variables of the protocol's FSM together with their enumeration values. This output bounds the states the protocol can take during execution, and the state set $\mathcal{S}$ of M is instantiated as the Cartesian product of the enumeration values.

Next, the system identifies the initial state s_0, i.e., the phase at startup before any network message has been processed. The state variables and their value domains from the previous step are fed into the initial-state prompt in Figure 5. From these domains and the typical structure of protocol initialization routines, the LLM infers the constant assigned to each state variable at startup.

Construction of transition function δ and variable set $\mathcal{V}$.

With the state space fixed, the main task is to extract the transition function δ and to abstract, from conditional branches, the variable set $\mathcal{V}$ that governs the transitions. Protocol code has deep and branching control flow, and an LLM asked to infer δ over the whole program confuses unrelated paths. The system therefore works on the local code snippets produced by the slicer.

The prompt for this phase asks the LLM to perform two tasks: first, identify the source and target states of the transition in the snippet and extract the control-flow conditions that trigger it; second, translate these code-level conditions into formal expressions and collect the variables that appear in them into $\mathcal{V}$. As shown in Figure 6, the prompt takes the sliced snippet {`code_slice`} as input and returns the source and target states, the formalized conditions, and the involved variables with natural-language descriptions.

By querying the LLM over all snippets, the system assembles the global transition function δ from the `source_state`, `action`, `target_state`, and `formal_conditions` fields of each snippet. It also merges and deduplicates the `involved_variables` of all snippets into the global variable set $\mathcal{V}$.

[Instruction]
You are an expert in formal verification and static analysis. Given the protocol code slice {code_slice}: (1) determine the source and target states of the transition; (2) extract all data-flow and control-flow preconditions and abstract them into formal Boolean expressions; (3) list involved variables with descriptions; (4) identify the triggering protocol action.

[Output example]:

```
{"source_state": "NULL_STATE",
  "target_state": "SERVER_HELLO_COMPLETE",
  "action": "recv_ClientHello",
  "formal_conditions": "msg_type == 1 && packet_len >= 64",
  "involved_variables": [
    {"variable_name": "msg_type",
      "description": "Parsed handshake message type; 1 = ClientHello"},
    {"variable_name": "packet_len",
      "description": "Received packet length"}]}
```

Fig. 6 Prompt template for extracting state transition conditions and actions.

Construction of action set $\mathcal{A}$.

The same prompt asks the LLM to label each transition with the protocol action that triggers it (the `action` field). Actions are of two kinds: *external actions* that involve the peer, such as sending or receiving a handshake message (e.g., `recv_ClientHello`, `send_ServerHello`), and *internal actions* that involve no network message, such as key derivation, certificate verification, or timeout handling. The system deduplicates the action labels of all snippets to form the finite action set $\mathcal{A}$. Each action $a \in \mathcal{A}$ labels one or more transitions in δ, which links the transitions of M to observable protocol behavior.

4.4 Model-Aware Security Property Specialization

Given the model M, the Security Property Generator specializes the meta-security properties of the RFC into LTL formulas over the extracted model. As illustrated in Figure 7, the specialization prompt takes the extracted model {`fsm_model`} and an RFC security property {`meta_property`} as input and directs the LLM to find the state variables and transition predicates that match the property, map the abstract requirement to implementation-level variables, and write the corresponding LTL formula.

For each target protocol, we first write a set of meta-properties that state its core security requirements in natural language. The Security Property Generator passes each meta-property together with the extracted model to the LLM through the prompt in Figure 7. The TLS meta-properties and example specializations are given in Appendix A.

[Instruction]
You are an expert in protocol specifications and formal verification. Given the extracted state machine model {fsm_model} and an RFC security property {meta_property}, perform model-aware property specialization: (1) identify concrete state variables, constants, and transition predicates matching the abstract security concept; (2) map abstract requirements to implementation-level variables; (3) derive Linear Temporal Logic (LTL) expressions tailored to this codebase.

[Output example]:

```
{"meta_property": "Complete handshake before data exchange",
  "ltl": "G (hand_state = TLS_ST_OK -> key_block != NULL)"}
```

Fig. 7 Prompt template for model-aware security property specialization.

4.5 Vulnerability Discovery with Model Checking

After model extraction and property specialization, State-inferer uses nuXmv (Cavada et al. 2014) as the back end for model checking. The extracted model and the LTL properties are first translated into the SMV language.

A translation script takes the state transition graph and a dictionary of LTL properties as input and performs the following mapping: variables and enumeration constants of the state space are declared under VAR; the transition logic is encoded as `next()` assignments in the ASSIGN block; and the properties are appended as LTLSPEC statements.

nuXmv then explores the state space of the generated model exhaustively. If the model satisfies all LTL properties, verification succeeds; otherwise, nuXmv outputs a counterexample trace that witnesses the violation.

5 Evaluation

The evaluation addresses six questions. **RQ1** (accuracy): how accurately does State-inferer recover implementation-level transitions compared with ProtocolGPT and general-purpose coding agents (Section 5.4)? **RQ2** (representation): which reference transitions lie outside a Mealy-style representation, and how much of the baseline's loss do they explain (Section 5.5)? **RQ3** (context): how does dependency slicing compare with retrieval-based and agentic context construction (Section 5.6)? **RQ4** (properties): how accurately does the LLM specialize meta-properties into implementation-level LTL properties (Section 5.7)? **RQ5** (sensitivity): how sensitive are the results to the slicing depth bounds (Section 5.8) and to the model backend (Section 5.9)? **RQ6** (real-world vulnerabilities): how effectively does State-inferer discover vulnerabilities in unmodified real-world implementations (Section 5.10)?

Table 2 Average token consumption and cost per task.

Task	Avg. tokens	Avg. cost (CNY)
Code extraction	14.5k	0.14
Model construction	51.4k	0.42
Property generation	1.9k	0.02

Tokens: mean prompt + completion tokens per implementation and run for each stage. Cost: GLM-5.1 list price (CNY 6 per million input tokens, 24 per million output tokens; all prompts fall in the $\leq$32K tier).

5.1 Implementation Details and Baseline Setup

All evaluation subjects are analyzed with the same pipeline and configuration, without per-project tuning. Code extraction, model construction, and property generation use the same model (GLM-5.1) and the same prompt templates; slicing parameters are identical across projects, and a fixed retry strategy is triggered only when parsing fails. The data-flow and control-flow depth bounds are set to $k_d = 8$ and $k_c = 5$ for all projects. All LLM calls use temperature 0.0.

In the comparison with ProtocolGPT, Codex, and Claude Code, all methods recover implementation-level state transitions for the same implementation and protocol scope, receive the same protocol information, state-variable hints, and task specification, and are evaluated against the same reference models. All LLM-based methods, including ProtocolGPT and the two coding agents, use GLM-5.1 as the model backend.

All experiments run on a desktop with 64 GB of RAM and an Intel Core i9-14900K processor (24 cores, 32 threads); code slicing (on Joern-generated dependence graphs) and nuXmv verification run locally.

5.2 Benchmarks, Reference Models, and Cost

We evaluate seven open-source implementations of TLS, IKEv2, and SSH (The OpenSSL Project 2026; wolfSSL Inc. 2026; Suica 2024; StrongSwan Project 2026; Libreswan Project 2026; The OpenSSH Project 2026; tinySSH Project 2026). For each implementation, the evaluation focuses on the server-side state machine of a single protocol version at a fixed release (Table 3).

Two domain experts independently constructed a reference state machine for each implementation following the annotation guideline in Appendix B. Their agreement before reconciliation is the strict F-score between the two independently constructed models (Table 3); remaining disagreements were resolved through discussion. The state and transition counts of the final reference models are those in the Code columns of Table 1.

Three stages of State-inferer call the LLM: code extraction, model construction, and property generation. Table 2 reports their average token consumption and cost. Slicing and model checking run locally: the slicer takes 3.2 min per implementation on average, and nuXmv finishes within tens of seconds. Analyzing one implementation takes about 18 min in total.

Table 3 Benchmark releases and reference-model annotation.

Implementation	Release	States	Transitions	Stars	Guarded	Agr.
OpenSSL	4.0.2	20	38	29.7k	14	0.95
WolfSSL	5.8.2	10	16	2.7k	5	0.94
TLSE	–	12	22	659	8	0.95
strongSwan	6.1.0	12	27	2.8k	11	0.96
Libreswan	5.3.1	22	29	927	13	0.97
OpenSSH	10.6	14	21	–	7	0.95
tinySSH	20260601	9	13	1.5k	4	0.92

States/Transitions: reference-model size. Release: evaluated release. Stars: GitHub stars of the project repository. Guarded: reference transitions whose guard depends on internal variables beyond the received message type. Agr.: strict F-score between the two experts' independently constructed models before reconciliation.

5.3 Metrics and Statistical Reporting

Transition representation and alignment. Each transition is a quadruple $T = (s_i, a, g, s_t)$ (Chow 1978) consisting of a source state, an action, a guard, and a target state. Actions distinguish message sends, message receipts, and internal operations. Before matching, the two experts map each method's state names and action labels to the reference vocabulary. The mapping is built from the label list of each output alone, with method names hidden and without access to matching results; distinct semantic states, message types and directions, and internal operations are never merged. Duplicate predictions are counted once.

Guard equivalence. Two guards are equivalent if they admit the same valuations under the variable domains and implementation configuration of the reference model. All guard pairs are judged independently by the two experts (Cohen's $\kappa = 0.87$), and disagreements are resolved through discussion.

Matching criteria. We match predictions $\widehat{\mathcal{T}}$ to reference transitions $\mathcal{R}$ under two nested criteria. A *skeleton match* requires the same source state, action, and target state; a *strict match* additionally requires an equivalent guard. Matching is one-to-one and proceeds in three steps: (1) a maximum matching under the strict criterion yields the true positives (TP); (2) among the remaining items, a maximum matching under the skeleton criterion yields guard errors (E_g), i.e., transitions recovered with the right structure but a wrong guard; (3) among the items still remaining, predictions and reference transitions that share the source state and at least one of the action and the target state are paired one-to-one as skeleton errors (E_s), i.e., transitions with the right origin but a wrong action or a wrong target. Unpaired predictions are spurious (X), and unpaired reference transitions are omitted (O). Each prediction and each reference transition thus falls into exactly one category:

$$N_P = \mathrm{TP} + E_g + E_s + X,$$
$$N_G = \mathrm{TP} + E_g + E_s + O.$$

Metrics. The primary metrics are the strict precision, recall, and F-score,

$$P = \frac{\mathrm{TP}}{N_P}, \qquad R = \frac{\mathrm{TP}}{N_G}, \qquad F = \frac{2PR}{P+R}.$$

Because N_P contains every guard error, skeleton error, and spurious prediction, extra or wrong transitions always lower P. To separate structure from guards, we additionally report (i) the F-score under the skeleton criterion, F_{sk}, computed from $P_{\mathrm{sk}} = (\mathrm{TP} + E_g)/N_P$ and $R_{\mathrm{sk}} = (\mathrm{TP} + E_g)/N_G$, which does not depend on how a method represents guards, and (ii) the guard accuracy $A_g = \mathrm{TP}/(\mathrm{TP} + E_g)$, the share of structurally recovered transitions whose guard is correct. Skeleton metrics matter for methods that report no guards, such as ProtocolGPT (recorded as `true`, Section 5.4): its strict scores include a penalty for that output format, whereas its skeleton scores do not. Reference transitions that lie outside a method's representation (Section 5.5) are marked independently of any prediction and are not a separate outcome class: a prediction that fails to match one is counted as E_g, E_s, X, or O like any other.

Runs and aggregation. Each configuration is evaluated in three runs per implementation. Micro averages pool counts over all implementations within a run; macro averages weight each implementation equally. All reported values are computed from counts summed over the three runs (pooled).

5.4 Extraction Accuracy against ProtocolGPT and Coding Agents

This experiment compares complete pipelines end to end. The primary baseline is ProtocolGPT (Wei et al. 2025), which infers protocol states, message types, and transitions from source code; it is run with its released prompts and retrieval configuration. Each ProtocolGPT transition is mapped to a quadruple whose action is the message type and direction, and whose guard is the reported condition, or `true` when none is reported. Codex and Claude Code serve as general-purpose coding-agent baselines: each agent explores the target repository with its own tools and reports transitions as source states, actions, guards, and target states. The coding agents thus share State-inferer's backend and output format and differ from it only in how the code is selected and analyzed.

Table 4 reports per-implementation counts summed over the three runs, so that every rate can be recomputed from integers. Table 5 reports micro- and macro-averaged metrics, and Table C1 in Appendix C gives per-implementation P, R, and A_g.

Table 5 reports the results. State-inferer reaches a macro strict F of 95.2, against 78.9 for Codex, 77.1 for Claude Code, and 46.2 for ProtocolGPT. On the skeleton criterion (§5.3) the macro F_{sk} is 99.2, 88.6, 88.3, and 67.5 respectively, and the guard accuracy A_g is 95.9, 89.0, 87.4, and 68.5. State-inferer thus leads the two coding agents both in structure (by 10.6–10.9 points of F_{sk}) and in guards, with the same backend and output format. Because ProtocolGPT records absent guards as `true`, its A_g mainly reflects that it does not model guards; its F_{sk} of 67.5 is the fairer measure of its structural accuracy.

Table 4 Transition counts per implementation, summed over three runs.

Implementation	Method	$3N_G$	N_P	TP	E_g	E_s	X	O
OpenSSL	State-inferer	114	114	107	5	0	2	2
	ProtocolGPT		103	48	22	22	11	22
	Codex		112	85	12	10	5	7
	Claude Code		111	83	14	10	4	7
WolfSSL	State-inferer	48	48	45	3	0	0	0
	ProtocolGPT		41	19	9	8	5	12
	Codex		45	35	5	3	2	5
	Claude Code		45	34	6	4	1	4
TLSE	State-inferer	66	67	61	5	0	1	0
	ProtocolGPT		64	30	14	13	7	9
	Codex		69	53	7	6	3	0
	Claude Code		68	52	8	6	2	0
strongSwan	State-inferer	81	82	78	3	0	1	0
	ProtocolGPT		79	37	17	16	9	11
	Codex		84	66	8	7	3	0
	Claude Code		82	64	9	7	2	1
Libreswan	State-inferer	87	85	81	3	0	1	3
	ProtocolGPT		88	41	19	18	10	9
	Codex		89	72	8	6	3	1
	Claude Code		88	70	9	7	2	1
OpenSSH	State-inferer	63	63	61	2	0	0	0
	ProtocolGPT		65	31	14	13	7	5
	Codex		63	50	6	5	2	2
	Claude Code		62	48	7	5	2	3
tinySSH	State-inferer	39	39	39	0	0	0	0
	ProtocolGPT		35	18	8	6	3	7
	Codex		37	31	3	2	1	3
	Claude Code		36	30	3	2	1	4

N_G: reference transitions per run; N_P: predicted transitions; TP: strict matches; E_g: guard errors (skeleton match with a non-equivalent guard); E_s: skeleton errors (same source state and the same action or target, but a different target or action); X: spurious predictions; O: omitted reference transitions. All counts are summed over three runs. $N_P = \mathrm{TP} + E_g + E_s + X$ and $3N_G = \mathrm{TP} + E_g + E_s + O$.

Table 5 Transition recovery metrics: strict match, skeleton match, and guard accuracy (pooled over three runs).

		Strict match			Skeleton	Guard
Average	Method	P	R	F	F_{sk}	A_g
Micro	State-inferer	94.8	94.8	94.8	99.0	95.7
	ProtocolGPT	47.2	45.0	46.0	67.2	68.5
	Codex	78.6	78.7	78.6	88.5	88.9
	Claude Code	77.4	76.5	77.0	88.3	87.2
Macro	State-inferer	95.1	95.2	95.2	99.2	95.9
	ProtocolGPT	47.5	45.0	46.2	67.5	68.5
	Codex	79.0	78.7	78.9	88.6	89.0
	Claude Code	77.9	76.4	77.1	88.3	87.4

Strict match: same source state, action, target state, and an equivalent guard. F_{sk}: F-score when the guard is ignored (skeleton match). $A_g = \mathrm{TP}/(\mathrm{TP} + E_g)$: share of skeleton-matched transitions whose guard is correct. ProtocolGPT reports no guards (recorded as `true`), so its strict scores and A_g include a penalty for that format, whereas F_{sk} does not. All values are percentages computed from counts summed over three runs. Micro: counts pooled over the seven implementations; Macro: per-implementation metrics averaged with equal weight (macro F and F_{sk} are computed from the macro-averaged precision and recall).

The counts in Table 4 locate the remaining errors. ProtocolGPT loses through omissions (O, 75 over three runs), guard errors (E_g, 103), skeleton errors (E_s, 96), and spurious predictions (X, 52): its single-shot extraction over retrieved chunks both over-generates and misses transitions. The coding agents lose on guards and on structure: 49 and 56 guard errors and 39 and 41 skeleton errors (Codex and Claude Code) against 392 and 381 strict matches, with 19 and 14 spurious predictions. State-inferer's residual errors are 21 guard errors, 0 skeleton errors, 5 omissions, and 5 spurious predictions across all seven implementations and three runs; its guard errors are distributed across implementations (OpenSSL 5, TLSE 5, WolfSSL 3, strongSwan 3, Libreswan 3, OpenSSH 2).

5.5 STS versus Mealy Representation

State-inferer builds an STS with explicit internal variables and guards, whereas ProtocolGPT reports states, message types, and transitions in the style of a Mealy machine. A Mealy machine can encode finite internal configurations as states, so what it can express depends on the chosen state abstraction, input alphabet, and observation interface. This experiment measures which reference transitions lie outside the representation that ProtocolGPT actually uses.

Inexpressible transitions. Independently of any prediction, the experts mark the reference transitions that cannot be stated in a method's configured representation and assign each to the first applicable reason in the following order:

- U_{ev} (**internal event**): the transition is triggered by an internal operation, such as a timer expiry or a local error, that is not a message in the method's input alphabet;
- U_{gd} (**internal guard**): the transition shares its source state and input message with another reference transition and is distinguished from it only by a guard over internal variables that are not encoded in the method's states;
- U_{mg} (**merged configuration**): its source or target state is merged with another reference state because the two are indistinguishable at the method's observation interface.

The categories are mutually exclusive, and $U = U_{\text{ev}} + U_{\text{gd}} + U_{\text{mg}}$. These transitions cannot be matched strictly and are always counted in E_g, E_s, or O. Besides the overall recall R, we report the recall on expressible transitions,

$$R_{\text{exp}} = \frac{\text{TP}}{N_G - U},$$

which separates the effect of representational coverage from extraction accuracy; $R_{\text{exp}} = R$ when $U = 0$. The same marking is applied to State-inferer, for which U counts transitions outside its guard language. Codex and Claude Code report the same quadruple format as State-inferer, so their U coincides with that of State-inferer. Table 6 reports U by reason for each implementation, together with ProtocolGPT's R and R_{exp}.

Table 6 shows that most transitions outside ProtocolGPT's representation come from merged configurations (U_{mg}, 30 of 54), followed by internal-guard distinctions

Table 6 Inexpressible reference transitions per implementation by reason.

		ProtocolGPT						State-inferer			
Implementation	N_G	U_{ev}	U_{gd}	U_{mg}	U	R	R_{exp}	U_{ev}	U_{gd}	U_{mg}	U
OpenSSL	38	2	3	7	12	42.1	61.5	0	0	0	0
WolfSSL	16	1	1	3	5	39.6	57.6	0	0	0	0
TLSE	22	1	2	4	7	45.5	66.7	0	0	0	0
strongSwan	27	3	2	5	10	45.7	72.5	0	0	0	0
Libreswan	29	2	2	5	9	47.1	68.3	0	0	0	0
OpenSSH	21	2	1	4	7	49.2	73.8	0	0	0	0
tinySSH	13	1	1	2	4	46.2	66.7	0	0	0	0
Total / macro	166	12	12	30	54	45.0	66.7	0	0	0	0

N_G: reference transitions. U_{ev}: triggered by an internal event outside the input alphabet; U_{gd}: distinguished only by a guard over internal variables not encoded in the states; U_{mg}: endpoint merged with another state at the observation interface; $U = U_{\text{ev}} + U_{\text{gd}} + U_{\text{mg}}$. All U values are marked before any prediction is inspected and are fixed across runs. R, R_{exp}: mean over three runs. Codex and Claude Code share State-inferer's U. Total / macro: counts summed and recall macro-averaged over implementations.

(U_{gd}, 12) and timer-driven internal events (U_{ev}, 12; keepalive and retransmission expiries). ProtocolGPT's recall rises from 45.0 to 66.7 when these transitions are excluded, so about 22 points of its loss are due to the representation and the rest to extraction errors. State-inferer's U is zero on every implementation: every reference transition can be written in its guard language.

5.6 Context Construction

This experiment isolates how the code context supplied to the LLM is built. All variants use the state variables identified by State-inferer, the same model-construction prompt, the STS output format, and the same backend; only the code that accompanies each pair of state-variable assignments differs.

- **Dependency slicing** (State-inferer): the gadgets of Section 4.2.2.
- **RAG context**: the top-k code chunks retrieved by embedding similarity to each state-variable assignment ($k = 10$, chunk size 30 lines, embedding model `Qwen3-Embedding-8B`).
- **Agentic search**: Claude Code explores the repository with its own search and file-reading tools and returns the code it considers relevant to each pair of assignments; this code is then passed to State-inferer's model-construction prompt. Here Claude Code acts only as a retriever and does not report transitions, unlike its end-to-end role in Section 5.4.

Table 7 shows that RAG context reaches a strict F of 70.6 against 95.2 for slicing, with lower precision (70.2 vs. 95.1) and recall (71.1 vs. 95.2). Most of the loss is already structural: RAG's F_{sk} is 75.1 against 99.2, only 4.5 points above its strict F. Retrieval returns fragments that mention the state variable but rarely the call chains that connect assignments across functions, so transitions get wrong endpoints before guards come into play. Agentic search recovers more of these chains because the agent follows the dispatch code itself (strict F 85.1, 10.1 points below slicing), but it consumes 48.2k tokens per implementation, against 0.57k for embedding retrieval and

Table 7 Effect of context construction (macro average over seven implementations, pooled over three runs).

Context	Tokens/impl.	Time/impl.	F_{sk}	P	R	F
Dependency slicing	0	192	99.2	95.1	95.2	95.2
RAG context	570	12.8	75.1	70.2	71.1	70.6
Agentic search (Claude Code)	48200	96.3	87.9	85.4	84.8	85.1

Tokens/impl.: mean tokens consumed by context construction per implementation and run: embedding requests for RAG, and the agent tool-loop calls for agentic search; dependency slicing is pure static analysis and consumes none. Time/impl.: mean wall-clock time (s) of context construction per implementation. P, R, F: strict metrics; F_{sk}: skeleton F-score (Section 5.3).

none for slicing. Slicing is the slowest option in wall-clock time: 192 s per implementation, mostly for the one-time construction of the dependence graph, against 96.3 s for the agent and 12.8 s for RAG. Overall, slicing gives the highest accuracy and uses no LLM tokens, at the cost of a few minutes of static analysis.

5.7 Correctness of LLM-Generated Security Properties

This experiment assesses whether the LTL properties produced by model-aware specialization (Section 4.4) faithfully express the corresponding meta-properties on each implementation. For every implementation and run, State-inferer specializes each meta-property of the target protocol on the extracted model. Independently of the generated properties, the two experts write a reference LTL formula for each meta-property on the reference model of the implementation. Each generated property is then assigned to the first applicable category in the following order:

- **Syntax error**: the formula is rejected by the nuXmv parser;
- **Mapping error**: the formula refers to a state, state variable, or constant that does not exist in the model, or maps the abstract concept of the meta-property to the wrong implementation-level variable or state;
- **Semantic error**: the formula is well formed and correctly mapped, but its logical or temporal structure differs from the reference formula, so that it is strictly weaker, strictly stronger, or incomparable with the intended requirement;
- **Correct**: the formula is equivalent to the reference formula under the variable domains of the reference model.

The categories are mutually exclusive. The classification is performed independently by the two experts, and disagreements are resolved through discussion.

Table 8 shows an overall specialization accuracy of 93.2% over 117 generated formulas (three runs over 39 meta-property/implementation pairs): 109 formulas are equivalent to the expert reference, 4 are semantic errors, 2 are syntax errors, and 2 are mapping errors. Syntax and mapping errors are caught mechanically before model checking. Semantic errors are harder to detect: a formula that is strictly weaker than intended can let a violating trace pass. The workflow therefore includes a manual review of every generated property before model checking.

Table 8 Correctness of generated LTL properties (counts summed over three runs).

Implementation	Meta	Gen.	Syn.	Map.	Sem.	Corr.	Acc.
OpenSSL	5	15	1	0	1	13	86.7
WolfSSL	5	15	0	1	0	14	93.3
TLSE	5	15	0	0	0	15	100.0
strongSwan	6	18	0	1	1	16	88.9
Libreswan	6	18	0	0	1	17	94.4
OpenSSH	6	18	1	0	0	17	94.4
tinySSH	6	18	0	0	1	17	94.4
Total	39	117	2	2	4	109	93.2

Meta: meta-properties of the protocol. Gen.: generated properties over three runs (3×Meta when every meta-property is specialized). Syn./Map./Sem.: syntax, mapping, and semantic errors. Corr.: properties equivalent to the expert reference formula. Gen. = Syn. + Map. + Sem. + Corr. Acc.: Corr./Gen. in percent (pooled over three runs); the Total row pools all implementations.

Table 9 Sensitivity to the slicing depth bounds (macro average over seven implementations, pooled over three runs).

Setting	Gadget LOC	Functions	P	R	F	X
k_d=2, k_c=5	42.6	6.8	90.8	89.4	90.1	0.5
k_d=4, k_c=5	87.9	11.2	93.8	92.5	93.1	0.3
k_d=8, k_c=5 (default)	151.3	16.0	95.1	95.2	95.2	0.2
k_d=12, k_c=5	193.8	18.9	95.1	95.1	95.1	0.2
k_d=16, k_c=5	225.4	20.7	95.0	95.0	95.0	0.2
k_d=8, k_c=3	103.6	12.1	92.9	91.5	92.2	0.3
k_d=8, k_c=5 (default)	151.3	16.0	95.1	95.2	95.2	0.2
k_d=8, k_c=8	212.7	19.4	95.2	95.1	95.2	0.2

Upper block: k_d varied with k_c fixed; lower block: k_c varied with k_d fixed. Gadget LOC: mean lines per gadget (one gadget is composed per transition anchor). Functions: mean number of distinct functions involved in the gadgets per implementation. X: spurious predictions per implementation and run.

5.8 Sensitivity to Slicing Depth

The depth bounds k_d and k_c trade slice completeness against slice size. We vary one bound at a time over $k_d \in \{2, 4, 8, 12, 16\}$ or $k_c \in \{3, 5, 8\}$ while fixing the other at its default value (Section 5.1), run each setting three times on all seven implementations, and report gadget size, the number of functions involved, and strict metrics (Table 9).

Macro F saturates at the default k_d=8: raising k_d from 2 to 8 lifts F from 90.1 to 95.2, while k_d=12 and k_d=16 add nothing. Beyond saturation, gadget size keeps growing (43 to 225 lines per gadget from k_d=2 to 16). k_c behaves similarly: F rises from 92.2 at k_c=3 to 95.2 at the default k_c=5, and k_c=8 brings no further gain while growing gadgets to 212.7 lines.

5.9 Sensitivity to Model Backend

To test whether the results depend on GLM-5.1, we rerun the complete pipeline with two models from other providers: GPT-5.5 (`gpt-5.5`) and Claude Opus 4.8 (`claude-opus-4.8`); the default backend is GLM-5.1 (`glm-5.1`). Prompts, slicing parameters, decoding settings, and reference models are unchanged, and each backend

Table 10 Extraction accuracy of State-inferer with different model backends (macro average, pooled over three runs).

Backend	P	R	F	F_{sk}
GLM-5.1 (default)	95.1	95.2	95.2	99.2
GPT-5.5	96.3	96.5	96.4	99.6
Claude Opus 4.8	95.8	96.0	95.9	99.5

P, R, F: strict metrics; F_{sk}: skeleton F-score. The GLM-5.1 row is the State-inferer macro row of Table 5.

is run three times on all seven implementations. Table 10 reports the macro-averaged extraction metrics.

Table 10 shows macro strict F values of 95.2 (GLM-5.1), 96.4 (GPT-5.5), and 95.9 (Claude Opus 4.8). The three backends lie within 1.2 points of each other, and F_{sk} stays between 99.2 and 99.6. The two stronger models give a small gain over the default, so the results reported above do not depend on GLM-5.1.

5.10 Real-World Vulnerability Discovery

This experiment applies State-inferer to unmodified protocol implementations to find vulnerabilities through model checking.

State-inferer found four previously unknown vulnerabilities, all reported to the maintainers through responsible disclosure. The WolfSSL vulnerability has been assigned CVE-2026-2645, and the three TLSE vulnerabilities CNVD-2026-27960, CNVD-2026-27959, and CNVD-2026-27957. Each violation was reproduced on the unmodified implementation with a concrete message sequence derived from the nuXmv counterexample, independently of the extracted model. We describe root causes and impact below and withhold exploitation details.

1. **Authentication State-Machine Bypass in WolfSSL TLS 1.2** (CVE: CVE-2026-2645)

   The transition-level evaluation uses the TLS 1.3 server of WolfSSL, for which a reference model is available; in the vulnerability-discovery stage, State-inferer is additionally applied to its TLS 1.2 server, where this vulnerability was found. In the TLS 1.2 server of WolfSSL 5.8.2, State-inferer found a non-compliant transition in the receive path. The guard of the CertificateVerify handler checks only that the Certificate message is non-empty and unique; it does not require that a ClientKeyExchange (CKE) message has been received. An attacker can therefore send CertificateVerify before key exchange is complete, advance the handshake state machine, and bypass the subsequent certificate verification.
2. **Extension Parsing Error Causing Protocol Version Confusion in TLSE** (CNVD-2026-27960)

   When TLSE receives a TLS 1.3 ClientHello whose `key_share` extension is present but empty, it does not take the HelloRetryRequest transition. No branch handles this case, and control falls through to the TLS 1.2 server branch. A server in a TLS 1.3 handshake then sends TLS 1.2 messages such as ServerKeyExchange, which confuses the protocol state and violates RFC 8446 (Rescorla 2018).

3. **Missing State Variable Leading to Unbounded CCS Processing in TLSE** (CNVD-2026-27959)

   The extracted TLSE model shows no constraint that ChangeCipherSpec (CCS) is accepted only once. TLSE controls the handshake only through the global `connection_status` variable and keeps no flag recording that a CCS has been received. The server therefore accepts any number of consecutive CCS messages, contrary to TLS semantics, which opens the way to state exhaustion and message injection.
4. **Alert Handling Desynchronized from Transport-Layer State in TLSE** (CNVD-2026-27957)

   RFC 8446 (Rescorla 2018) requires a party that sends a `close_notify` alert to close the write side of the connection and send no further data. The `tls_close_notify` routine in TLSE sends the alert and sets the `critical_error` flag but does not close the socket. Later I/O dispatching does not check this flag, so the TLS state and the TCP connection diverge: the peer can keep sending data after the alert, and resources such as file descriptors are not released.

5.11 Discussion and Threats to Validity

Applicability of other inference approaches.

Besides ProtocolGPT, automata-learning frameworks such as LearnLib (Isberner et al. 2015), NGLL (Bauer et al. 2016), Libalf (Bollig et al. 2010), Tomte (Aarts et al. 2015), ROLL (Li et al. 2021), and Scikit-SpLearn (Arrivault et al. 2017), and the protocol inspection tool MPInspector (Wang et al. 2021), can also produce state machines. They are not built to recover state machines uniformly across protocols and code bases, so we do not compare against them directly: each new implementation needs manual adaptation (e.g., test adapters, abstract alphabets, or initial symbol tables), and active learning becomes much more expensive as the input–output space grows (Qin et al. 2023; Zhao et al. 2025). Methods that extract state machines from RFCs or other natural-language specifications, such as Hermes (Al Ishtiaq et al. 2024) and RFCNLP (Pacheco et al. 2022), produce specification-level models and address a different goal from the implementation-level models targeted here.

External validity.

The evaluation covers TLS, IKEv2, and SSH. Other protocol stacks, event-driven frameworks, and proprietary industrial implementations remain to be tested. The evaluation also covers only the server side of a single protocol version per implementation; client-side logic and other versions are not assessed.

Internal validity.

The reference models are built manually and reflect expert judgment; two experts annotated independently and reconciled their differences, but some ambiguity remains. The baselines are evaluated only with GLM-5.1, and every configuration is run three times; State-inferer is additionally evaluated with two other backends (Section 5.9), but results may differ with other backends or decoding settings. The default depth

bounds k_d=8 and k_c=5 were chosen from the sensitivity analysis on the same subjects (Section 5.8) and may slightly favor these seven implementations; F, however, is flat around the defaults, and larger bounds do not change the results.

Construct validity.

Transition-level matching measures agreement with the reference model edge by edge. It does not establish trace equivalence or long-run behavioral equivalence, and guard pairs outside the decidable fragment rely on expert judgment. In implementations with deeply nested calls, heavy macro expansion, or dynamic dispatch, slicing may still miss guard conditions.

6 Conclusion

This paper presented State-inferer, a white-box framework that combines program slicing, LLMs, and model checking to extract and verify protocol state machines. State-inferer builds the context for each transition by program slicing instead of similarity-based retrieval, models internal variables and guards in a state transition system, and specializes RFC security requirements into LTL properties over that system. On seven open-source protocol implementations, it achieves a macro-average strict F of 95.2, against 78.9 for the strongest baseline, and found four previously unknown vulnerabilities, all assigned CVE or CNVD identifiers.

Building reference models by hand remains costly. Future work will extend the analysis to deep call chains, complex macros, and asynchronous control flow, and will automate parts of reference-model construction and property alignment so that larger implementations can be analyzed.

Data availability. No datasets were generated or analysed during the current study.

Declarations

Ethical approval. Not applicable.

Competing interests. The authors declare no competing interests.

References

Aarts F, De Ruiter J, Poll E (2013) Formal models of bank cards for free. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, IEEE, pp 461–468

Aarts F, Fiterău-Broştean P, Kuppens H, et al. (2015) Learning register automata with fresh value generation. In: Theoretical Aspects of Computing – ICTAC 2015: 12th International Colloquium, Cali, Colombia, October 29-31, 2015, Proceedings, Springer, pp 165–183

Al Ishtiaq A, Das SSS, Rashid SMM, et al. (2024) Hermes: Unlocking security analysis of cellular network protocols by synthesizing finite state machines from natural

language specifications. In: 33rd USENIX Security Symposium (USENIX Security 24), pp 4445–4462

Angluin D (1987) Learning regular sets from queries and counterexamples. Information and computation 75(2):87–106

Arrivault D, Benielli D, Denis F, et al. (2017) Scikit-SpLearn: a toolbox for the spectral learning of weighted automata compatible with scikit-learn. In: Actes de la 19e conférence sur l'Apprentissage Automatique, CAP 2017

Ba J, Böhme M, Mirzamomen Z, et al. (2022) Stateful greybox fuzzing. In: 31st USENIX Security Symposium (USENIX Security 22), pp 3255–3272

Bauer O, Neubauer J, Isberner M (2016) Model-driven active automata learning with LearnLib studio. In: Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications – 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II, Springer, pp 128–142

Beurdouche B, Bhargavan K, Delignat-Lavaud A, et al. (2015) A messy state of the union: taming the composite state machines of TLS. In: 2015 IEEE Symposium on Security and Privacy (SP), IEEE, pp 535–552, https://doi.org/10.1109/SP.2015.39

Bhargavan K, Lavaud AD, Fournet C, et al. (2014) Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In: 2014 IEEE Symposium on Security and Privacy, IEEE, pp 98–113

Bollig B, Katoen JP, Kern C, et al. (2010) libalf: The automata learning framework. In: Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings, Springer, pp 360–364, https://doi.org/10.1007/978-3-642-14295-6_32

Cavada R, Cimatti A, Dorigatti M, et al. (2014) The nuXmv symbolic model checker. In: Computer Aided Verification – 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, Springer, pp 334–342, https://doi.org/10.1007/978-3-319-08867-9_22

Chen Y, Mei Y, Lan T, et al. (2024) Exploring effective fuzzing strategies to analyze communication protocols. Digital Threats: Research and Practice 5(1):1–22

Chow TS (1978) Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering 4(3):178–187

Daniele C, Andarzian SB, Poll E (2024) Fuzzers for stateful systems: Survey and research directions. ACM Computing Surveys 56(9):1–23

De Ruiter J, Poll E (2015) Protocol state fuzzing of TLS implementations. In: 24th USENIX Security Symposium (USENIX Security 15), pp 193–206

Fiterău-Broştean P, Howar F (2017) Learning-based testing the sliding window behavior of TCP implementations. In: Critical Systems: Formal Methods and Automated Verification: Joint 22nd International Workshop on Formal Methods for Industrial Critical Systems and 17th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS 2017, Turin, Italy, September 18–20, 2017, Proceedings, Springer, pp 185–200

Fiterău-Broştean P, Janssen R, Vaandrager F (2016) Combining model learning and model checking to analyze TCP implementations. In: International Conference on Computer Aided Verification, Springer, pp 454–471

Fiterău-Broştean P, Lenaerts T, Poll E, et al. (2017) Model learning and model checking of SSH implementations. In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, pp 142–151

Fujiwara S, Bochmann GV, Khendek F, et al. (1991) Test selection based on finite state models. IEEE Transactions on Software Engineering 17(6):591–603

Guo J, Gu C, Chen X, et al. (2019) Model learning and model checking of IPsec implementations for Internet of Things. IEEE Access 7:171322–171332

Horwitz S, Reps T, Binkley D (1990) Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems (TOPLAS) 12(1):26–60

Howar F, Steffen B, Jonsson B, et al. (2012) Inferring canonical register automata. In: Verification, Model Checking, and Abstract Interpretation: 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012, Proceedings, Springer, pp 251–266

Isberner M, Howar F, Steffen B (2015) The open-source LearnLib: A framework for active automata learning. In: Computer Aided Verification – 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II, Springer, pp 487–495, https://doi.org/10.1007/978-3-319-21690-4_32

Jackson D, Rollins EJ (1994) Chopping: A generalization of slicing. Tech. Rep. CMU-CS-94-169, Carnegie Mellon University

Kleber S, Kopp H, Kargl F (2018) NEMESYS: Network message syntax reverse engineering by analysis of the intrinsic structure of individual messages. In: 12th USENIX Workshop on Offensive Technologies (WOOT 18)

Kleber S, van der Heijden RW, Kargl F (2020) Message type identification of binary network protocols using continuous segment similarity. In: IEEE INFOCOM 2020-IEEE conference on computer communications, IEEE, pp 2243–2252

Lewis P, Perez E, Piktus A, et al. (2020) Retrieval-augmented generation for knowledge-intensive NLP tasks. In: Advances in Neural Information Processing Systems (NeurIPS), pp 9459–9474

Li Y, Chen YF, Zhang L, et al. (2021) A novel learning algorithm for Büchi automata based on family of DFAs and classification trees. Information and Computation 281:104678

Li Z, Dutta S, Naik M (2025) IRIS: LLM-assisted static analysis for detecting security vulnerabilities. In: International Conference on Learning Representations, pp 35735–35758

Libreswan Project (2026) Libreswan: Free Software implementation of IPsec and IKE. https://libreswan.org/, accessed: 2026-10-14

McMahon Stone C, Thomas SL, Vanhoef M, et al. (2022) The closer you look, the more you learn: A grey-box approach to protocol state machine learning. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pp 2265–2278

Natella R (2022) StateAFL: Greybox fuzzing for stateful network servers. Empirical Software Engineering 27(7):191

Pacheco ML, von Hippel M, Weintraub B, et al. (2022) Automated attack synthesis by extracting finite state machines from protocol specification documents. In: 2022 IEEE Symposium on Security and Privacy (SP), IEEE, pp 51–68

Pham VT, Böhme M, Roychoudhury A (2020) AFLNet: A greybox fuzzer for network protocols. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), IEEE, pp 460–465

Pnueli A (1977) The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (FOCS), IEEE, pp 46–57, https://doi.org/10.1109/SFCS.1977.32

Qin S, Hu F, Ma Z, et al. (2023) NSFuzz: Towards efficient and state-aware network service fuzzing. ACM Transactions on Software Engineering and Methodology 32(6):1–26

Rescorla E (2018) The transport layer security (TLS) protocol version 1.3. RFC 8446, IETF, URL https://www.rfc-editor.org/rfc/rfc8446

de Ruiter J (2016) A tale of the OpenSSL state machine: A large-scale black-box analysis. In: Secure IT Systems: 21st Nordic Conference, NordSec 2016, Oulu, Finland, November 2-4, 2016, Proceedings, Springer, pp 169–184

Shi Q, Xu X, Zhang X (2023) Extracting protocol format as state machine via controlled static loop analysis. In: 32nd USENIX security symposium (USENIX Security 23), pp 7019–7036

StrongSwan Project (2026) StrongSwan: Open Source IPsec Implementation for Linux. https://www.strongswan.org/, accessed: 2026-10-14

Suica E (2024) TLSE: TLS library in C. https://github.com/eduardsui/tlse

Tappler M, Aichernig BK, Bloem R (2017) Model-based testing IoT communication via active automata learning. In: 2017 IEEE International conference on software testing, verification and validation (ICST), IEEE, pp 276–287

The OpenSSH Project (2026) OpenSSH: OpenBSD Secure Shell. https://www.openssh.com/, accessed: 2026-10-14

The OpenSSL Project (2026) OpenSSL: Cryptography and SSL/TLS Toolkit. https://www.openssl.org/, accessed: 2026-10-14

tinySSH Project (2026) tinySSH: Minimal SSH server. https://tinyssh.org/, accessed: 2026-10-14

Wang Q, Ji S, Tian Y, et al. (2021) MPInspector: A systematic and automatic approach for evaluating the security of IoT messaging protocols. In: 30th USENIX Security Symposium (USENIX Security 21), pp 4205–4222

Wei H, Chen L, Du Z, et al. (2025) Unleashing the power of llm to infer state machine from the protocol implementation. In: 2025 IEEE/ACM 33rd International Symposium on Quality of Service (IWQoS), pp 1–10, https://doi.org/10.1109/IWQoS65803.2025.11143461

Weiser M (1984) Program slicing. IEEE Transactions on Software Engineering 10(4):352–366

wolfSSL Inc. (2026) wolfSSL: Embedded SSL/TLS Library. https://www.wolfssl.com/, accessed: 2026-10-14

Yamaguchi F, Golde N, Arp D, et al. (2014) Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy (SP), IEEE, pp 590–604

Ye Y, Zhang Z, Wang F, et al. (2021) NetPlier: Probabilistic network protocol reverse engineering from message traces. In: NDSS

Yuan B, Song Z, Jia Y, et al. (2024) MQTTactic: Security analysis and verification for logic flaws in MQTT implementations. In: 2024 IEEE Symposium on Security and Privacy (SP), IEEE, pp 2385–2403

Zhang Z, Chen C, Liu B, et al. (2023) Unifying the perspectives of NLP and software engineering: A survey on language models for code. arXiv preprint arXiv:2311.07989, URL https://arxiv.org/abs/2311.07989

Zhao D, Guo J, Gu C, et al. (2025) AGLFuzz: Automata-guided fuzzing for detecting logic errors in security protocol implementations. Computers & Security 149:103979

Appendix A Security Meta-Properties and LTL Specialization Examples

This appendix uses TLS as an example. It presents the RFC specification basis (Rescorla 2018) for the five TLS security meta-properties used in this paper, along with examples of specialized LTL expressions derived for the OpenSSL and WolfSSL state-machine models, as shown in Table A1.

Table A1 Security meta-properties and LTL specialization examples (**O** = OpenSSL, **W** = WolfSSL).

ID	RFC description	Specialized LTL expressions
SP1	Complete handshake before application data exchange	**O:** `G (hand_state = TLS_ST_OK → s3.tmp.key_block ≠ NULL)` **W:** `G (serverState = SERVER_FINISHED_ACKED → msgsReceived.got_change_cipher = 1)`
SP2	No cipher change before key agreement complete	**O:** `G (hand_state = TLS_ST_SR_CHANGE ∨ hand_state = TLS_ST_CR_SRVR_DONE → s3.tmp.new_cipher ≠ NULL)` **W:** `G (serverState = SERVER_CHANGECIPHERSPEC_COMPLETE → msgsReceived.got_server_hello_done = 1 ∨ msgsReceived.got_client_key_exchange = 1)`
SP3	Initial handshake must get valid response or safe termination	**O:** `G (hand_state = TLS_ST_SR_CLNT_HELLO → F (server_hello_sent = 1 ∨ ssl_error ≠ 0))` **W:** `G (serverState = NULL_STATE → F (server_hello_sent = 1 ∨ ret ≠ 0))`
SP4	Finished message must follow cipher change	**O:** `G (hand_state = TLS_ST_SR_FINISHED ∨ hand_state = TLS_ST_CR_FINISHED → s3.tmp.key_block ≠ NULL)` **W:** `G (serverState = SERVER_FINISHED_COMPLETE → msgsReceived.got_change_cipher = 1)`
SP5	No handshake processing after abnormal termination	**O:** `G (statem.state = MSG_FLOW_ERROR → G (ssl_error ≠ 0))` **W:** `G (serverState = SERVER_CLOSED → G (options.isClosed = 1))`

The mappings in Table A1 are directly instantiated from RFC-level requirements and implementation-level state variables. They provide concrete LTL constraints used by the model checker in our verification pipeline.

Appendix B Annotation Guideline

This appendix summarizes the annotation guideline used by the two experts to construct the reference models and to judge transition matches.

Table C1 Per-implementation strict precision, recall, and guard accuracy (pooled over three runs).

	State-inferer			ProtocolGPT			Codex			Claude Code		
Implementation	P	R	A_g	P	R	A_g	P	R	A_g	P	R	A_g
OpenSSL	93.9	93.9	95.5	46.6	42.1	68.6	75.9	74.6	87.6	74.8	72.8	85.6
WolfSSL	93.8	93.8	93.8	46.3	39.6	67.9	77.8	72.9	87.5	75.6	70.8	85.0
TLSE	91.0	92.4	92.4	46.9	45.5	68.2	76.8	80.3	88.3	76.5	78.8	86.7
strongSwan	95.1	96.3	96.3	46.8	45.7	68.5	78.6	81.5	89.2	78.0	79.0	87.7
Libreswan	95.3	93.1	96.4	46.6	47.1	68.3	80.9	82.8	90.0	79.5	80.5	88.6
OpenSSH	96.8	96.8	96.8	47.7	49.2	68.9	79.4	79.4	89.3	77.4	76.2	87.3
tinySSH	100.0	100.0	100.0	51.4	46.2	69.2	83.8	79.5	91.2	83.3	76.9	90.9

Strict $P = \mathrm{TP}/N_P$, strict $R = \mathrm{TP}/N_G$, and guard accuracy $A_g = \mathrm{TP}/(\mathrm{TP} + E_g)$, all in percent, computed from the counts in Table 4 summed over three runs.

- **Scope.** Each reference model covers the server side of one protocol version at the release listed in Table 3, under the default build configuration (GCC 13.2, `-O2`).
- **States.** A state is a value of the protocol state variables identified in the source code. Distinct enumeration values are kept as distinct states; values that the code never assigns on the server path are excluded.
- **Actions.** Each transition carries one action: receipt of a message, sending of a message, or an internal operation (e.g., key derivation, certificate verification, timeout). Message actions record the message type and direction.
- **Guards.** A guard is a Boolean expression over the code variables that must hold for the transition to fire, written with the variable names used in the implementation. Conditions that only check the received message type are encoded by the action, not repeated in the guard.
- **Transitions.** A transition is a quadruple (source state, action, guard, target state). Transitions with identical source state, action, target state, and equivalent guards are merged.
- **Alignment.** Each method's state names and action labels are mapped to the reference vocabulary from its label list alone, with method names hidden and before any matching. Labels with different meanings are never mapped to the same reference label.
- **Guard equivalence.** Guards in quantifier-free linear integer arithmetic are compared with Z3; other guard pairs are judged by both experts.
- **Representational coverage.** For each method, the experts mark the reference transitions that fall outside its configured representation, before inspecting any of its predictions.
- **Disagreements.** The experts annotate independently and resolve disagreements through discussion.

Appendix C Per-Implementation Results

This appendix reports strict precision, recall, and guard accuracy for each implementation, complementing the micro and macro averages in Table 5. All values are computed from the matching counts in Table 4, summed over the three runs.