ProInspector: Uncovering Logical Bugs in Protocol Implementations

Zichao Zhang
Carnegie Mellon University
Pittsburgh, USA
zichaozhang@cmu.edu

Limin Jia
Carnegie Mellon University
Pittsburgh, USA
liminjia@cmu.edu

Corina Păsăreanu
Carnegie Mellon University
Mountain View, USA
pcorina@cmu.edu

2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P) | 979-8-3503-5425-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/EUROSP60621.2024.00040

***Abstract*—Cryptographic protocols play a crucial role in safeguarding network communications. However, it has been shown that many design flaws and implementation bugs in cryptographic protocols lie in plain sight, only to be discovered many years after their deployment. At the design level, symbolic protocol provers, such as Tamarin and ProVerif, assume a symbolic or Dolev-Yao attacker model and are shown to be effective in ruling out logical errors in protocol specifications. However, little work has been done on automatically analyzing such security guarantees (secure under Dolev-Yao) for an existing protocol implementation.**

We present an automated and systematic framework, ProInspector, to uncover logic errors in protocol implementations. Central to our approach is a tailored conformance testing algorithm which generates test cases, taking into consideration, a Dolev-Yao attacker. Our approach enables us to generate test cases that contain inputs from the attacker. ProInspector then uses generic symbolic provers to check if inconsistencies between the specification and implementation lead to exploits. We test ProInspector on popular TLS implementations and rediscover several CVEs.

1. Introduction

Cryptographic protocols use cryptographic primitives, such as encryption and signatures, to secure communication on an insecure network. The high-level security goals of these protocols include confidentiality, authentication, and forward secrecy. Protocols such as TLS [18, 38] are crucial for securing software systems and services and are widely deployed across various domains, including e-commerce, online banking, and mobile communication. However, it is extremely difficult to ensure the correctness of both the design and the implementation of these protocols, as evidenced by numerous high-profile attacks [8, 17, 36]. Consequently, there is a pressing need for formal verification and rigorous testing of these protocols to enhance their reliability and security.

Over the years, the research community has developed several approaches to analyzing the design and the implementation of cryptographic protocols (see [7] for a survey). These methodologies and tools cover a wide range of the problem space; they differ in the target of the analysis: protocol design vs. implementation; the attacker model considered: symbolic Dolev-Yao network attacker vs. computational attackers; and analysis method used: formal verification vs. testing.

At the design level, symbolic protocol analysis aims at ruling out logical bugs in protocol designs where the messages in the protocol are represented symbolically as terms (rather than bitstrings). The protocol is assumed to be executed in the presence of an active attacker who has complete control over the network but is constrained to perfect black-box assumptions about the cryptographic primitives (e.g., the adversary cannot decrypt a message without the corresponding key). This symbolic attacker model is commonly referred-to as the Dolev-Yao attacker [19]. Symbolic approximations allow us to reason about sophisticated protocol designs automatically using tools like ProVerif [13] or Tamarin [31]. Notably, these symbolic verification tools have been employed during the standardization of TLS 1.3 [10, 16] to verify its resilience against logical attacks such as downgrade [3] and authentication attacks [11] that had impacted previous versions of TLS. However, it is important to note that symbolic analysis operates on abstract specifications of the protocol. The security guarantees achieved at the design level are decoupled from the deployed protocol implementations. Even if the protocol specification is verified to satisfy all desired security properties, the implementation bugs may undermine the security guarantees.

Ensuring that the implementation faithfully aligns with the specification is a challenging task. Protocol specifications often span hundreds of pages and encompass various modes and interoperability considerations for legacy versions. It is the responsibility of developers to comprehend the specification and faithfully translate it into code. Additionally, implementations are typically written in diverse programming languages and have to accommodate specific hardware requirements and efficiency concerns. Furthermore, specific applications may need to tweak the specification to suit their particular usage scenarios. Thus, while symbolic analysis can effectively verify that a given protocol specification is free from logical flaws, software developers may reintroduce such errors while implementing the protocols. Indeed, numerous implementation-level logical attacks [8, 17, 36] have been found in real-world protocol implementations such as TLS, which could have resulted in devastating consequences. For example, the SKIP attack [8] found in an early version of Java Secure Socket Extension (JSSE) allows a network attacker to impersonate any server to a client running JSSE TLS and read confidential application data.

Checking these implementation-level logical errors that violate security guarantees (as opposed to crashes due to unsafe memory operations) is particularly difficult with

DOI 10.1109/EuroSP60621.2024.00040

off-the-shelf program analysis tools such as model checking, symbolic verification, and fuzzing. This difficulty arises primarily due to the lack of a reference protocol state-machine model and a dedicated attacker model (the Dolev-Yao attacker) tailored to protocol implementations.

Prior work has applied model learning to identifying protocol implementation flaws by extracting and analyzing high-level models from low-level implementations [17, 20, 37, 36, 22, 4]. However, these works are hard to scale since they involve manual analysis to determine whether the implementation flaws that deviate from the protocol specification violate any high-level security goals. On the other hand, few automated tools for analyzing security properties can be directly applied to analyzing existing protocol implementations. They either require protocols to be developed in a new language [9]; or work only at the specification level with symbolic terms [13, 31]. In this paper, we aim to bridge this gap and investigate how to identify logical errors in protocol implementations that lead to security property violations.

More specifically, we propose PROINSPECTOR, a novel automated and systematic testing framework that combines conformance testing of protocol implementations against a given reference specification and automated symbolic protocol analysis tools such as ProVerif [13] and Tamarin [31]. At the heart of our framework, there is component that models a Dolev-Yao attacker and connects the two analyses. Traditional conformance testing such as the Wp-method [23] does not take into consideration Dolev-Yao attackers and thus cannot be directly used to generate test cases that involve inputs from an attacker. On the other hand, symbolic protocol analysis tools operates at a higher-level of abstraction and do not understand the concrete format of the network packets that the implementation inputs and outputs.

On one side of PROINSPECTOR's pipeline, a conformance testing tool tests a protocol implementation against a protocol specification represented as a finite state machine. We augment the conformance testing algorithm to perform fine-grained mutations at the protocol message level based on the Dolev-Yao assumptions. We generate a suite of test cases, with each test case consisting of a list of inputs/stimuli and a list of expected outputs/observations. We then evaluate the conformance of the Implementation Under Test (IUT) with these test cases. On the other side of the pipeline, any non-conforming test cases are automatically translated into the domain-specific language of an automated symbolic protocol analysis tool such as ProVerif [13]. This allows us to effectively search for potential vulnerabilities and exploits caused by nonconforming execution traces. Further, we implement a mapper module to translate between abstract messages that the symbolic protocol analysis tool understands and network packets that the implementation inputs and outputs.

To demonstrate the effectiveness of our approach, we instantiate PROINSPECTOR with ProVerif as the symbolic analysis tool and analyze several versions of *wolfSSL* and *OpenSSL*, implementing TLS 1.3. PROINSPECTOR successfully rediscovers several logic bugs.

This paper makes the following contributions:

- We introduce PROINSPECTOR, a systematic and automated framework for uncovering logical bugs in protocol implementations by connecting existing automated symbolic protocol analysis tools with conformance testing and
- We extend the orginal Wp-method with a new module that can generate concrete test traces for cryptographic protocols with a built-in Dolev-Yao attacker model.
- We apply our methodology on popular TLS 1.3 implementations and confirm existing security vulnerabilities.
- PROINSPECTOR is open-sourced and available at https://github.com/proj-proinspector/proinspector

The rest of this paper proceeds as follows. We first provide necessary background in Section 2. Section 3 presents an overview of our testing framework. Using a simple protocol as an example, we explain PROINSPECTOR's novel conformance testing module in Section 4 and PROINSPECTOR's pipeline for identifying logic errors using the symbolic protocol verifier in Section 5. We then demonstrate the effectiveness of our framework by analyzing real-world implementations of TLS 1.3 in Section 6. Section 7 discusses related work. We discuss future work and conclude in Section 8.

2. Background

We review the definitions of Mealy machines, conformance testing, and Dolev-Yao attacker, and set up the Needham-Schroeder-Lowe protocol example.

2.1. Mealy Machine

Mealy machines are finite-state machines that distinguish between input and output actions and are a popular modeling formalism for reactive systems. We use Mealy machines to describe the reference behavior model of protocols. Formally a Mealy machine $\mathcal{M}$ is a 6-tuple $\langle Q, q_0, I, O, \delta, \lambda \rangle$ where Q is a finite set of states, $q_0 \in Q$ is the start state, $I = \{i_1, i_2, ..., i_n\}$ and $O = \{o_1, o_2, ..., o_m\}$ are finite sets of input and output symbols respectively, $\delta : Q \times I \to Q$ is the state transition function, and $\lambda : Q \times I \to O$ is the output function.

Definition 2.1 (Trace)**.** A trace t in a Mealy machine $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda \rangle$, is a sequence of input-output pairs $t = (in_0, out_0)...(in_n, out_n) \in (I, O)^*$ such that, starting at the initial state, the observed outputs $out_0...out_n$ are the result of simulating $\mathcal{M}$ with the sequence of inputs $in_0...in_n$, i.e., $\lambda(q_0, in_0) = out_0, \lambda(\delta(q_0, in_0), in_1) = out_1, ..$ and so on. We write $t \in \mathcal{M}$ if t is a trace in $\mathcal{M}$ and otherwise $t \notin \mathcal{M}$. We use $t \cdot t'$ to denote the concatenation of two traces $t, t' \in (I, O)^*$; $|t|$ for the length of t, which is the number of input-output pairs in t, and $t[n]$ to represent the n^{th} input-output pair (in_n, out_n) on the trace t starting at index 0. We write *.in* and *.out* to denote the projections on inputs and outputs, respectively; for example, $t.in$ refers to an ordered list of inputs on the trace and $t[n].out$ refers to the n^{th} output on the trace.

2.2. Conformance Testing

Conformance testing is a well-known technique for checking if an implementation conforms to a specification.

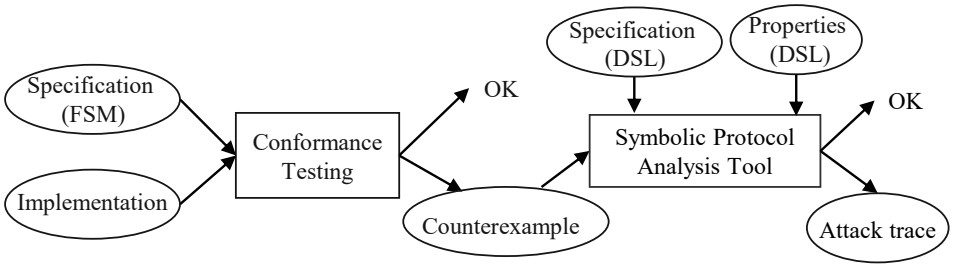


Figure 1: PROINSPECTOR architecture overview.

In this testing approach, the implementation is considered a black-box and the specification is typically modeled using a finite-state automaton such as a Mealy machine. An implementation $\mathcal{I}$ is said to conform to its specification $\mathcal{S}$ if it implements the behavior defined in the specification. Following Tretmans [41], we define the implementation relation $\mathcal{I}$ **imp** $\mathcal{S}$ to mean $\mathcal{I}$ conforms to $\mathcal{S}$ and conversely $\mathcal{I}$ $\overline{\textbf{imp}}$ $\mathcal{S}$. Testing is used to check the **imp** relation where a test suite $T_\mathcal{S}$ consists of test cases generated from the specification. If the implementation $\mathcal{I}$ conforms to the behavior of a test case $t \in T_\mathcal{S}$ where given $t.in$ the outputs of the implementation are $t.out$, we say $\mathcal{I}$ **passes** t and otherwise $\mathcal{I}$ **fails** t. The **imp** relation is satisfied if **passes** relation is true for every test case in the test suite. Formally,

$$\mathcal{I} \textbf{ imp } \mathcal{S} \Leftrightarrow \forall t \in T_S : \mathcal{I} \textbf{ passes } t$$

We formalize the specification $\mathcal{S}$ as a Mealy machine $\mathcal{M}_\mathcal{S}$ and we aim to check $\mathcal{I}$ **imp** $\mathcal{M}_\mathcal{S}$, defined as follows:

$$\mathcal{I} \textbf{ imp } \mathcal{M}_\mathcal{S} \Leftrightarrow \forall t \in T_{\mathcal{M}_\mathcal{S}} : \mathcal{I} \textbf{ passes } t$$

where $T_{\mathcal{M}_\mathcal{S}}$ is a test suite generated from the Mealy machine $\mathcal{M}_\mathcal{S}$.

Several test suite generation algorithms have been developed that operate on a Mealy machine specification, such as W-method [14], Wp-method [23] and Random Walks [27] to approximate an ideal complete test suite (which is infeasible in practice). We choose the Wp-method which has a parameterized bound n. We write $\mathcal{I}$ $\textbf{imp}_n$ $\mathcal{M}_\mathcal{S}$ to mean that $\mathcal{I}$ implements the behavior defined in $\mathcal{M}_\mathcal{S}$ under the assumption that $\mathcal{I}$'s behavior can be represented in a Mealy machine of at most n states, defined below:

$$\mathcal{I} \textbf{ imp}_n \mathcal{M}_\mathcal{S} \Leftrightarrow \forall t \in T_{\mathcal{M}_\mathcal{S},n} : \mathcal{I} \textbf{ passes } t$$

where $T_{\mathcal{M}_\mathcal{S},n}$ is the test suite that the Wp-method with bound n generates, consisting of a set of traces (Definition 2.1) of the Mealy machine $\mathcal{M}_\mathcal{S}$. We estimate n to be the same as the number of states in the reference Mealy machine. We show with our case study that this estimation is enough to uncover logical flaws in the implementations.

2.3. Dolev-Yao Attacker

In the Symbolic or Dolev-Yao model, messages are modeled symbolically as *terms* $\mathcal{T}$ using a term algebra. Let $\mathcal{A}$, $\mathcal{N}$, $\mathcal{K}$ and $\mathcal{V}$ be fixed countable sets of agents, nonces, keys and variables respectively and $\mathcal{B} = \mathcal{A} \cup \mathcal{N} \cup \mathcal{K}$ denotes the set of basic terms. Cryptographic primitives are treated as black-box functions. We distinguish between two types of functions: *constructors* and *destructors*. A constructor $f \in \mathcal{F}$, where $\mathcal{F}$ is a bounded fixed set, is used to build new terms such as encryption and appears in the terms. Formally a term $term \in \mathcal{T}$ can be defined with the following grammar:

$$term := m \mid f(term)$$

where $m \in \mathcal{B} \cup \mathcal{V}$ and $f \in \mathcal{F}$. Ground terms $\mathcal{T}_g$ are terms without variables. A destructor $g \in \mathcal{G}$ computes on terms, such as decryption and does not appear explicitly in terms. For instance, symmetric encryption and decryption are modeled using constructor $\texttt{enc} \in \mathcal{F}$ and destructor $\texttt{dec} \in \mathcal{G}$ with the equation:

$$\texttt{dec}(\texttt{enc}(m, k), k) = m$$

This means that, for a symbolic attacker, it is simply impossible to get back the plaintext m from ciphertext $\texttt{enc}(m, k)$ without the corresponding key k. In this paper, we use the shorthand $\{m\}_k$ to represent $\texttt{enc}(m, k)$.

A protocol is assumed to be executed in the presence of a symbolic attacker who has complete control over the network, that is, it can eavesdrop on communications, replay, block, and inject messages, but can only compute using these functions.

2.4. Needham-Schroeder-Lowe Protocol Example

We will use the Needham-Schroeder-Lowe protocol (NSL) [29] as an example to explain how PROINSPECTOR works. NSL is intended to provide mutual authentication of two agents, Alice (A) and Bob (B). We present a short version of the protocol as follows, where we assume Alice is a server that is willing to accept connections from any client whose identity is received in message 1 and agents are assumed to know each other's public keys:

$$\begin{aligned}
&(1)\ B \rightarrow A : B \\
&(2)\ A \rightarrow B : \{N_A, A\}pk_B \\
&(3)\ B \rightarrow A : \{N_A, N_B, B\}pk_A \\
&(4)\ A \rightarrow B : \{N_B\}pk_B
\end{aligned}$$

N_A and N_B are two fresh nonces generated by A with public key pk_A and B with public key pk_B respectively. $\{m\}_{pk_x}$ is the public key encryption of message m using public key pk_x. Intuitively, when Alice receives message 3, she knows Bob has tried to contact her and when Bob receives message 4, he knows that Alice accepts his connection. NSL fixes the flaws in Needham-Schroder protocol [33] and has been formally verified using symbolic provers. We use NSL to demonstrate how we can potentially find logic flaws that exist in implementations of the protocol.

3. Architecture Overview

A high-level overview of PROINSPECTOR is illustrated in Figure 1. It includes two main components: a custom conformance testing module and a symbolic protocol analysis tool such as ProVerif [13] or Tamarin [31]. The conformance testing module takes as inputs: the specification of a protocol under test and the implementation under test. The specification is a Mealy machine model $\mathcal{M}_{spec}$ (detailed in Section 4.1) that depicts the correct behavior model of the specification. We assume the specification is manually extracted, following a similar process of modelling a protocol in a symbolic prover such as Tamarin or ProVerif. It involves a meticulous examination of the protocol's RFC (Request for Comments) document to extract the control flow and message interactions defined within the protocol. The conformance testing will either output "OK", indicating all tested execution traces from the implementation are allowed by the specification; or counterexample traces that deviate from the specification.

In the latter case, these counterexample traces are given to an automated symbolic protocol analysis tool to check for security property violations. In this paper, we use ProVerif, but tools like Tamarin can be incorporated as well. The symbolic analysis tool typically takes as inputs, protocol specifications and encoding of the desired security property. In the case of ProVerif, the protocol specification is encoded using a process calculus and the properties we focused on are authentication and secrecy. In PROINSPECTOR, we additionally encode the deviant behavior as a separate process, so it can be analyzed together with the protocol specification for security property violations. The symbolic analysis tool will either return OK or a counterexample attack trace.

The main novelty of PROINSPECTOR lies in the conformance testing module; it incorporates a Dolev-Yao attacker model so that conformance testing can induce potentially dangerous behavior from the protocol implementation. To connect to symbolic analysis tools, the conformance testing module is also responsible for translating between messages in the form of symbolic terms that the symbolic analysis tools understand and the network packets that the implementation receives or sends. We will detail the conformance testing module in Section 4 and the components for identifying logic errors using a symbolic protocol verifier in Section 5.

4. Conformance Testing Under Dolev-Yao

In this section, we present our custom conformance testing module that takes into consideration, the Dolev-Yao attacker model. We also explain the two phases of generating testing cases that are needed both for incorporating the Dolev-Yao attacker model and for connecting concrete messages from the implementation to abstract messages used in specifications.

4.1. Protocol Specification as a Mealy Machine

Recall that PROINSPECTOR needs a manually extracted Mealy machine specification representing the behavior model of the protocol. We illustrate the process by constructing the Mealy machine representing Alice

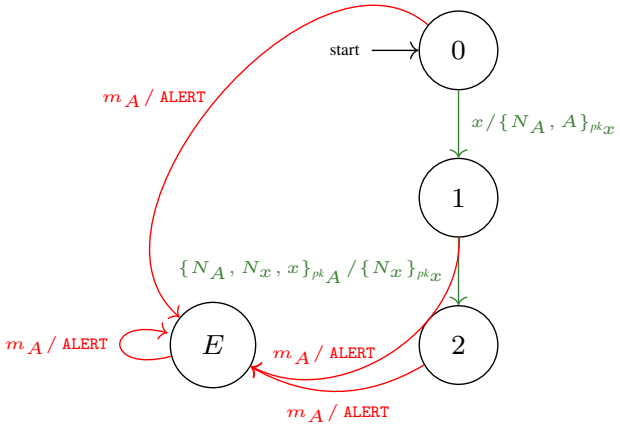


Figure 2: The reference Mealy Machine depicts Alice's behavior in NSL. The input of this mealy machine is $I_{NSL} = \{x, \{N_A, N_x, x\}_{pk_A}, m_A\}$ and the output of this mealy machine is $O_{NSL} = \{\{N_A, A\}_{pk_x}, \{N_x\}_{pk_x}, \texttt{ALERT}\}$. $x, N_x, pk_x \in \mathcal{V}$ are variables.

in the NSL protocol (Section 2.4). We interpret from the specification that Alice first receives a variable x representing any agent she is willing to communicate with and reply with her nonce N_A and identity A encrypted using the public key of the other party pk_X. Alice then expects to receive a message consisting of her nonce challenge N_A and the nonce challenge N_x and identity x of the other party all encrypted with Alice's public key. Alice can decrypt this message using her private key. Alice will check if the identity received is the same as the first message. Only if the identities match she will reply the nonce challenge N_x encrypted using the other party's public key. The control flow of the protocol described above is what we call a *happy* flow of the protocol where an honest agent connects to Alice without an active attacker blocking or modifying messages. This *happy* flow of the protocol is represented as green arrows in Figure 2.

Since we intend to analyze the protocol symbolically using a Dolev-Yao model, the Mealy machine is also augmented with transitions caused by an attacker's interactions with Alice. Thus, an abstract attacker message m_A is added to every state where we expect a correct implementation to output some kind of alert or error message (generalized as ALERT) and transition to an error state E. We call these transitions *unhappy* transitions, which are shown as red arrows in Figure 2. Having the *unhappy* transition on every state is key to modeling the capability of a Dolev-Yao attacker (Section 2.3), who has complete control over the network and can choose to block, inject, or modify messages at any point during the protocol execution. Note that this is a conservative representation of Alice's behavior, as not every attacker message should transition to an error state. For example, the attacker may choose to act as an honest agent in the protocol by sending valid and well-formed messages which a correct implementation would accept. We discuss

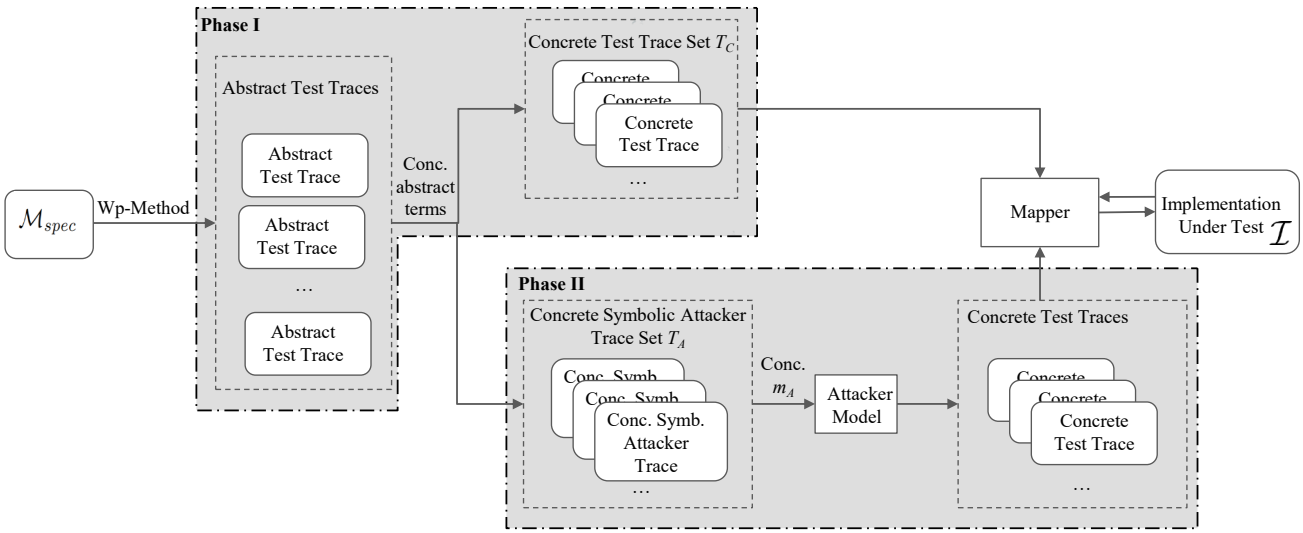


Figure 3: Conformance testing module of PROINSPECTOR.

how we handle this case in Section 4.4.

Figure 2 depicts one way to construct the reference Mealy machine. In this construction, any unexpected message causes the automaton to transition to an error state meaning that we only consider one run of the protocol. Another way to construct the reference Mealy machine, which we do not consider in this paper, would be to stay in the same state with m_A input and add transitions to loop back to the initial state such that the protocol can run multiple times back to back. This would increase the number of traces in the test suite and potentially detect bugs only manifest in multiple runs. Yet another way to construct the reference model is to include multiple sessions; whereas the reference model here only runs a single session (i.e., Alice can only communicate with one agent at a time). Introducing multiple session will increase the number of the states in the reference model, but this is unnecessary because symbolic protocol analysis tools such as ProVerif can handle infinite number of sessions. As we shall see in Section 6, our simple construction is enough to confirm the known vulnerabilities using our framework

An assumption we made about the IUTs is that the IUTs implement at least the *happy* flow of the reference Mealy machine. In other words, we are interested in finding logical errors in an implementation that can correctly communicate with an honest agent running the same protocol. While most protocols are more complex than our NSL example, the process of building the Mealy machine is identical where we start by examining the specification to identify the control flow, message formats, and interactions.

4.2. Conformance Testing Architecture

At a high level, conformance testing generates test cases from the specification. These test cases are then given to the implementation under test. We use Wp-Method, which is bounded by a parameter n, typically set to be the maximum number of states of the system under test. Figure 3 illustrates our novel conformance testing procedure, which involves two phases of test case generation. The first phase generates concrete test cases that only assess the normal (*happy*) traces of the protocol are also emitted by the IUT. The second phase uses our Dolev-Yao attacker model to concretize attacker messages left abstract (uninterpreted) during the first phase to generate a larger set of concrete test cases (detailed in Section 4.4).

Additionally, a mapper component translates the symbolic representation of the concrete test cases into the actual bitstring representation of network packages and vice versa. We defer the explanation of this component to Section 5, where we explain how to find logic errors using the symbolic analysis tool.

4.3. TG Phase I: Generating Concrete Test with Abstract Attacker Message

In this phase of the test case generation, inputs and outputs of our reference Mealy machine are abstract, meaning they contain uninstantiated variables. Abstraction is a well-studied technique in model-based testing aimed to reduce the size of inputs and outputs of the model [1]. Our abstraction is based on the term algebra used in the Dolev-Yao model which contains variables that can be instantiated into concrete terms. For example, if Bob (B) initiates a conversation with Alice when Alice's internal state is at state 0 (Alice receives B at state 0), the expected behavior of the specification (represented by the Mealy machine) says that Alice should send a fresh nonce N_A along with its identity A encrypted using Bob's public key pk_B ($\{N_A, A\}_{pk_B}$).

We use the Wp-Method on the Mealy machine $\mathcal{M}_S$ to generate a set of abstract test cases (denoted T_{abs}), as shown on the left of Figure 3. The set of abstract test cases for our NSL example is shown in Figure 4. We define the abstract test cases as follows:

Definition 4.1 (Abstract Test Trace (or Test Case) t_{abs})**.** Given a Mealy machine $\mathcal{M}_S = \langle Q, q_0, I, O, \delta, \lambda \rangle$ where the inputs and outputs are defined using the *Term algebra* introduced in Section 2.3 ($I, O \subseteq \mathcal{T}$), an abstract test case t_{abs} is a sequence of input-output pairs $(in_0, out_0)...(in_n, out_n) \in (I, O)^*$ s.t.

- t_{abs} is a trace (Definition 2.1) in $\mathcal{M}_{\mathcal{S}}$ ($t_{abs} \in \mathcal{M}_{\mathcal{S}}$) and
- if there is a variable $x \in \mathcal{V}$ in out_i, then there exists a previous input in_j in t_{abs} that contains the same variable x and $j \leq i$.

Example 4.1. The first column of Figure 4 lists the set of abstract test cases (traces).

We cannot execute directly an abstract test case on the IUT because it contains variables like x, a placeholder for the identity of other protocol participant, and attacker messages m_A. Thus, we need to concretize each abstract test case by substituting the variables with concrete values (e.g., B for x) and concrete attacker messages for m_A. This will result in one or more concrete test cases. Concretization is a two-step process which involves (1) concretize the variables and (2) concretize the attacker message m_A. Phase I here only considers (1) and returns two sets of traces T_C and T_A. T_C is the concrete trace set which we can already test on the implementation. T_A is what we call the *concrete symbolic attack trace set* which contains abstract attacker messages m_A. Here we use *symbolic* to indicate the attacker message is abstract.

Next we formally define concrete test traces and concrete symbolic attacker traces.

Definition 4.2 (Concrete Test Trace t_{conc} and Concrete Symbolic Attacker Trace t_a)**.** t is a concrete test trace if all inputs and outputs on the trace are ground terms, i.e., $\forall m \in t.in \cup t.out : m \in \mathcal{T}_g$. t is a concrete symbolic attacker trace if all inputs and outputs on the trace are (1) either ground terms or abstract attacker message m_A and (2) the trace contains at least one attacker message m_A. Formally defined as follows:

$$(\forall m \in t.in \cup t.out : m \in \mathcal{T}_g \vee m = m_A) \wedge$$
$$(\exists n : t[n].in = m_A).$$

Example 4.2. In the second column of Figure 4 concrete symbolic attacker traces are colored in red and concrete test traces are colored in green.

Concretization algorithm (no attack message) We describe the process of concretizing a single abstract test trace here. Let G be a finite set of ground terms and t_{abs} be an abstract test trace. For every variable in t_{abs}, replace it with a ground term $term \in G$ of the same type. Repeat the above process for all possible combinations of the variable substitution induced by G.

For example, given an abstract test trace $t_{abs} = (x, \{N_A, A\}_{pk_x}) \cdot (\{N_A, N_x, x\}_{pk_A}, \{N_x\}_{pk_x})$ and a set of terms $G = \{B, M, N_B, N_M, pk_B, pk_M\}$. This process will generate the following two traces:

$$(B, \{N_A, A\}_{pk_x}) \cdot (\{N_A, N_B, B\}_{pk_A}, \{N_B\}_{pk_B})$$
$$(M, \{N_A, A\}_{pk_M}) \cdot (\{N_A, N_M, M\}_{pk_A}, \{N_M\}_{pk_M})$$

Note there is an internal linking between agents and their nonces and keys visa the shared variable: $\{N_B, pk_B\} \mapsto B$ and $\{N_M, pk_M\} \mapsto M$.

We repeat this process for all traces in T_{abs} which generates the set of concrete symbolic attacker traces T_A and concrete test traces T_C. For our NSL example, T_C includes the traces colored green and T_A includes traces colored red in Figure 4. Essentially T_C includes test cases that only test the *happy* flow of the protocol. Because every test trace in T_C contains only ground terms, we can already test this set. If all the tests in T_C pass, i.e., $\forall t \in T_C : \mathcal{I}$ **passes** t, we are confident that the IUT at least implements the *happy* flow of the protocol. If this is not the case, we will manually check (and revise if necessary) whether our reference Mealy machine is a faithful representation of the specification or if the IUT is a custom protocol not suitable for being tested using this specification. One such example is when we want to test the 0-RTT mode of TLS (a faster mode to resume a previous connection), but certain implementations disable this mode to prevent replay attacks. In order to test T_A on the IUT, we need to further concretize the attacker message m_A which we explain next.

4.4. TG Phase II: Concretizing Attacker Messages under Dolev-Yao

In this phase, all occurrences of the abstract attacker message m_A are replaced with concrete messages the attacker knows. We first explain how we model attacker's knowledge before presenting the concretization algorithm.

Attacker's knowledge We define the predicate $\mathsf{knows}(K, m)$ to mean that the attacker can produce m after seeing all messages in the attacker's knowledge set K. We define selected rules below.

$$\frac{m \in K}{\mathsf{knows}(K, m)}\ \textsc{Init}$$

$$\frac{\forall i \in \{1, \ldots, n\}.\ \mathsf{knows}(K, m_i)}{\mathsf{knows}(K, f(m_1, \ldots, m_n))}\ \textsc{Comp}$$

$$\frac{\mathsf{knows}(K, (m_1, \ldots, m_n))}{\mathsf{knows}(K, m_i)}\ \textsc{Proj}$$

$$\frac{\mathsf{knows}(K, k) \qquad \mathsf{knows}(K, \{m\}_k)}{\mathsf{knows}(K, m)}\ \textsc{SDec}$$

The INIT rule states that an attacker can construct m if it is in its knowledge set K. Using the COMP rule, an attacker can construct messages using messages it already knows. Here f ranges over operations of the symbolic model. The next two rules show how an attacker can use the destructors for tuples (PROJ) and symmetric decryption (SDEC) respectively.

The attacker's knowledge set K is updated at the protocol runs. We can divide a concrete symbolic attacker trace t_a into a concrete test trace t_{conc} concatenated with another concrete symbolic attacker trace t'_a; specifically, $t_a = t_{conc} \cdot t'_a$ where $t_{conc} = t_a[0..n-1]$ and contains only ground terms; $t'_a = t_a[n..|t_a|]$ with $t_a[n].in = m_A$. Essentially, t_{conc} represents a snapshot of protocol execution in which the attacker simply observes messages exchanged and updates its knowledge. The attacker's current knowledge K_{curr} includes its initial knowledge K_I and all the messages appear in the network ($K_I \cup t_{conc}[0..n-1].in \cup t_{conc}[0..n-1].out$).

The attacker message for a Dolev-Yao attacker as represented by the predicate $\mathsf{knows}(K_{curr}, m)$ is unbounded, because the attacker can apply a constructor for infinite

Abstract Test Cases	Test Cases after First Phase of Concretization
$(x, \{N_A, A\}_{pk_x})$	$(B, \{N_A, A\}_{pk_B})$ $(M, \{N_A, A\}_{pk_M})$
$(x, \{N_A, A\}_{pk_x}) \cdot (\{N_A, N_x, x\}_{pk_A}, \{N_x\}_{pk_x})$	$(B, \{N_A, A\}_{pk_B}) \cdot (\{N_A, N_B, B\}_{pk_A}, \{N_B\}_{pk_B})$ $(M, \{N_A, A\}_{pk_M}) \cdot (\{N_A, N_M, M\}_{pk_A}, \{N_M\}_{pk_M})$
$(m_A, \texttt{ALERT})$	$(m_A, \texttt{ALERT})$
$(x, \{N_A, A\}_{pk_x}) \cdot (m_A, \texttt{ALERT})$	$(B, \{N_A, A\}_{pk_B}) \cdot (m_A, \texttt{ALERT})$ $(M, \{N_A, A\}_{pk_M}) \cdot (m_A, \texttt{ALERT})$
$(m_A, \texttt{ALERT}) \cdot (m_A, \texttt{ALERT})$	$(m_A, \texttt{ALERT}) \cdot (m_A, \texttt{ALERT})$
$(x, \{N_A, A\}_{pk_x}) \cdot (\{N_A, N_x, x\}_{pk_A}, \{N_x\}_{pk_x}) \cdot (m_A, \texttt{ALERT})$	$(B, \{N_A, A\}_{pk_B}) \cdot (\{N_A, N_B, B\}_{pk_A}, \{N_B\}_{pk_B}) \cdot (m_A, \texttt{ALERT})$ $(M, \{N_A, A\}_{pk_M}) \cdot (\{N_A, N_M, M\}_{pk_A}, \{N_M\}_{pk_M}) \cdot (m_A, \texttt{ALERT})$

Figure 4: Abstract test cases and test cases after first phase of concretization for NSL example mealy machine in Figure 2. We bound the Wp-Method with the depth of 4 and set the use the finite set $v = \{B, M, N_B, N_M, pk_B, pk_M\}$ for variables.

number of times. This leads to an infinite number of concrete test cases. To perform efficient testing, we limit the messages an attacker can send to a set called *allowed set* $S_{allowed}$. The allowed set is a user controlled parameter which lists the messages the attacker can send; any other message is discarded. This set is selected using heuristics, and the larger the size of the allowed set, the more complete the testing would be. In general, one would pick the allowed set to contain messages that respect the message format described in the protocol specification but with some message content mutated. For example, for the NSL protocol we define the allowed set as follows:

$$\begin{aligned} S_{allowed} = &\{B, M, \{n_1, n_2, p\}_{pk_x}\} \\ &\text{where } n_1 \leftarrow \{N_A, N_B, N_M\}, \\ &\quad n_2 \leftarrow \{N_A, N_B, N_M\}, \\ &\quad p \leftarrow \{A, B, M\}, \\ &\quad pk_x \leftarrow \{pk_A, pk_B, pk_M\} \end{aligned}$$

Concretization algorithm for attacker messages Next, we describe the algorithm for concretizing m_A.

Given a finite set of ground terms K_I representing attacker's initial knowledge, an allowed set $S_{allowed}$, and a finite set of a concrete symbolic attacker trace set T_A produced by Phase I Section 4.3. The set of concrete traces (initially empty) T_{conc} returned by Phase II is computed as follows:

Step 1: If T_A is not empty: pop a trace $t_a \in T_A$ and proceed to Step 2, otherwise, return T_{conc}.

Step 2: Scan the trace t_a starting from the first element until we find the first attacker message m_A at some index n, or we reach the end of the trace and no attacker message is found in the trace.

- If no attacker message is found, add t_a to T_{conc} and return to step 1.
- We find the attacker message at index n ($t_a[n].in = m_A$).
  We first need to update attacker's current knowledge set which is the union of attacker's initial knowledge and all the messages on the trace before the current attacker message. Furthermore, we add the concrete message an honest agent can send at this stage (termed $t_a[n].agent$). This essentially model the attacker's ability to block messages in the network. That is, the attacker blocks the message from honest agent and uses it to compute any other messages the attacker knows. $K_A = K_I \cup t_a[0..n-1].in \cup t_a[0..n-1].out \cup t_a[n].agent$.
  We can then compute the bounded attacker message set $M_A = \{m | \mathsf{knows}(K_A, m) \wedge m \in S_{allowed}\}$.
  Instantiate m_A with the set M_A and add all the traces back to T_A ($T_A = T_A \cup \{t_a[0..n-1] \cdot (m, \texttt{ALERT}) \cdot t_a[n+1..|t_a|] | m \in M_A\}$) and return to step 1.

For example, given a concrete symbolic attacker trace $t_a = (M, \{N_A, A\}_{pk_M}) \cdot (m_A, \texttt{ALERT})$, attacker initial knowledge $K_I = \{A, B, M, pk_A, pk_B, pk_M, sk_M, N_M\}$, and the allowed set $S_{allowed}$ above, t_a is transformed to the following concrete traces:

$$\begin{aligned} &(M, \{N_A, A\}_{pk_M}) \cdot (B, \texttt{ALERT}) \\ &(M, \{N_A, A\}_{pk_M}) \cdot (M, \texttt{ALERT}) \\ &(M, \{N_A, A\}_{pk_M}) \cdot (\{n_1, n_2, p\}_{pk_x}, \texttt{ALERT}) \\ &\qquad \text{where } n_1 \leftarrow \{N_A, N_M\}, \\ &\qquad\quad n_2 \leftarrow \{N_A, N_M\}, \\ &\qquad\quad p \leftarrow \{A, B, M\}, \\ &\qquad\quad pk_x \leftarrow \{pk_A, pk_B, pk_M\} \end{aligned}$$

$(M, \{N_A, A\}_{pk_M}) \cdot (\{N_A, N_M, M\}_{pk_A}, \texttt{ALERT})$ is one of the concrete trace generated after concretizing attacker messages. However, there is another concrete trace with the same inputs but different outputs generated earlier by Phase I: $(M, \{N_A, A\}_{pk_M}) \cdot (\{N_A, N_M, M\}_{pk_A}, \{N_M\}_{pk_M}) \in T_C$. As we mentioned, not all attacker messages should lead to an error (attacker could act as an honest agent in the protocol). Thus, we eliminate those traces generated by Phase II if there is already another trace in T_C with the same inputs sequence.

5. Leveraging Symbolic Analysis Tools

Once test cases are generated, PROINSPECTOR will test various protocol implementations with the help of the *Mapper* component to connect to a symbolic analysis tool. We first describe the Mapper component and then explain how to analyze failed traces automatically.

5.1. Mapper

The mapper is a component that sits in between the test cases generated and the IUT as shown in Figure 3. For the outward direction, the Mapper translates the concrete test cases written in term algebra into actual bitstrings and sends them to the IUT. For the inward direction, the Mapper parses and translates the responses/outputs from the IUT back into term algebra representation and compares the actual outputs with the expected outputs. If the outputs are different, the trace of the inputs with the actual outputs are stored for further analysis.

This component can leverage existing implementations of the protocol. The major implementation effort lies in crafting Dolev-Yao attacker messages that contain malformed or mutated content prescribed by our reference model together with the allowed attacker message set. This includes scenarios such as encrypting or signing with incorrect keys and mutating message fields to arbitrary content within attacker's knowledge including null or zero-padded values. Parsing the responses from a diverse set of implementations also requires significant implementation effort.

The mapper component used for our case study in Section 6 is implemented using `Scapy` [39], a Python-based library for encoding and decoding TLS packets. While the Mapper component needs to be implemented for different protocols (e.g., TLS [18, 38] vs. DTLS [30]), it can be reused for different implementations of the same protocol with minimal to no change required. We used the same Mapper for wolfSSL and OpenSSL where we first implemented the mapper that works with wolfSSL and then needed to slightly alter it because OpenSSL returns an error code on certain inputs before closing the connection, while wolfSSL silently close the connection.

5.2. Analyze Failed Traces

Recall that, IUT receives the input sequence of the test case $t.in$ and the Mapper checks if the output sequence from the IUTs is the same as in the test case $t.out$. If the outputs agree that means the execution passes the test, i.e., $\mathcal{I}$ **passes** t. Otherwise, a counterexample trace t_f is obtained, which behaves differently in the implementation. However, not all counterexamples lead to vulnerabilities. We could be overly conservative in our reference model, restricting honest actions that should be accepted. Furthermore, it is not obvious how to reconstruct an attack with a failed trace t_f, because the attack scenario might need multiple sessions of the protocol running concurrently, while the trace may record only one session of the protocol execution.

To check whether the erroneous transitions lead to attacks, we use automated tools such as ProVerif [13]. These automated symbolic protocol provers can automatically verify protocol or find attacks with unbounded sessions and an unbounded Dolev-Yao attacker. Next, we use our NSL example to demonstrate this process when we use ProVerif to perform the analysis.

One of the test cases we generated in Section 4.4 is $(M, \{N_A, A\}_{pk_M}) \cdot (\{N_A, N_M, B\}_{pk_A}$ `ALERT`$)$. Suppose instead of emitting `ALERT` message on the second input the actual output sequence observed on the IUT is

```
    event endBparam(pk(skB)).


let proctf(pkB:pkey, pkM:pkey, skA:skey) =
    in(c, x: host);
    if x = B || x = M then
    let pkX = if x = B then pkB else pkM in
        new Na: bitstring;
        event beginBparam(pkX);
        out(c, aenc((Na, A), pkX));
        in(c, m: bitstring);
```

Figure 5: ProVerif encoding of $Proc_{t_f}$

$t_f = (M, \{N_A, A\}_{pk_M}) \cdot (\{N_A, N_M, B\}_{pk_A}, \{N_M\}_{pk_M})$. According to our assumptions, the IUT implements all the *happy* flows in the reference Mealy machine but includes extra erroneous transitions where a malformed message should result in an `ALERT` message but doesn't. Thus, the first step is to model a correct Alice's process $Proc_A$ in the domain specific language (applied pi calculus [2]) of ProVerif. To analyze the protocol we also model an honest agent Bob $Proc_{Bob}$ running the protocol with Alice. In addition, we formulate the mutual authentication property $Prop_{auth}$ to be verified for the protocol. The modeling of Alice and Bob follows the standard process for using ProVerif to check if a specification of the protocol is secure under a Dolev-Yao attacker. For our example, we add another process $Proc_{t_f}$, corresponding to the failed trace t_f. $Proc_{t_f}$ is automatically obtained from t_f by mapping the message to Alice's process and modifying message checking such that the failed trace is accepted. Specifically, the process $Proc_{t_f}$ performs the following steps, where x, y and z are variables:

$$\begin{aligned} \textit{receives} &: x \\ \textit{sends} &: \{N_A, A\}_{pk_x} \\ \textit{receives} &: \{N_A, y, z\}_{pk_A} \\ \textit{sends} &: \{y\}_{pk_x} \end{aligned}$$

We use ProVerif to check the following:

$$!Proc_A || !Proc_B || !Proc_{t_f} \models Prop_{auth}$$

The above aims to check if the mutual authentication property $Prop_{auth}$ holds under an unbounded number of sessions of Alice, Bob and the failed trace processes running in parallel. We show the ProVerif encoding of $Proc_{t_f}$ in Figure 5 and the full modeling of the protocol can be found in Appendix A. ProVerif finds a violation of the property $Prop_{auth}$ with the following attack trace:

$$\begin{array}{llll} (1*) & M & \rightarrow & A & : M \\ (2*) & A & \rightarrow & M & : \{N_A, A\}_{pk_M} \\ (3) & B & \rightarrow & A & : B \text{ (blocks by } M) \\ (4) & M & \rightarrow & B & : \{N_A, A\}_{pk_B} \\ (5*) & B & \rightarrow & A & : \{N_A, N_B, B\}_{pk_A} \\ (6*) & A & \rightarrow & M & : \{N_B\}_{pk_M} \\ (7) & M & \rightarrow & B & : \{N_B\}_{pk_B} \end{array}$$

Messages marked with (*) point to the potentially erroneous transitions we found in the implementation. The built-in Dolev-Yao attacker in ProVerif is able to leverage

the trace from the implementation and finds the attack trace above. This attack trace shows a MITM attack where Bob thinks he is running the protocol with Alice but in reality he is running the protocol with Mallory acting as Alice. This attack trace mirrors the original flaw found in the Needham-Schroeder protocol [28, 29]. The NSL protocol fixes this flaw by adding the identity in $\{N_A, N_B, B\}_{pk_B}$ where Alice is supposed to check if the identity matches with previous messages. This revised version (NSL protocol) has been proven to be secure under Dolev-Yao [29]. However, developers might fail to implement the identity check (checking that the identity B received in message (5*) is the same as the identity received in message (1*)) correctly and reintroduce the bug in the implementations as shown in the example. This error cannot be found without the Dolev-Yao attacker in ProVerif, as the honest agents would not send the malformed message to trigger the erroneous transition.

6. Case Study

In this section, we demonstrate the effectiveness of PROINSPECTOR by using it to analyze popular TLS implementations including *wolfSSL* and *OpenSSL*. PROINSPECTOR rediscovered several known vulnerabilities.

6.1. Overview of TLS

The Internet standard TLS is a protocol to establish a secure channel between two agents, a client and a server, communicating over an untrusted network. TLS is most notably employed in the context of HTTPS, enabling secure interactions between web browsers and web servers. Specifically, the TLS protocol provides the following security assurances.

- *Authentication:* The server is always authenticated, while the client is optionally authenticated (requested by the server).
- *Secrecy:* Data transmitted over the secure channel, once established, is only visible to the endpoints.
- *Forward Secrecy:* Secrecy (above) holds even if long-term keys are leaked to the attacker after the session ends.

The TLS protocol comprises two fundamental subprotocols: firstly, the handshake protocol, which handles the negotiation of cipher suites, authenticates the parties, and establishes a shared session key. Subsequently, the record layer protocol utilizes the session key to establish the secure channel. We focus on the latest version TLS 1.3 [38], which enhances previous versions by removing legacy cryptography that was deemed insecure and by improving efficiency. A typical TLS 1.3 connection is shown in Figure 6 where both the client and the server are authenticated, and they do not resume a prior connection (no pre-shared key).

In the Handshake protocol, the client sends the `ClientHello` message which contains a random nonce and various negotiation parameters (supported version, cipher suites etc.) with a Diffie-Hellman key share in the `key_share` extension. The server processes the `ClientHello` message and responds with its own

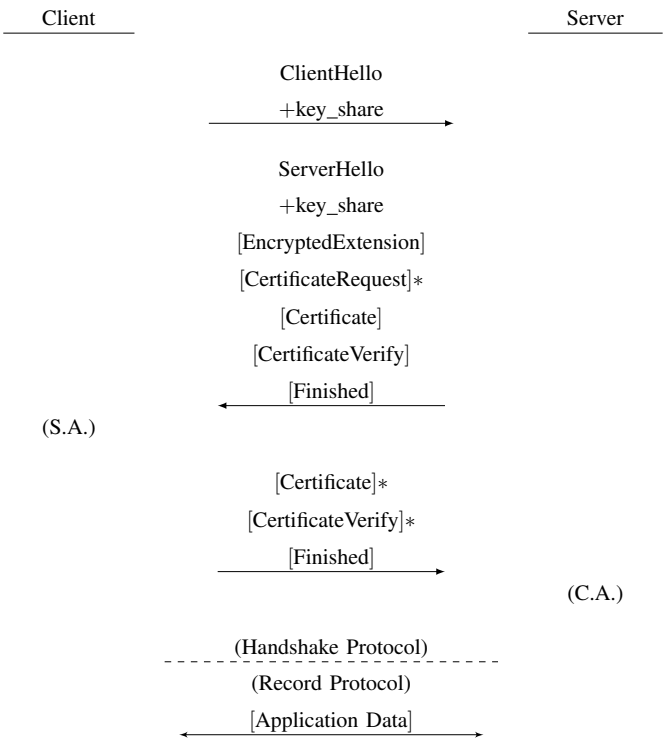


Figure 6: A typical TLS 1.3 connection where the Client is authenticated and there is no PSK. Messages enclosed with [ ] are encrypted. Messages marked with * are optional. S.A. and C.A. stands for Client and Server Authenticated, respectively.

`ServerHello`, which indicates the negotiated parameters, and a Diffie-Hellman key share in the `key_share` extension. At this point a `hand_shake_secret` is established and subsequent handshake messages are encrypted using this shared secret key. The server sends the encrypted hand shake messages `EncryptedExtension` (a list of extensions that can be protected), `Certificate` (contains server's public key), `CertificateVerify` (a signature over the handshake), `CertificateRequest` (request for client authentication) and `Finished` (MAC over the entire handshake). Upon checking the `Certificate` and `CertificateVerify` the client knows that the server is authenticated and the client sends its own `Certificate` and `CertificateVerify` for authentication together with the `Finished` message. Upon checking the client's `Certificate` and `CertificateVerify`, the server knows the client is authenticated. At this point the handshake protocol completes and the client and the server both establish a session key for the record layer protocol. In this TLS connection we explicitly require client authentication and disable session resumption.

6.2. IUTs and Reference State Machines

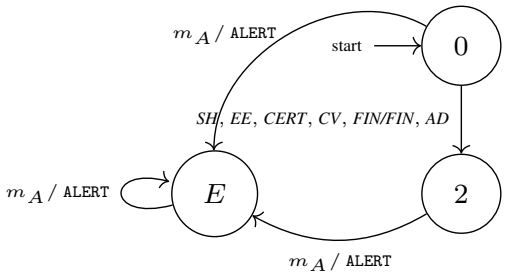
We chose *wolfSSL* and *OpenSSL* implementations for our case study; wolfSSL is an open-source implementation of SSL and TLS designed to be lightweight and cross-platform. However, early versions of wolfSSL were found to have multiple authentication bypass errors [36]. These authentication bypasses are a form of logic error

in which the implementation fails to perform the required authentication check, opening the door for impersonation and MITM (man-in-the-middle) attacks.

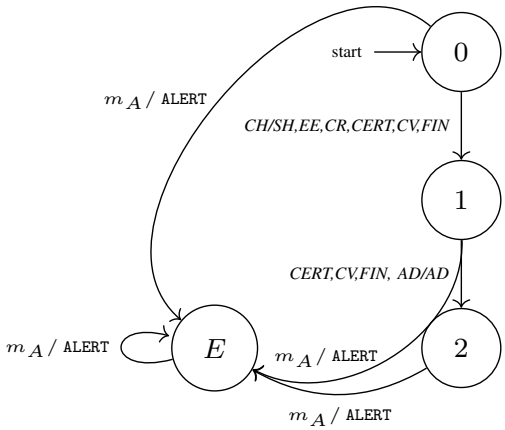
We used the flawed versions of wolfSSL that contains these authentication bypass errors to test if our framework can rediscover the errors and evaluate the effectiveness of our approach. To demonstrate the reusability of the mapper component, we further test an *OpenSSL* implementation using the same test cases and mapper program. The versions we used in our case study are: wolfSSL 4.4.0, 4.6.0, 5.1.0 and OpenSSL 3.0.2. We tested both client-side and server-side code.

- **Client-side**: In this scenario the IUT is a TLS 1.3 client and initiates a connection with the server. The client IUTs are configured with trusted certification authority to check for server authentication. The client IUTs will send application data only if the handshake protocol finishes correctly and the server is authenticated. Figure 7a shows the reference client Mealy machine for our conformance testing. For simplicity, we did not test client authentication and leave `CertificateRequest` out of inputs.
- **Server-side**: In this scenario the IUT is a TLS 1.3 server. The server is configured to require client authentication with a given certificate authority. This is also reflected in our reference Mealy machine where only paths with valid `Certificate` and `CertificateVerify` are accepted. Figure 7b shows our reference Mealy machine for the TLS 1.3 server.

The inputs and outputs shown in the reference Mealy machines in Figure 7 are simplified for illustration purpose. In particular, we do not show the variables in the inputs and outputs. These variables are needed so that the attacker can generate attacker messages that vary in the variable fields. For example, the attacker can replace the public key in the `Certificate` message with its own untrusted key or include an arbitrary signature scheme algorithm in `CertificateVerify`. The actual reference Mealy machines we used in the case study have the same states and transitions but the messages are more detailed (similar to the one shown in Figure 2).



(a) Reference Mealy Machine for TLS 1.3 client. This reference does not consider session resumption and client authentication



(b) Reference Mealy Machine for TLS 1.3 server.

Figure 7: Reference Mealy Machines for TLS 1.3 client and server. Shorthands used on the inputs and outputs are *CH:* `ClientHello` *SH*: `ServerHello`, *EE*: `EncryptionExtension`, *CERT*:`Certificate`, *CV*: `CertificateVerify`, *CR*: `CertificateRequest`, *FIN*: `Finished` and *AD*: `ApplicationData`.

6.3. Experimental Setup

As discussed in Section 4.4, our framework includes a parameterized attacker that has an *initial knowledge* and can send any message in the *allowed set* if the attacker possesses all the elements in that message. We configure attacker's initial knowledge such that it includes a random number/nonce, attacker's own Diffie-Hellman key to initiate a session with the client or server, empty and untrusted certificates, and an uncertified/bad signing key. Notably, the attacker does not know the signing key of a valid certificate or the Diffie-Hellman key of an honest client/server initially. However, the attacker could learn these values acting as a Dolev-Yao attacker, for example, if an honest agent sends its private signing key to the network in plaintext then the attacker can use this signing key subsequently.

The allowed set includes all the messages client/server sends over the network s.t. the attacker can replay these messages. The set also includes the messages the attacker can send impersonating an honest client/server to initiate a session. Moreover, we allow any combination of subcomponent of a message to be sent. For example, *SH+EE+CERT+CV+FIN* is a message that has 5 components, any combination of these components is allowed (e.g., *SH+EE* or *FIN*). This effectively allows the attacker to skip protocol messages of their choice.

We set the bound of Wp-method to be the number of states in the reference Mealy machines (3 and 4) and generate test cases for the client-side and server-side IUTs respectively.

Our ProVerif specification of the protocol is based on RefTLS [10]. RefTLS is a verified implementation of TLS protocol. It was developed during the standardization of TLS 1.3 to check its security guarantees and contains an extracted symbolic model of the protocol in ProVerif. We simplified this model for our specific needs. Our ProVerif properties checks server authentication in client-side IUTs

and mutual authentication in server-side IUTs. In addition, we check the secrecy of application data in both scenarios.

6.4. Authentication Bypasses in wolfSSL

The first authentication bypass vulnerabilities on TLS implementations were found by two academic research works [8, 17] around 2015. The vulnerabilities were affecting early versions of OpenSSL and JSSE TLS 1.2 implementations. More authentication bypass vulnerabilities were found affecting wolfSSL TLS 1.3 implementations [36]. The four known authentication bypasses on wolfSSL are listed in the first column of Table 1 with their CVE numbers. Our framework is able to rediscover all four vulnerabilities. The second column of Table 1 shows the versions of implementations in our case study that exhibit the vulnerability in the first column. We confirmed with CVE reports that our results are accurate, that is we are able to find all affected versions in our test suit for the CVEs in Table 1.

TABLE 1: Vulnerabilities in wolfSSL.

CVE #	Version(s) with Vuln. Confirmed
CVE-2020-24613	wolfSSL 4.4.0
CVE-2021-3336	wolfSSL 4.4.0, 4.6.0
CVE-2022-25638	wolfSSL 4.4.0, 4.6.0, 5.1.0
CVE-2022-25640	wolfSSL 4.4.0, 4.6.0, 5.1.0

Next we explain each CVE re-discovered.

CVE-2020-24613: This vulnerability affected wolfSSL TLS 1.3 client-side code up to version 4.4.0. It allows a malicious server to skip the `CertificateVerify` message and thus impersonate any server to a vulnerable client. PROINSPECTOR is able to generate the following test cases that rediscover this vulnerability. The first one skips the server certificate message. The client is supposed to output `ALERT`, indicating an error; instead, it completes the handshake and starts data transmission. The first input/output pair is the expected behavior. The second input/output pair is the actual trace emitted by the client implementation.

Expected:
$(\texttt{ServerHello} + \texttt{EncryptionExtension} + \texttt{Finished}, \texttt{Alert})$
Actual:
$(\texttt{ServerHello} + \texttt{EncryptionExtension} + \texttt{Finished}, \texttt{Finished} + \texttt{ApplicationData})$

The second test case identifies the same bug. The difference compared to the test case above is that the server sends an empty or a valid certificate. Here, we write $\texttt{Certificate}^1$ to denote an empty certificate and $\texttt{Certificate}^2$ to denote a valid certificate.

Expected:
$(\texttt{ServerHello} + \texttt{EncryptionExtension} + \texttt{Certificate}^{1,2} + \texttt{Finished}, \texttt{Alert})$
Actual:
$(\texttt{ServerHello} + \texttt{EncryptionExtension} + \texttt{Certificate}^{1,2} + \texttt{Finished}, \texttt{Finished} + \texttt{ApplicationData})$

In both cases, the client did not check the `CertificateVerify` message, violating the authentication property. PROINSPECTOR's conformance testing module is able to generate these attacker input sequences by consulting the Dolev-Yao attacker model. The original Wp-method would not have been able to generate these test cases. CVE-2020-24613 was patched, such that neither `Certificate` nor `CertificateVerify` messages cannot be skipped.

CVE-2021-3336: After patching the above CVE, the client code still accepts an empty certificate $\texttt{Certificate}^1$ followed by a $\texttt{CertificateVerify}^1$ message signed by arbitrary RSA keys. The test cases (multiple test cases map to this vulnerability as the signing key for the signature is arbitrary) that rediscovers the vulnerabilities in wolfSSL 4.6.0 is as follows:

Expected:
$(\texttt{ServerHello} + \texttt{EncryptionExtension} + \texttt{Certificate}^1 + \texttt{CertificateVerify}^1 + \texttt{Finished}, \texttt{Alert})$
Actual:
$(\texttt{ServerHello} + \texttt{EncryptionExtension} + \texttt{Certificate}^1 + \texttt{CertificateVerify}^1 + \texttt{Finished}, \texttt{Finished} + \texttt{ApplicationData})$

CVE-2022-25638: Later, a vulnerability similar to CVE-2021-3336 was discovered on the client code, where the client accepts an empty certificate message $\texttt{Certificate}^1$ followed by a $\texttt{CertificateVerify}^2$ message with an unknown signature algorithm. This again enables server impersonation attack. The following test case confirmed this vulnerability in both wolfSSL 4.6.0 and 5.1.0:

Expected:
$(\texttt{ServerHello} + \texttt{EncryptionExtension} + \texttt{Certificate}^1 + \texttt{CertificateVerify}^2 + \texttt{Finished}, \texttt{Alert})$

Actual:

(ServerHello + EncryptionExtension
+ Certificate^1 + CertificateVerify^2
+ Finished,
Finished + ApplicationData)

CVE-2022-25640: Authentication bypasses also exist on server side where a buggy server requests for client authentication but is happy to establish connection even if the client skips the `CertificateVerify` message and optionally skips the `Certificate` message. Without the authentication check, any user could pretend to be an authenticated client and establish sessions with the flawed server. This vulnerability exists in all versions in our test suite. Similar to the first case, PROINSPECTOR generated the following test cases to trigger the same vulnerability. Both test cases consist of two input/output pairs, delimitated by •.

Expected:

(ClientHello,
ServerHello + EncryptionExtension
+ CertificateRequest + Certificate
+ CertificateVerify + Finished)
•(Finished + ApplicationData, Alert)

Actual:

(ClientHello,
ServerHello + EncryptionExtension
+ CertificateRequest + Certificate
+ CertificateVerify + Finished)
•(Finished + ApplicationData,
ApplicationData)

Expected:

(ClientHello,
ServerHello + EncryptionExtension
+ CertificateRequest + Certificate
+ CertificateVerify + Finished)
•(Certificate^{1,2} + Finished + ApplicationData,
Alert)

Actual:

(ClientHello,
ServerHello + EncryptionExtension
+ CertificateRequest + Certificate
+ CertificateVerifyFinished)
•(Certificate^{1,2} + Finished + ApplicationData,
ApplicationData)

These test cases are similar to the ones we have seen before, except that these test the server and previous ones test the client. As a result, the input starts with `ClientHello`, instead of `ServerHello` and that we need two roundtrip communications with the server instead of one with the client. It should be noted that all versions of wolfSSL we tested are subject to this vulnerability. However, wolfSSL 4.6.0 and 5.1.0 reject the empty certificate Certificate^1, while wolfSSL 4.4.0 accepts the empty certificate and, optionally, a CertificateVerify^1 message signed by arbitrary keys.

6.5. Evaluation results

Failed traces and vulnerability confirmation: We summarize our evaluation results in Table 2. For all the failed traces, our ProVerif models were able to demonstrate a violation of the authentication properties and secrecy property. The attacks show that an untrusted server or client is able to impersonate a trusted entity and obtain confidential data. Apart from the test traces that exhibit these vulnerabilities, all other traces behave as expected, and we didn't observe any false positives.

Performance: The experiment was conducted on a desktop equipped with 32GB of RAM and an Intel i7-12700 processor. Generating the test cases for conformance testing required less than 1 second. The execution time of test cases on the IUTs largely depends on the size of the test cases and the timeout value we set when the IUT is not responding to the input. We observed that the responses from the IUTs were all within 1 second, so we set the timeout value to be 2 seconds. We also tested with a larger timeout value (1 minute), and the results did not change. Running test cases on an IUT took less than 10 minutes for all the IUTs we tested. Using ProVerif to check non-conformance traces took less than 1 second in all cases. Furthermore, the experiment is highly parallelizable since all the generated test cases are independent

6.6. Discussion

PROINSPECTOR's ability to detect vulnerabilities relies on our reference model and some framework configurations. In this section we discuss our design decisions and limitations of our framework.

Reference model We model the protocol using a Mealy machine, which is natural for security protocols and also is compatible with the Wp-method. Our reference Mealy machine not only has to model the correct protocol logic and the message sequences but also has to have a meaningful abstraction level. Fine-grained messages reveal more logic states of the implementations which might be abstracted away in a coarser-grained model. We believe existing verified protocol models used in symbolic provers offer a good guideline. However, this introduces repeated work in the framework and possibly internal inconsistencies between the reference Mealy machine model and the model used in the symbolic prover. Automatically extracting the reference model from the verified symbolic model (e.g., of ProVerif or Tamarin) or performing conformance testing directly on the verified symbolic model would be interesting future work. Tamarin's model is a good candidate as its multiset rewriting representation of protocols closely aligns with our Mealy machine representation.

Wp-method bound The parameter 'n' of the Wp-method represents the number of state-machine states; the test cases generated are guaranteed to cover all paths/transitions up to a bound which we set to n. Setting up the bound to be the number of states in the reference Mealy machine allows us to cover all the states in the reference

TABLE 2: Experimental results. The total number of test cases we generate for client IUTs and Server IUTs are **318** and **184** respectively. The first column lists the version of the IUT; the second column shows the number non-conforming traces; the fourth column shows the number of non-conforming traces ProVerif is able to find an attack; the last column corresponds to the CVE number of the found attack.

Version#	*# of Non-conforming Attack Traces*	*# of ProVerif's Traces*	*Comfirmed CVE#*
4.4.0 Client	7	7	2020-24613,2021-3336,2022-25638
4.6.0 Client	4	4	2021-3336,2022-25638
5.1.0 Client	1	1	2022-25638
OpenSSL Client	0	-	-
4.4.0 Server	7	7	2022-25640
4.6.0 Server	2	2	2022-25640
5.1.0 Server	2	2	2022-25640
OpenSSL Server	0	-	-

Mealy machine which is also employed in other works that utilize conformance testing [17]. Moreover, our framework, can be easily adapted to support other conformance testing algorithms such as Random Walks [27]. We plan to add support for the selection of conformance testing algorithms in the future.

Attacker bound We also need to configure our attacker model, which involves defining the allowed set of attacker messages. We have found it helpful to include invalid values, such as untrusted keys, unsupported encryption algorithms and empty or zero-padded message fields. The ability to arbitrarily drop a compound message that contains multiple subfields enables us to rediscover authentication bypasses in wolfSSL. Developing a principled approach to automatically generate attacker's configurations and enhance coverage would be interesting future work.

TLS protocol features and properties analyzed Our case study only considers a simple TLS connection which does not includes session resumption or early data transmission. Additionally, we only consider basic secrecy and authentication properties overlooking advance security features such as forward secrecy. Testing the full TLS protocol and forward secrecy necessitate a different and more complex modelling scheme.

Scalability and false positives Potential true/false positives and scalability issues may arise as the complexity of our reference model increases. To alleviate scalability issues, we must employ a principled approach to generating attacker's configurations as mentioned earlier.

We only test a single session of TLS protocol without pre-shared key or early data. As we expand our testing to include more TLS modes and refine our symbolic message representation, both the number of true positives and false positives may increase. False positives may manifest in two areas within our framework: firstly, in non-conforming traces between the IUT and the reference model. We utilize ProVerif to verify if these non-conforming traces indeed result in security vulnerabilities; and therefore, secondly, false positives may occur in attack traces returned by ProVerif. In our case study, these attack traces match the traces executed on the IUTs. However, it is possible that attack trace differs from the trace executed on the IUTs (as shown in Section 5.2). In such cases, we need to verify if the attack trace also exists in the IUT. This check can be performed using the mapper component, which translates textual representations of protocol messages into the actual bitstrings executed on the IUT.

7. Related Work

Much work has been done on the verification [6, 12, 10, 9, 24, 26] and testing of protocol implementations [22, 32, 8, 36, 4, 35, 20, 37, 40, 34]. Our work complements recent efforts on verified TLS implementations [10, 12, 9]. Verification efforts focus on developing new verified protocol implementations which are yet to be used in practice [6, 12, 10, 9, 24, 26]. In contrast, PROINSPECTOR employs light-weight testing to identify vulnerabilities in already deployed implementations.

While our framework bears similarities to property-based testing methods such as QuickCheck [15, 25], one important distinction is the use of a bounded Dolev-Yao attacker model to explore protocol states. Our bounded Dolev-Yao attacker is similar to that used in Taglierino [43, 42], which performs bounded model checking of the protocol specifications.

For the rest of this section, we discuss closely related work on testing protocol implementations.

Protocol state machine bugs Two seminal works [8, 17] discovered a form of logic errors in TLS 1.2 implementations which are termed as *protocol state machine bugs*. De Ruiter et al. adopted a model learning-based approach [17], which we discuss in the next paragraph. Similar to our method, Beurdouche et al.'s approach involved using a reference protocol state machine to generate deviant traces for testing implementations [8]. The deviant traces in the implementations were manually inspected for potential exploits. Three heuristics were proposed to generate those deviant traces: *Skip*, *Hop* and *Repeat*. Our method offers a more principled way to generate the deviant traces with less manual effort. Furthermore, the abstraction in Beurdouche et al.'s approach is at the level of protocol states, which allows testing with different message sequences [8]. We offer more fine-grained testing where we not only test for different message sequences but also allow mutations in the messages themselves. Further, we also use automated provers to find exploits.

Model-learning-based approaches Model learning can be used to automatically extract the (potentially buggy) protocol state machine from implementations. For instance, De Ruiter et al. apply model learning techniques to analyze the TLS 1.2 protocol [17]. The same methodology was later extended to the DTLS protocol [20] and QUIC [37]. However, the resulting state machines still require manual inspection to identify potential bugs. Fiterau-Brostean et al. offer more automation by model checking the learned models of SSH implementations [22]. However, these learned models all focus on protocol states [17, 20, 37, 22], limiting their scope to protocol state machine bugs. Therefore, the method could fail on our case study where we have found a bug triggered by a message signed with a bad key.

Following works [36, 4] also leverage model learning techniques but introduce mutation to protocol messages. Aichernig et al. [4] proposes learning-based fuzzing of IoT message brokers. Their approach involves first learning the state machine of the IoT message broker followed by fuzzing to mutate protocol messages. Different from our approach, they do not target logic errors and instead look for unexpected behavior caused by accepting illegal characters in the mutated messages. Rasoamanana et al. [36] learn a protocol state machine with more fine-grained messages. However, they also manually inspect the learned models for potential attacks.

Our method is similar to the above works in that we also analyze the implementation based on a state-machine model. However, instead of doing multiple rounds of learning (which is expensive in practice) to learn a (potentially buggy) model and compare it with a reference specification or across different learned models, we directly use conformance testing between the specification and implementation and do not assess differences across different implementations. In addition, we provide a principled method for mutating messages by leveraging the Dolev-Yao attacker, enabling detailed analysis not only of protocol states but also of protocol messages. We also use automated symbolic protocol analysis tools to increase automation.

Fuzzing-based approaches There are many fuzzing techniques for protocols implementations [40, 35, 21, 34]. However, these existing techniques do not target logic errors but rather focus on memory safety errors instead. LTL-Fuzzer [32] uses linear temporal logic to guide grey-box fuzzing of protocol implementations. However, it narrowly focuses on finding violations of basic correspondence of messages and does not include a Dolev-Yao attacker. Most recently, Ammann et al. introduced DY Fuzzing [5], which mutates fuzzing inputs based on Dolev-Yao attacker. This is very similar to our methodology, except that our attacker model can automatically update its knowledge (e.g. the available en/decryption keys) based on the messages it has seen. In addition, our connection to the reference symbolic model enables us to detect security properties violations using the symbolic prover.

8. Conclusion and Future Work

We presented an automated and systematic framework to uncover logic errors in protocol implementations. Central to our approach is a tailored conformance testing algorithm that can generate inputs based on a bounded Dolev-Yao attacker. The bounded attacker model serves as a valuable bridge between symbolic protocol verification and protocol implementation testing. Our approach enables us to generate test cases that contain fine-grained, structured term-level mutations at the protocol message level, such as using different encryption keys or swapping nonces, rather than unstructured random bit-flipping mutations or coarse-grained reordered message sequences. We used a generic symbolic prover to check if inconsistencies between the specification and implementation lead to exploits. We implemented the proposed methodology in the PROINSPECTOR tool and applied it on popular TLS implementations leading to the rediscovery of several logical bugs.

Future work includes making our tool more robust s.t. it can generate attacker configurations automatically. We also want to investigate possible avenues to either extract a reference Mealy machine model from a verified symbolic model or perform conformance testing on the verified symbolic model directly. Finally, we plan to apply PROINSPECTOR to additional protocols such as MQTT.

Data Availability

ProInspector and the experiment data are open-sourced at https://github.com/proj-proinspector/proinspector

Acknowledgements

We thank the anonymous reviewers for their feedback. This work was partially funded by ONR award no. N000141812618 and Carnegie Mellon CyLab.

References

[1] Fides Aarts, Bengt Jonsson, Johan Uijen, and Frits W. Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods Syst. Des.*, 46(1):1–41, 2015.

[2] Martín Abadi, Bruno Blanchet, and Cédric Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *Journal of the ACM (JACM)*, 65(1):1–41, 2018.

[3] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, et al. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[4] Bernhard K Aichernig, Edi Muškardin, and Andrea Pferscher. Learning-based fuzzing of IoT message brokers. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2021.

[5] Max Ammann, Lucca Hirschi, and Steve Kremer. DY fuzzing: formal Dolev-Yao models meet cryptographic protocol fuzz testing. *Cryptology ePrint Archive*, 2023.

[6] Matteo Avalle, Alfredo Pironti, and Riccardo Sisto. Formal verification of security protocol implementations: a survey. *Formal Aspects Comput.*, 26(1):99–123, 2014.

[7] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. In *42nd IEEE Symposium on Security and Privacy (SP)*, 2021.

[8] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *2015 IEEE Symposium on Security and Privacy (SP)*, 2015.

[9] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. DY*: A modular symbolic verification framework for executable cryptographic protocol code. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021.

[10] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[11] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *2014 IEEE Symposium on Security and Privacy (SP)*, 2014.

[12] Karthikeyan Bhargavan, Cédric Fournet, and Markulf Kohlweiss. miTLS: Verifying protocol implementations against real-world attacks. *IEEE Secur. Priv.*, 14(6):18–25, 2016.

[13] Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations (CSFW)*, 2001.

[14] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng.*, 4(3):178–187, 1978.

[15] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000.

[16] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[17] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security)*, 2015.

[18] Tim Dierks and Eric Rescorla. RFC 5246: The transport layer security (TLS) protocol version 1.2, 2008.

[19] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29(2):198–207, 2006.

[20] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS implementations using protocol state fuzzing. In *29th USENIX Security Symposium (USENIX Security)*, 2020.

[21] Paul Fiterău-Broştean, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. DTLS-fuzzer: A DTLS protocol state fuzzer. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2022.

[22] Paul Fiterau-Brostean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits W. Vaandrager, and Patrick Verleg. Model learning and model checking of SSH implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, 2017.

[23] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Trans. Software Eng.*, 17(6):591–603, 1991.

[24] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. Noise*: A library of verified high-performance secure channel protocol implementations. In *43rd IEEE Symposium on Security and Privacy (SP)*, 2022.

[25] John Hughes. Quickcheck testing for fun and profit. In *9th International Symposium of Practical Aspects of Declarative Languages (PADL)*, 2007.

[26] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[27] László Lovász. Random walks on graphs. *Combinatorics, Paul erdos is eighty*, 2(1-46):4, 1993.

[28] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56(3):131–133, 1995.

[29] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop (TACAS)*, 1996.

[30] David McGrew and Eric Rescorla. RFC 5764: Datagram transport layer security (DTLS) extension to establish keys for the secure real-time transport protocol (SRTP), 2010.

[31] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification - 25th International Conference (CAV)*, 2013.

[32] Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. Linear-time temporal logic guided greybox fuzzing. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022.

[33] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[34] Yan Pan, Wei Lin, Liang Jiao, Yuefei Zhu, et al. Model-based grey-box fuzzing of network protocols. *Security and Communication Networks*, 2022.

[35] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNET: A greybox fuzzer for network protocols. In *13th IEEE International Conference on Software Testing, Validation and Verification (ICST)*, 2020.

[36] Aina Toky Rasoamanana, Olivier Levillain, and Hervé Debar. Towards a systematic and automatic use of state machine inference to uncover security flaws and fingerprint TLS stacks. In *27th European Symposium on Research in Computer Security (ESORICS)*, 2022.

[37] Abdullah Rasool, Greg Alpár, and Joeri de Ruiter. State machine inference of QUIC. *arXiv preprint arXiv:1903.04384*, 2019.

[38] Eric Rescorla. RFC 8446: The transport layer security (TLS) protocol version 1.3, 2018.

[39] R Rohith, Minal Moharir, G Shobha, et al. SCAPY-A powerful interactive packet manipulation program. In *2018 international conference on networking, embedded and wireless systems (ICNEWS)*, 2018.

[40] Juraj Somorovsky. Systematic fuzzing and testing of TLS libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[41] Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*, pages 1–38. 2008.

[42] Zichao Zhang, Arthur Azevedo de Amorim, Limin Jia, and Corina Păsăreanu. Learning assumptions for verifying cryptographic protocols compositionally. In *Formal Aspects of Component Software: 17th International Conference (FACS)*, 2021.

[43] Zichao Zhang, Arthur Azevedo de Amorim, Limin Jia, and Corina S. Păsăreanu. Automating compositional analysis of authentication protocols. In *2020 Formal Methods in Computer Aided Design (FMCAD)*, 2020.

A. ProVerif Encoding of the Example from Section 5

```
free c: channel.

(* Public key encryption *)
type pkey.
type skey.

fun pk(skey): pkey.
fun aenc(bitstring, pkey): bitstring.
reduc forall x: bitstring, y: skey;
  adec(aenc(x, pk(y)),y) = x.

type host.
free A, B, M: host.

(* Authentication queries *)
event beginBparam(pkey).
event endBparam(pkey).
event beginAparam(pkey).
event endAparam(pkey).

query x: pkey; inj-event(endBparam(x)) ==>
    inj-event(beginBparam(x)).
query x: pkey; inj-event(endAparam(x)) ==>
    inj-event(beginAparam(x)).

let procA(pkB:pkey, pkM:pkey, skA:skey) =
    in(c, x: host);
    if x = B || x = M then
    let pkX = if x = B then pkB else pkM in
        new Na: bitstring;
        event beginBparam(pkX);
        out(c, aenc((Na, A), pkX));
        in(c, m: bitstring);
        let (=Na, NX: bitstring, =x) =
            adec(m, skA) in
        out(c, aenc(NX, pkX));
        if pkX = pkB  then
        event endAparam(pk(skA)).

let procB(pkA: pkey, skB: skey) =
        in(c, m: bitstring);
        let (NY: bitstring, y: host) = adec
            (m, skB) in
    if y = A then
        event beginAparam(pkA);
        new Nb: bitstring;
        out(c, aenc((NY, Nb, B), pkA));
        in(c, m3: bitstring);
        if Nb = adec(m3, skB) then
        event endBparam(pk(skB)).

let proctf(pkB:pkey, pkM:pkey, skA:skey) =
    in(c, x: host);
    if x = B || x = M then
    let pkX = if x = B then pkB else pkM in
        new Na: bitstring;
        event beginBparam(pkX);
        out(c, aenc((Na, A), pkX));
        in(c, m: bitstring);
        let (=Na, y: bitstring, z:host) =
            adec(m, skA) in
        out(c, aenc(y, pkX));
        if pkX = pkB  then
        event endAparam(pk(skA)).

(* Main *)
process
        new skA: skey; let pkA = pk(skA) in
             out(c, pkA);
        new skB: skey; let pkB = pk(skB) in
             out(c, pkB);
        new skM: skey; let pkM = pk(skM) in
             out(c, pkM); out(c,skM);
        ( (!procA(pkB, pkM, skA)) | (!procB
            (pkA, skB)) | (!proctf(pkB, pkM
            , skA)) )
```