Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH

Open Access

An enhanced state-aware model learning approach for security analysis in lightweight protocol implementations

Jiaxing Guo[1], Dongliang Zhao[1], Chunxiang Gu[1*], Xi Chen[1], Xieli Zhang[1] and Mengcheng Ju[1]

Abstract

Owing to the emergence and rapid advances of new-generation information and digitalization technologies, the concept of model-driven digital twin has received widespread attentions and is developing vigorously. Driven by data and simulators, the digital twin can create the virtual twins of physical objects to perform monitoring, simulation, prediction, optimization, and so on. Hence, the application of digital twin can increase efficiency and security of systems by providing reliable model and decision supports. In this paper, we propose a state-aware model learning method to simulate and analyze the lightweight protocol implementations in edge/cloud environments. We introduce the data flow of program execution and network interaction inputs/outputs (I/O) into the extended finite state machine (EFSM) to expand the modeling scope and insight. We aim to calibrate the states and construct an accurate state-machine model using a digital twin based layered approach to reasonably reflect the correlation of a device's external behavior and internal data. This, in turn, improves our ability to verify the logic and evaluate the security for protocol implementations. This method firstly involves instrumenting the target device to monitor variable activity during its execution. We then employ learning algorithms to produce multiple rounds of message queries. Both the I/O data corresponding to these query sequences and the state calibration information derived from filtered memory variables are obtained through the mapper and execution monitor, respectively. These two aspects of information are combined to dynamically and incrementally construct the protocol's state machine. We apply this method to develop SALearn and evaluate the effectiveness of SALearn on two lightweight protocol implementations. Our experimental results indicate that SALearn outperforms existing protocol model learning tools, achieving higher learning efficiency and uncovering more interesting states and security issues. In total, we identified two violation scenarios of rekey logic. These situations also reflect the differences in details between different implementations.

Keywords Model learning, State aware, Digital twin, Lightweight protocol implementations

Introduction

With the flourishing development of smart and digital technologies including Internet of Things (IoT), the fifth-generation cellular network (5G), cloud computing, and big data, various kinds of intelligent and digitalized products are widely revolutionizing today's society [1–3]. To provide efficient monitoring, modeling, analysis, and optimization from an overall perspective for these smart systems and devices, digital twin, an emerging technology based on model-driven to achieve physical-virtual convergence, is proposed [4, 5]. Digital twin can map the physical world to the digital world by using novel hybrid simulation and data-driven modeling approach [6]. Therefore, the digital twin model can support design decisions, function tests, logic verifications, and performance statistics for the complex system [7].

*Correspondence:
Chunxiang Gu
gcx5209@126.com
[1] Henan Key Laboratory of Network Cryptography Technology, Zhengzhou 450000, China

To evaluate and test the functionality, logic, and security of network protocols in actual operation, model learning methods have been studied and applied in both researches and practical applications [8–12]. These methods employ automata learning algorithms and corresponding learning frameworks to automatically establish a state-machine model of a protocol device's behavior. The inferred state-machine model is then analyzed to uncover vulnerabilities and violations thus improving the security of the protocol systems. Serving as a complementary technique in model-driven digital twin construction, model learning is characterized by its simplicity, intuitiveness, and broad adaptability. It offers a focused lens to examine the execution logic of protocols and identify anomalous behaviors [13–15].

Most current protocol automata learning are based on black-box setup, focusing exclusively on I/O interactions. They assume that the model is finite and employ learning algorithms (such as L* [16] or TTT [17]) to infer hypotheses of the model. They also set an upper limit, known as the testing depth, for input sequence combinations and perform equivalence queries, such as the Wp method [18], to confirm if the inferred hypothesis aligns with the true model. While this black-box approach is straightforward and requires minimal setup, it tends to produce models that only reflect observable interactions, neglecting the internal complexities of the program. This limitation gives rise to several challenges in inferring and analyzing protocol state machines:

1. Difficulty in inferring accurate models: Black-box learning verifies the hypotheses through equivalence tests constrained by a set testing depth [8–10]. If the testing depth is too low, the model may miss critical behaviors, such as a backdoor activated by a series of repeated messages. However, increasing the depth exponentially elevates query costs, particularly in worst-case scenarios. Restricted by learning time and query limitations, black-box learning struggles to uncover deep and concealed protocol states, impeding the inference of an accurate model.
2. Limitations in identifying security risks: Since black-box learning relies entirely on I/O observations, it lacks the capability to perceive or interpret other vital information within the protocol such as cryptographic primitive algorithms [11, 12, 19]. This limitation not only hampers the discovery of logical issues at different program state nodes—such as authentication bypasses or key leakages—but also impedes the detection of code implementation problems, such as memory leaks, buffer overflows, or null pointer invocations.
3. Challenges in filtering redundant queries: To ensure the completeness of the state-machine model, black-box learning typically queries all messages in every possible state. However, for certain states where the tested program has terminated its interactions—such as disconnection or periods of implicit silence—these queries become not only time-consuming but also minimally informative.

The aforementioned issues arise from the conventional black-box learning framework's exclusive reliance on I/O observations. Mere parameter modifications cannot overcome these limitations. Some scholars proposed potential solutions including learning register automatas from source code based on taint analysis [20], inferring state machines via symbolic execution [21], and deriving models from specification documents [22]. However, these methods are generally applicable only for small-scale protocols or fall short in revealing vulnerabilities in real-world implementations.

To enhance the perception of state machines, a more comprehensive approach is required. In this study, we propose a state-aware model learning method that synergistically integrates network interaction I/O with filtered dynamic lifetime memory variables. This integration enables the inferred state-machine model to provide a more nuanced understanding of the system under learning (SUL), increasing the probability of discovering deep-level abnormal states and behaviors. As illustrated in Fig. 1, the first step involves instrumenting the SUL to monitor variables used during testing. Subsequently, the learning engine generates multiple rounds of message queries based on the learning algorithms. A mapper retrieves the results from these interactions with the SUL, while an execution monitor gathers state calibration information from filtered memory variables. Utilizing both I/O data and state calibration information, the learning engine incrementally constructs the protocol extended finite state machine (EFSM). This workflow continues until the learning process is completed, resulting in the final output of the state-machine model. For the inferred state machine, further analysis can be conducted to identify the potential suspicious paths that correspond to the search for program vulnerabilities.

We applied this novel method to develop SALearn, focusing on IKEv2 as our protocol for analysis. In recent studies, a lightweight version of the IKEv2 protocol has been standardized [23] and has been applied for multiple security schemes in edge/cloud environments [24–26]. We conducted tests on two widely used IKEv2 implementations—StrongSwan and Libreswan—and compared our method with existing model learning methods. Our experimental results demonstrate that state-aware model

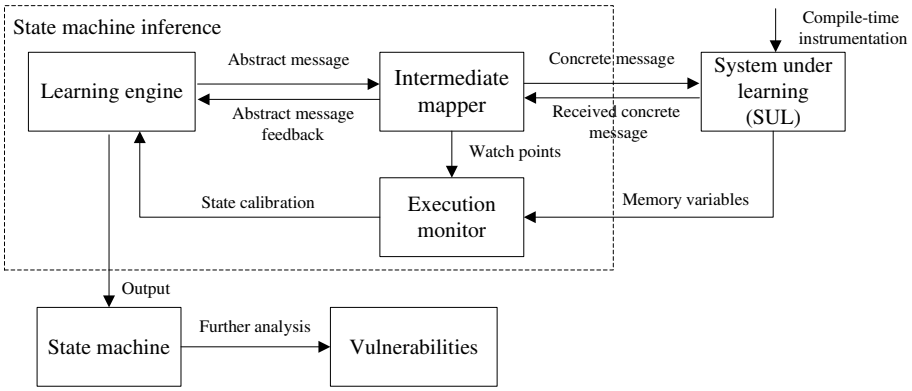

Fig. 1 Basic framework of state-aware based model learning method

learning not only reduces the learning time but also exhibits greater adaptability, enabling the discovery of more intriguing states and security issues. Specifically, we identified two violation scenarios as follows: abnormal management when rekeying IPsec SA in Strongswan and anomalous logic of rekeying IKE SA in Libreswan.

Our contributions in this study can be summarized as follows:

1. First, we propose a state-aware model learning method that enhances the quality of inferred states by correlating them more closely with the internal dynamics of the program. Automated model-driven digital twins reduce barriers to understanding the behaviors and analyzing the security for lightweight protocol implementations.
2. Second, we develop SALearn based on the state-aware model learning approach, designing a testing scheme and implementing a mapper for IKEv2 model learning.
3. Third, we conduct an extensive evaluation of two widely used IKEv2 implementations: Strongswan and Libreswan. Our results demonstrate that SALearn is more efficient in learning and is capable of discovering more interesting states and security issues. We also discuss the impacts of the two specific violations we identified.

The remainder of this paper is structured as follows: Related works section provides an overview of the related work in the fields of model learning and state-aware fuzzing test. Preliminaries section offers background knowledge of network protocol models and IPsec. State-aware model learning framework using digital twins section unveils a comprehensive framework and methodology for state-aware model learning. Experimental evaluation and analysis section introduces the results of our experimental evaluation and analysis. Finally, Conclusion section concludes the article and proposes potential directions for future research.

Related works

Program states serve as observable running attributes that can differentiate program behaviors or can be combined with certain specifications to determine the correctness of those behaviors. These states can further guide model inference or optimize adversarial test cases [27]. In the realm of network protocols, which are engineered to fulfill communication, program states can be expressed from two levels: how the program processes response events, known as its external representation, and the data context within which the program operates, known as its internal representation [28]. Academic research in protocol analysis and testing has been extensive, which includes model learning and state-aware fuzzing tests.

Model learning

Model learning, also known as state-machine inference, is a technique used for constructing state-machine models of both software and hardware systems. This is done by providing input and observing the corresponding output, which is crucial for comprehending the system's functionality and behavior [29]. Existing research in this field can be broadly classified into two categories: active learning, which involves generating queries and analyzing responses, and passive learning, which is based on collected samples. Our study predominantly focuses on the active learning approach.

The concept of model learning was formalized in 1987 when Angluin proposed the L* learning algorithm, providing a foundational framework for modeling reactive systems [16] . In 1999, Peled et al. [30] applied model learning to software analysis. More recently, the

model learning has been widely employed in network protocol testing [31]. For instance, Joeri de Ruiter and Poll [8] grabbed scholarly attention with their work on analyzing SSL/TLS protocols using state-machine inference. They created a tool called StateLearner for black-box state-machine inference and applied it to nine SSL/TLS implementations. Their approach led to the discovery of three security vulnerabilities, including client authentication bypasses, encrypted data leakages, and anomalies during rekeying. Stone et al. [9] conducted state-machine inference to seven Wi-Fi implementations and unearthed two downgrade attacks and one encrypted multicast leakage. Brostean et al. [10] inferred state-machine models from three SSH implementations, identifying seven subtle violations of specifications. Furthermore, Brostean et al. [11] proposed a protocol state fuzzing framework for DTLS to analyze 13 widely used DTLS servers and discover four security vulnerabilities, including client authentication bypass and handshake sequencing anomalies. Moreover, Brostean et al. [12] inferred state machines and constructed bug pattern catalogs to test implementation flaws. They tested 12 SSH and DTLS implementations and discovered 96 new vulnerabilities and errors.

Furthermore, model learning can be synergistically combined with other techniques such as symbolic execution, formal analysis, and fuzzing tests to broaden its scope and efficacy. For example, Marcovich et al. [22] combined model learning and symbolic execution to directly infer state machines and message formats from binary codes. They developed PISE, a tool based on this hybrid method, which successfully inferred the command-and-control (C &C) protocol state machine for the Gh0st RAT malware. In a similar vein, Wang et al. [32] combined model learning with formal analysis to create an automated solution, MPInspector, designed to scrutinize the security of message-passing (MP) protocols. When applied to nine popular IoT platforms—including MQTT, CoAP, and AMQP—MPInspector identified 252 attribute violations and proposed 11 different types of attacks within two realistic scenarios. Brostean et al. [33] developed a tool called DTLS-Fuzzer that couples model learning with fuzzing tests. The tool can generate DTLS state-machine models and subsequently perform fuzzing tests based on the states to uncover specification violations or security vulnerabilities. Similarly, Shu and Yan [19] explored a new heuristic method based on finite state machine inference to guide the generation of black box fuzzy test cases for IoT network protocol implementation. They implemented IoTInfer for Bluetooth and Telnet protocols. The experimental results indicate that IoTInfor can effectively generate meaningful test cases based on state guidance.

Despite the promise of such hybrid approaches, most inferred state models remain heavily dependent on I/O interactions and show a limited understanding of the broader program context. To address this issue, Stone et al. [34] combined runtime memory analysis with I/O observations developing a tool named StateInspector. Their approach enabled the exploration of deeper states within the program and was successfully tested on five TLS and two WPA/WPA2 protocol implementations, revealing two new CVEs in WolfSSL and IWD. Although effective for detecting backdoor behavior, this gray-box method largely focuses on candidate state variables in heap memory and may not be applicable to protocols that use stack memory or have system calls distributed across multiple subprocesses.

State-aware fuzzing test

State-aware fuzzing generally refers to tracing program variables in network protocols to perceive insight into program states, thereby optimizing test-case generation and identifying vulnerabilities. The called program state refers to the complete execution context of a running program, encompassing all variable values from a software perspective, as well as virtual memory and register states from a hardware perspective [28].These state variables are typically accessed and shared by different segments of the program and can influence control flows or memory access pointers either directly or indirectly. Therefore, comprehensively exploring these states is beneficial to uncover hidden vulnerabilities in a program.

Recently, the field of state-aware fuzz testing has seen significant advancements. Aschermann et al. [35] proposed IJON, an innovative mechanism using artificial annotations to trace program states, which allows fuzzers to explore a program's behavior more systematically. Experimental evaluations indicate that IJON, based on manual annotations, can explore deeper into program states than traditional fuzzing or symbolic execution tools. Fioraldi et al. [27] developed InsvCov, which uses program invariants as boundaries to partition a program's state space. It employs a combination of control flow and program invariants as feedback for fuzz testing. Similarly, Pham et al. [36] designed AFLNET, which leverages server response codes as indicators of program states in network protocols. AFLNET not only performs substantially better than its counterparts but also identified two new CVEs. Natella [37] designed StateAFL, a tool that employs localized sensitive hashing on long-lifetime variables to identify program states. It incrementally builds a protocol state machine for guiding the fuzzing process. Compared to AFLNET, StateAFL achieves comparable, if not superior, code coverage and produces more accurate state inferences. Ba et al. [38]

developed SGFuzz, which annotates program states based on various variable types like enumerations and macros. Experimental data indicates that SGFuzz discovers state errors and achieves code coverage more quickly than other mainstream stateful fuzzers. It has also unearthed eight new CVEs. Wen et al. [39] proposed MemLock, a memory-usage-oriented fuzzing technique. It uses extreme memory allocation values as additional feedback to detect memory leaks, outperforming other fuzzing techniques and identifying 15 new CVEs. Zhou et al. [40] presented Ferry, a state-aware symbolic execution approach that considers conditionally branched and input-influenced variables. Experimental evaluations demonstrate that compared to existing tools, Ferry achieves broader code coverage and triggers more states and vulnerabilities. Zhao et al. [28] developed StateFuzz, which employs static analysis to select long-lifetime, real-time updated variables that influence program control flow. It employs a unique feedback mechanism, proving effective in discovering new vulnerabilities and identifying 15 new CVEs. However, these state-aware approaches mainly focus on optimizing test-case generation using coarse-grained combinations of variables. They lack the nuance needed for fine-grained state calibration and accurate state-machine inference.

Our study takes a different approach. We combine network interaction I/O with carefully selected dynamic lifetime memory variables for a more nuanced state calibration. Compared to black-box model learning, our method offers a deeper understanding of the program's internal context. Unlike Stone et al.'s gray-box approach [34], we directly obtain runtime variable content, bypassing limitations imposed by stack memory and multiple subprocesses. Lastly, distinct from state-aware fuzzing, our focus is on inferring a detailed state-machine model to improve the efficiency of discovering both crash errors and logic issues, setting our work apart from traditional state-aware fuzzing.

Preliminaries

In this section, we introduce the foundational concepts of network protocol models and the IPsec-IKE protocol.

Network protocol model

In general, a network protocol model depends on a server operating in a continuous cycle of receiving, processing, and responding to requests. In this scheme, both the client and server participate in a session, marked by a series of request and response messages. This basic loop can be succinctly captured by the following pseudocode in Algorithm 1:

Algorithm 1 A network server process model

```
1 init_server()
2 while True do
3 |   relevant_variables = new allocate()
4 |   message = receive()
5 |   response = process(message, revelant_variables)
6 |   send(response)
7 |   clean_variables()
8 end
9 exit()
```

As the session progresses, driven by various external events, the internal execution of the program adjusts and adapts, leading to updates in the program's state. In this context, "state" encompasses both the program's external expected behaviors and internal runtime data.

Externally, the protocol's state is typically reflected in terms of the actions the process is allowed to take, which events it expects to happen, and how it will respond to those events [37]. Most internet protocols elucidate their standardized states either through natural language descriptions or, less commonly, through finite state machines, as comprehensively illustrated in RFC documents. These states are closely tied to the current stage of the protocol, where both inputs and outputs are fully contextualized within this framework.

Internally, the state of the program refers to a rigorously maintained execution environment that includes all presently active components. These facets are accessed and manipulated by various program operations, which in turn can impact the program's control flow or memory pointers [28]. During the protocol's execution, variables and data are created, used, and eventually released, residing in both heap and stack memory. These variables undergo updates as the server handles each request and response interaction. Nonetheless, among these variables, there exist indeterminate values, such as those associated with time, random numbers, and temporary keys. Although they impact the program's operations, they lack the desirability of distinguishing and designating a concrete state. In this paper, we focus on calibrating the state using variables that record previous program operations or user interactions, possess deterministic values for the same operation, and are tied to critical nodes such as the client's current authentication status or working directory [37]. By carefully filtering and monitoring these variables, we achieve a more nuanced understanding of the program's state.

IPsec-IKE protocol

IPsec [41–43] is a suite of security protocols that operate at the IP layer and is composed of three sub-protocols: IKE, AH, and ESP. The IKE protocol handles

cryptographic algorithm selection, session key negotiation, and identity authentication. By contrast, AH and ESP are used for secure communication.

IKE serves as the signaling protocol for IPsec, overseeing secure communications by maintaining security associations (SAs): repositories of cipher and identity information for communicating parties. Initially, these parties set up an IKE SA, which is followed by negotiating an IPsec SA within the encrypted channel of the existing IKE SA. The IPsec SA is used for secure ESP or AH communications, while the IKE SA manages this IPsec SA.

IKE exists in two versions: IKEv1 [42] and IKEv2 [43]. Compared to the multi-modal complexities of IKEv1, IKEv2 offers a streamlined approach with only four types of interactions:

1. Initial exchange (IKE_SA_INIT)
2. Authentication exchange (IKE_AUTH)
3. Creating child SA exchange (CREATE_CHILD_SA)
4. Informational exchange (INFORMATIONAL)

Figure 2 illustrates the generic interaction flow within the IKEv2 protocol. Detailed explanations of payload field meanings and symbol notations are available in RFC7296 [43].

In the initial phase of communication, the two parties commence with the negotiation of cryptographic parameters and identity authentication via IKE_SA_INIT and IKE_AUTH interactions, respectively. Consequently, separate IKE SA and IPsec SA instances are established. The IPsec SA specifies parameters for safeguarding data communication. Moreover, during the course of communication, the CREATE_CHILD_SA interaction allows rekeying procedures, facilitating the creation of new IKE SA or IPsec SA instances to ensure forward secrecy. The communication session concludes when both parties utilize notification payloads to delete the established IPSec and IKE SAs.

State-aware model learning framework using digital twins

In the following section, we delve into the intricate details of the overarching insight, framework, and methodology for state-aware based model learning. We describe the techniques for defining states, capturing relevant state variables, and the specific learning processes involved.

Insights from digital twins

Linking digital twins with a state-aware model learning approach for security analysis in lightweight protocol implementations entails combining these two principles to improve digital system knowledge and protection. We can integrate the proposed model with in the digital twins through a layer approach as shown in Fig. 3.

1. Digital twin layer: This layer represents the system's digital twins, which are virtual representations of physical elements. Device twin: A digital twin that represents particular system devices. Communication twin: A digital twin that records the channels and patterns of communication between devices. Data flow twin: A digital twin that depicts the data flow within the system. This layer's digital twins capture real-time data using instrumentation and sensors implanted into the represented things.
2. State-aware model layer: This layer incorporates the state-aware model learning technique, which entails developing models that comprehend the dynamic states and behaviors of lightweight protocols. Dynamic states and behaviors model: A model that encapsulates the lightweight protocol's numerous states and

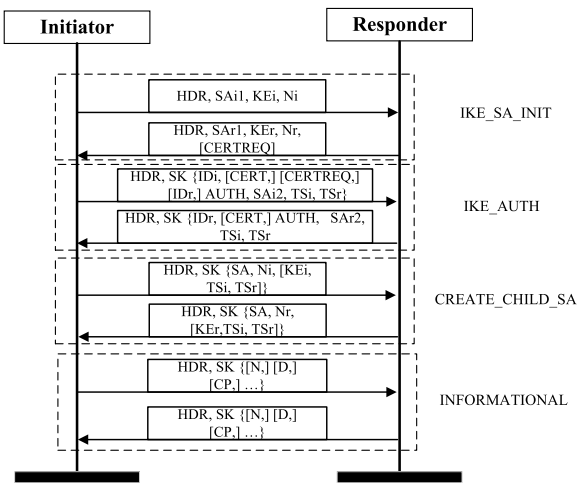

Fig. 2 General negotiation process for IKEv2

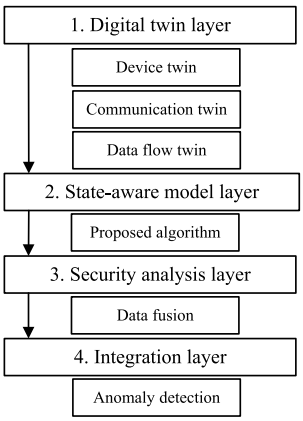

Fig. 3 Four layers of State-aware approach

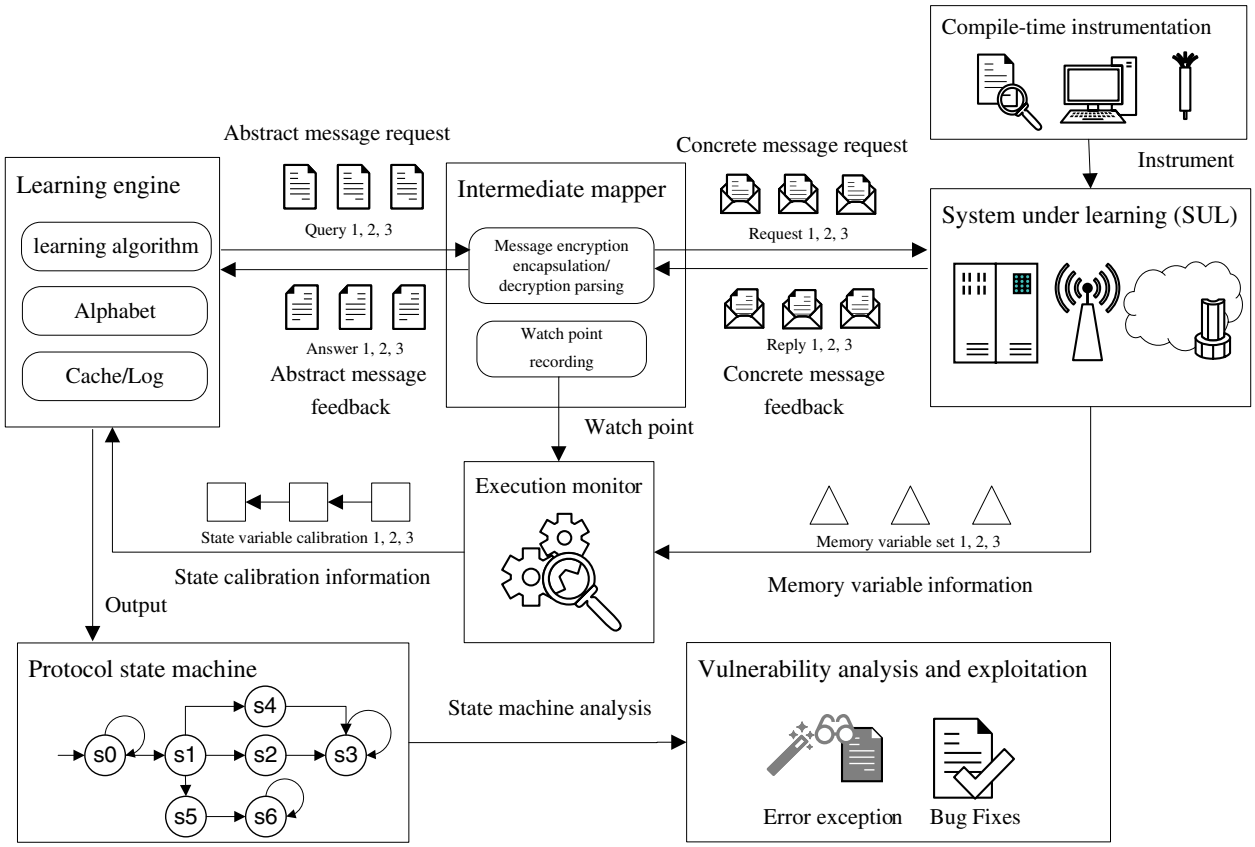

Fig. 4 Detailed framework of state-aware model learning method

behaviors throughout operation. Algorithms for state machine learning: Algorithms included into the model for learning and adapting to observable patterns. The state-aware model is intended to develop and learn from data gathered by digital twins.

3. Integration layer: This layer allows data from digital twins to be integrated with the state-aware model. Data fusion: The process of merging real-time data from digital twins with state-aware model knowledge. Correlation analysis: Methods for comparing data from digital twins to predicted states and behaviors provided by the state-aware model. This layer guarantees that the information gathered from the digital twins improves the state-aware model's learning capabilities.
4. Security analysis and response layer: The purpose of this layer is to use the integrated information for security analysis and response. Anomaly detection: Using integrated data, the system discovers abnormalities in lightweight protocol implementations. This layer tries to improve system security by proactively addressing possible security concerns.

Overall framework

Figure 4 describes the design of our state-aware based model learning framework, which comprises five major components: the state-machine learning engine, intermediate mapper, execution monitor, SUL, and vulnerability analysis and exploitation modules.

The state-machine learning engine employs a specific learning algorithm to generate multi-round message queries, process the results, and dynamically construct a protocol state-machine. It operates on an input alphabet consisting of abstract protocol messages. Guided by the learning algorithm, sequences of abstract message requests are formed and dispatched to the intermediate mapper for further processing. Upon receiving feedback sequences from the mapper and the state calibration identifications provided by the execution monitor, the learning algorithm incrementally constructs a protocol state machine, eventually yielding the final state-machine model. To enhance the learning process's efficiency and facilitate debugging, the engine comes equipped with features for query caching and logging.

The intermediate mapper serves as a critical intermediary, bridging the gap between abstract and concrete

messages. It also captures specific watch points during each packet transmission cycle. Functioning as a simulated client, the mapper converts abstract message queries from the learning engine into actionable packet requests for the SUL. After the SUL processes these packet requests, the mapper converts the resulting concrete messages back into abstract form for the learning engine's consumption. Each query cycle resets the SUL to its original state. The mapper processes each request in the query sequence sequentially, collecting the responses, and then furnishing the learning engine with a feedback sequence composed of abstract response messages. During these interactions, the intermediate mapper not only simulates the client's role but also maintains the session information with the SUL. If the message exchange involves encryption or decryption, the mapper handles these processes in real time, ensuring smooth interaction with the SUL. It also monitors and records various watch points related to SUL behavior during the learning phase—such as query initiation, encryption procedures, authentication completion, rekeying processes, disconnections, and query terminations—and communicates this data to the execution monitor, facilitating the selection of state-related variables.

The execution monitor plays a pivotal role in observing and filtering memory variables during the execution process of the SUL, it also calculates state calibration metrics in sync with specific watch points, thereby aiding the learning engine in crafting an accurate state machine. During each message interaction, the monitor identifies and logs variable values created by the SUL, focusing on those with lifetimes that align with the established watch points. Using hash calculations, it then derives the state calibration value pertinent to the current interaction. In this way, the execution monitor calculates the state calibration sequence that correlates with the query sequence from the learning engine, forwarding this information for the construction of the state machine.

The SUL functions as the protocol server under evaluation and can either be a white-box software library or a gray-box binary program that is susceptible to instrumentation. Before undergoing any tests, the SUL is compiled and instrumented to allow variable tracking during execution through shared memory.

The vulnerability analysis component scrutinizes the output state-machine model generated for the SUL, which involves using manual techniques or model checking methods to identify suspicious pathways within the state machine. These are then cross-referenced with predefined specifications or debugging tools to identify violations.

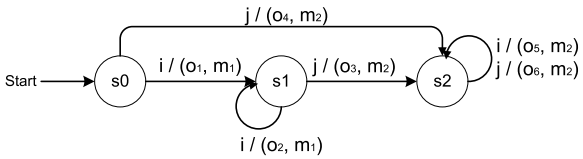

Fig. 5 An example of state machine

Description of state

To describe the state model, enriched with both I/O and memory information, we introduce the following definitions:

Definition 1 An alphabet is a finite, nonempty set of single letters, denoted as Σ, such as $\Sigma = \{a, b, ...\}$. The concatenation of letters is denoted as $a \cdot b$, where the concatenation symbol $\cdot$ can be omitted.

Definition 2 An extended finite state machine (EFSM) with memory calibration based on alphabet Σ can be represented by a seven-tuple $A =< S, s_0, I, O, M, \delta, \lambda >$, where

1. S is a finite, nonempty set of states;
2. $s_0 \in S$ is the initial state;
3. $I \subseteq \Sigma$ is an input set;
4. $O \subseteq \Sigma$ is an output set;
5. M is a set of memory calibrations;
6. $\delta : S \times \Sigma \to S$ is the state transition function, where $\delta(s, i) = s_1$ means that the state machine accepts input i in state s and transitions to s_1.
7. $\lambda : S \times \Sigma \to O \times M$ is the state output function, where $\lambda(s, i) = (o, m)$ indicates that the state machine accepts input i in state s, generates output o, and updates its memory calibration to m.

State machines with memory calibration have graphical representations that describe states as nodes and state transitions as edges. Figure 5 illustrates a simple graphical example. When the state machine is in state $s_0 \in S$ and receives input $i \in I$, the state transition $\delta(s_0, i) = s_1$ and the output $\lambda(s_0, i) = (o_1, m_1)$ correspond to an edge from s_0 to s_1 in the graph with the label $i/(o_1, m_1)$. Because memory-calibrated strings are only used to distinguish between different states and not as inputs for state transitions, they can be omitted from graphical representations.

Definition 3 A finite sequence of concatenated letters is called a string and is denoted as ω, and the set of strings it forms is denoted as Σ^*. The concatenation of letters can be extended to the concatenation of strings, denoted as $\omega_1 \cdot \omega_2$. In particular, ϵ denotes the empty string with a

length of 0. Similarly, $I^* \subset \Sigma^*$ represents the set of input containing strings.

Definition 4 δ can be extended to Σ^* by defining $\delta^* : S \times \Sigma^* \to S$, where

$$\delta^*(s, \epsilon) = s, \forall s \in S, \tag{1}$$

$$\delta^*(s, a \cdot \omega) = \delta^*(\delta(s, a), \omega), \forall s \in S, a \in \Sigma, \omega \in \Sigma^*. \tag{2}$$

δ denotes a special form of δ^* when the length of the second parameter is 1. Since it does not cause ambiguity, δ can be used instead of δ^* to represent the state transition function.

Furthermore, the state output function $\lambda(s, i)$ of A can be extended to Σ^*, that is, $\lambda(s, a \cdot \omega) = \lambda(s, a) \cdot \lambda(\delta(s, a), \omega)$.

Definition 5 Binary relation $\sim_\lambda$ is defined for $\forall s, s^{'}, s \sim_\lambda s^{'}$ if and only if $\forall i \in I^*, \lambda(s, i) = \lambda(s^{'}, i)$. It can be observed that this binary relationship is equivalent. The two equivalent states imply that the same input always produces the same output.

Definition 6 Based on the equivalence relation $\sim_\lambda$, an EFSM $M =< S_M, s_{0,M}, I_M, O_M, M_M, \delta_M, \lambda_M >$ is constructed for state-aware model learning, where the subscript M is only used to distinguish the state machine, and is constructed as follows:

1. $S_M = S/\sim_\lambda$;
2. $s_{0,M} \in S_M$ is the initial state;
3. I_M is a set of input strings that is a subset of Σ^*;
4. O_M is a set of output strings;
5. M_M is a set of memory calibrations;
6. $\delta_M(s_M, i_M) = s_M^{'}$, where under the action of the equivalence relation $\sim_\lambda$, δ_M transitions the state from s_M to $s_M^{'}$ upon receiving input i_M;
7. $\lambda_M(s_M, i_M) = (o_M, m_M)_{\sim_\lambda}$, where under the action of the equivalence relation $\sim_\lambda$, λ_M generates output o_M and updates the memory calibration information to m_M upon receiving input i_M in state s_M.

Capture memory state variables

In the EFSM model, states are described by a combination of network I/O behaviors and memory calibrations. Memory calibration, calculated from a specific set of memory variables during state transition, plays a significant role in determining the program's behavior through that transition. We elaborate on these concepts as follows:

Definition 7 Dynamic lifetime state variables refer to a group of memory variables that consistently produce the same value for identical inputs during a state transition instigated by message interactions. Importantly, the lifetime of these variables encompasses the time interval defined by the watch point range for that particular transition.

Watch point range refers to the time interval divided by the watch point. We define the watch points as follows:

Definition 8 Watch points refer to specific temporal markers associated with the start or end of learning queries, as well as any alterations in security attributes that may occur during program runtime.

In this paper, we provide six watch points: initiation of queries, enabling of encryption, completion of authentication, rekeying, disconnection, and query termination. These watch points unfold chronologically from the start of each query round. Scenarios may arise where these watch points overlap (such as enabling of encryption and completion of authentication at the same time) or are absent by default (such as the absence of rekeying). Memory variables according to these watch points are used to refine state calibration, thereby making state partition more relevant to the protocol's security attributes.

To collect real-time feedback regarding the protocol's state, we need to instrument probes within the target program's code during its compilation phase. These probes consist of external functions that are invoked during specific conditions. Running concurrently with the SUL, these probes detect triggering conditions and invoke external functions to either update or trace corresponding state memory variables. The probes are strategically placed in five specific code locations: query_start, memory_allocate, interaction_update, memory_deallocate, and query_end. The specific functionalities of these probes are detailed in Table 1.

These functions invoked by the probes are initialized at the start of a query and continuously monitor relevant variables through both memory allocation and deallocation phases. During each iteration of interaction, these tracked data are updated to reflect the shifts in state variables. At the termination of the query, the functions sift through and identify dynamic lifetime memory variables, computing the state's identification in the process. Information such as the start and end times of queries, interaction cycles, and watch points is transmitted to the probe's external function through shared memory by the intermediate mapper. The architecture of this system is illustrated in Fig. 6.

Calculate state identification

Once a query round concludes, the monitor scrutinizes the memory variables involved in the interactions. It

then refines this list to further filter and determine the dynamic lifetime memory variables. These variables are used to compute unique state identifiers, commonly referred to as "state_id," for each state transition.

Algorithm 2 Filter the dynamic lifetime memory variables and calculate the state identification

Input : Traced variables record TR, watch point record WR, total number of interactions N
Output: State identification sequence S

$S, H, SR \leftarrow newlist()$
for $v \in TR$ **do**
  if $cover\_any\_range(v, WR)$ **then**
    $SR.append(v)$
  end
end
for $i \leftarrow 1$ *to* N **do**
  $cr \leftarrow current\_range(i, WR)$
  for $v \in SR$ **do**
    if $cover\_current\_range(v, cr)$ **then**
      $H.append(v.get\_content(i))$
    end
  end
  $s \leftarrow sort\_and\_calculate\_hash(H)$
  $S.append(s)$
  $H \leftarrow newlist()$
end
return S

As shown in Algorithm 2, the process for computing state identification involves several steps. First, it excludes short lifetime variables that do not span any watch point range. Then, as interactions incrementally progress from the initial number, the variables encompassing the current watch point range are selectively filtered. These filtered variables are sorted, and their hash values are computed to serve as the state identifiers for the ongoing program interactions. Utilizing this methodology, the monitor calculates a sequence of state identifiers that align with the query sequence of the learning engine, facilitating the construction of a state machine.

Table 1 Probe types and descriptions

Probe	Description
Query_start	Initialize the record data at the beginning of the query
Memory_allocate	Record the address, size, and value of a memory area when it is allocated on the heap or stack
Interaction_update	Update the interaction number and add record values of tracked variables
Memory_deallocate	End the tracking of a memory when it is deallocated
Query_end	Determine the dynamic lifetime memory variables and calculate the state identification hash when the query terminates

Overall learning process

Our learning methodology is grounded in this state description and state identification method. The inputs for the learning algorithm include an alphabet, a happy flow (which is a standard negotiation sequence based on the given alphabet), a configurable depth for equivalence checking, an optional dictionary of variables that are irrelevant to the SUL states, and a optional number of dry run repetitions. The output is a finely inferred state-machine model.

Algorithm 3 State aware based model learning

Input : Alphabet Σ, happy flow HF, equivalence depth e, state-unrelated dict D, number of dry run repetitions r
Output: Extended finite state machine M

$M \leftarrow newstate\_machine()$
$QR \leftarrow newlist()$
$DQ \leftarrow dry\_run(HF)$
for $i \leftarrow 1$ *to* r **do**
  for $q \in DQ$ **do**
    $QR.append(query(q))$
  end
end
$update\_dict(QR, D)$
$update\_machine(QR, M)$
$l \leftarrow 1$
repeat
  $QR = generate(M, l, \Sigma)$
  for $q \in QR$ **do**
    $query\_and\_update(q, M)$
    $handel\_new\_state(q, M, D)$
  end
  $check\_state\_merge(M, D, e)$
  $l \leftarrow l + 1$
until $model\_complete(M, l)$
return M

As shown in Algorithm 3, the learning process unfolds in two primary phases. The first phase centers on gathering preliminary data and building an initial state-machine model. Leveraging the happy flow, initial query sequences are assembled and executed in a loop, with the monitor tracking and capturing relevant memory variables. The monitor uses the dictionary of state-irrelevant variables along with a differential comparison method to filter out unrelated variables. It also updates this filtering dictionary in real time. Upon capturing this data, the monitor assembles an initial state-machine model founded on the query outcomes. The second phase entails additional scrutiny of state variables and further refinement of the state-machine model. The algorithm generates query sequences

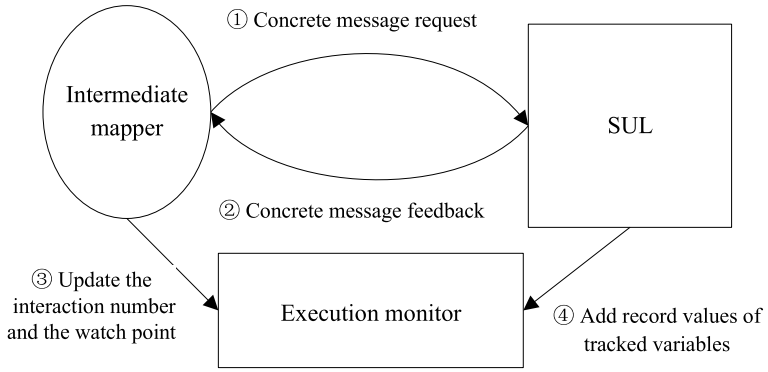

Fig. 6 Memory variables tracking architecture

based on the existing alphabet and model. These queries are then iteratively executed, and the results are integrated into the state transition paths of the model. Specifically, if a new state is uncovered, the algorithm reiterates the query, employs differential testing to eliminate potential state-irrelevant variables, updates the variable dictionary, and locks in the precise state identification. Moreover, the algorithm performs equivalence checks at each iteration, contemplating the consolidation of states with identical state identification. This iterative process continues until no new states emerge, culminating in a comprehensive and finalized state-machine model.

Specifically, the methodology for equivalence checking operates as follows: Starting from a designated initial state, if two distinct sequences ultimately lead to states sharing the same state identification, the algorithm delves further. It either seeks I/O sequences that can differentiate these similar states, guided by the pre-set equivalence checking depth, or it evaluates the feasibility of merging these states into one. This added layer of scrutiny ensures a robust and accurate representation of states within the final state-machine model. In addition, we set up judgment and optimization for the disconnected state, which accurately identifies the normal communication end state through state identification, and no further query testing will be performed in this state.

Experimental evaluation and analysis

In this section, we delve into the experimental results, the inferred models, and the issues identified, all stemming from our state-aware model learning framework, SALearn.

IPsec implementations

We evaluated SALearn using two mainstream open-source IPsec implementations: Strongswan and Libreswan. These implementations offer a range of IKEv2-based VPN functionalities, as outlined in Table 2.

Our experiments were conducted on two platforms:

The model learning platform: Ubuntu 20.04 OS + 11th Gen Intel (R) Core (TM) i9-11900 CPU + 32GB RAM + 1T ROM

SUL platform: Ubuntu 20.04 OS + Intel Xeon E5-2680 v2 CPU + 32GB RAM + 4T ROM

Learning scheme for IPsec

To develop a comprehensive model for IPsec, we utilized alphabets based on RFC7296 to facilitate learning under IKEv2 protocols.

In IKEv2, the alphabet focused primarily on standard negotiation messages and additional messages for rekeying, deleting, and testing both IKE and IPsec SAs, detailed in Table 3. Importantly, we differentiate between new and old SAs based on their creation time, assigning labels that allow the mapper to retrieve them in chronological order.

Table 2 Information of IPsec SUL

SUL	Version	Auth method	IKEv1/IKEv2 Support	Description
Strongswan	5.9.9	Cert+PSK	All	An open-source, modular, and cross platform IPsec VPN solution developed based on the FreeS/WAN project, but completely rewritten
Libreswan	4.11	Cert+PSK	All	An IPsec implementation for Linux that supports IKEv1, IKEv2, and most IPsec related extensions. Based on the FreeS/WAN code library, it has been extended on this basis

Table 3 IKEv2 learning alphabet

Alphabet	Message type	Description
IKE_SA_INIT	IKE_SA_INIT 34	IKE SA initialization
IKE_AUTH_CERT	IKE_AUTH 35	IKE certificate authentication
Rekey_IKE	CREATE_CHILD_SA 36	Create new IKE SA
Rekey_ESP_over_Current_IKE	CREATE_CHILD_SA 36	Create new IPsec SA(ESP) over current IKE SA
Rekey_ESP_over_Old_IKE	CREATE_CHILD_SA 36	Create new IPsec SA(ESP) over old IKE SA
Delete_Current_ESP_over_Current_IKE	INFORMATIONAL 37	Delete new IPsec SA(ESP) over current IKE SA
Delete_Old_ESP_over_Current_IKE	INFORMATIONAL 37	Delete old IPsec SA(ESP) over current IKE SA
Delete_Current_ESP_over_Old_IKE	INFORMATIONAL 37	Delete new IPsec SA(ESP) over old IKE SA
Delete_Old_ESP_over_Old_IKE	INFORMATIONAL 37	Delete old IPsec SA(ESP) over old IKE SA
Delete_Current_IKE	INFORMATIONAL 37	Delete current IKE SA
Delete_Old_IKE	INFORMATIONAL 37	Delete old IKE SA
Test_Current_IKE_Current_ESP	ESP	Test current IPsec SA(ESP) over current IKE SA
Test_Current_IKE_Old_ESP	ESP	Test old IPsec SA(ESP) over current IKE SA
Test_Old_IKE_Current_ESP	ESP	Test current IPsec SA(ESP) over old IKE SA
Test_Old_IKE_Old_ESP	ESP	Test old IPsec SA(ESP) over old IKE SA

After a successful rekeying operation based on RFC7296, the mapper transitions the IPsec SA from the old to the new IKE SA, while maintaining the previously assigned labels. It is worth noting that the unrestricted creation of SAs can lead to models that are overly complex and difficult to analyze. Therefore, we imposed a constraint: each IKE SA can have no more than two associated IPsec SAs. If a rekeying request exceeds this limit, the mapper returns a "None," thereby keeping the number of SAs within our defined range.

To enhance learning efficiency, we implemented several optimization strategies and integrated a message inspection mechanism to aid in the detection of anomalous behavior.

1. Mapper configuration and optimization: We configured reasonable timeout periods for each message type to speed up transmission. The mapper's message-parsing mechanism was also fine-tuned to reduce inconsistent feedback. To build on this, we introduced a query result cache. If a test's query sequence yielded a result that differed from a cached response, we conducted multiple retests until a consistent response was confirmed as the final query outcome. To ensure reliable interactions, TCP transmission is employed between the learning machine and the mapper.
2. Time synchronization: A synchronized clock is maintained between the intermediate mapper and the execution monitor, ensuring accurate interaction counts and watchpoints.
3. State-machine model refinements: In the final learning state-machine model, edges with identical start and end points are combined to simplify model inspection and analysis. For transitions lacking practical significance-such as querying to delete an IPsec SA when none exists-the mapper directly returns a "None" response. As a result, irrelevant edges are omitted from the final model.
4. A mechanism is integrated into the mapper to scrutinize both incoming and outgoing messages. This feature checks for behaviors that deviate from expected responses and flags abnormal messages, which could be indicative of service crashes or other issues.

Learning result

We tested two leading IPsec implementations and compared the performance of our approach, SALearn, with two existing tools: StateInspector [34] and StateLearner [8]. The statistical outcomes of the learning process are summarized in Table 4.

Table 4 Statistics of state-aware model learning

SUL	Protocol	Associated variables	Queries	States	Time (hh:mm)	Queries	States	Time (hh:mm)	Queries	States	Time (hh:mm)
		SALearn				StateInspector			StateLeaner		
Strong-swan	IKEv2	122	672	29	00:41	-	-	-	9536	20	12:15
Libreswan	IKEv2	47	1127	57	01:33	562	39	00:40	17634	43	23:42

The results reveal that SALearn significantly outpaces the black-box learning tool StateLearner in terms of both the number of queries required and the time needed for model inference. This efficiency stems from SALearn's use of correlated state variables, which allows for quicker differentiation between distinct states without the need for exhaustive queries. This ability to finely distinguish states also results in a state-machine model with greater complexity and depth. When compared to the gray-box tool StateInspector, SALearn demonstrates comprehensive model inference capabilities. StateInspector's limitations become evident when handling StrongSwan, as it can only trace the system calls in a single process. This constraint makes it incapable of capturing the right time to take a memory snapshot to filter candidate state memory for state-machine inference. Further, SALearn's fine-grained variable calibration leads to the inference of models with a greater number of states on Libreswan, as compared to StateInspector.

Case analysis

In this subsection, we discuss the characteristics and shortcomings of these implementations based on the state-machine model inferred by SALearn. Presenting the complete model would be cumbersome; therefore, we have optimized it for clarity. First, we removed certain non-essential transition paths to highlight the crucial aspects of the model. For example, in the case of IKEv2, some key IKE and rekey ESP transitions were omitted. Second, irrelevant transitions were eliminated. Transitions with a "None" response have been excluded. Finally, transitions with identical starting and ending nodes were merged, and any remaining unspecified input strings for the state were represented as "Other."

To further clarify the state machine, each model features bold lines to indicate normal negotiations (happy flows). Red dotted lines mark paths with issues were identified. Labels on the edges of the state-machine model correspond to the current state's inputs and outputs, denoted by "IO." Abbreviations are used to represent corresponding abstract messages. Labels for messages that resulted in meaningful responses and led to state transitions are highlighted in larger font sizes.

1. Strongswan IKEv2

Figure 7 presents a streamlined state-machine model of StrongSwan operating under the IKEv2 alphabet. The interaction sequence goes from states s0 to s1 and finally to s2, where IKE_SA_INIT and IKE_AUTH occur. In state s2, both parties establish IKE and IPsec SAs, thereby enabling Encapsulating Security Payload (ESP) communications. Advancing from s2 to s3, and finally to s4, involves deleting the active IPsec and IKE SAs, culminating in a communication termination. Furthermore, StrongSwan prohibits the insertion of unexpected messages during negotiations prior to authentication. As a result,

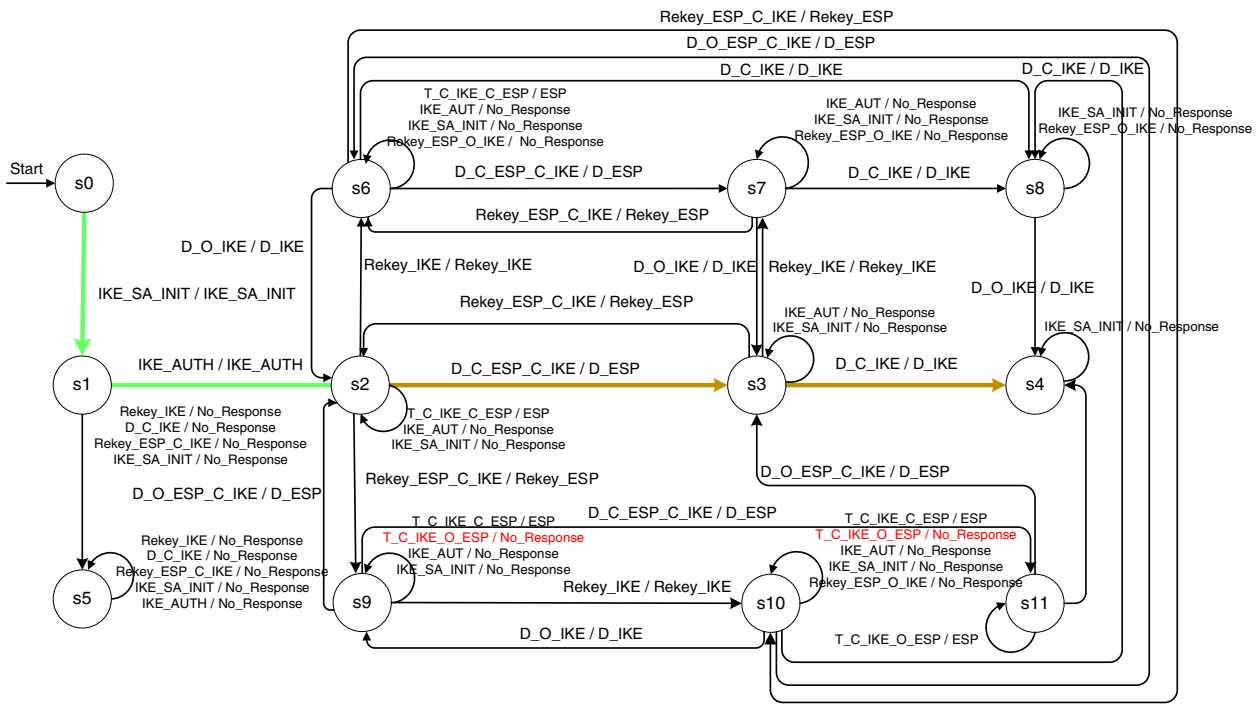

Fig. 7 Simplified state-machine model of StrongSwan with IKEv2 alphabet

submitting an abnormal authentication request in state s1 terminates the negotiation and transitions to state s5.

States s6, s7, s8, s9, s10, and s11 describe various scenarios that involve rekeying or deletion of IKE or IPsec SAs after completing authentication in state s2. Some of these states are interchangeable in terms of transitions. For instance, state s2 can morph into either s6 (by rekeying the IKE SA and then deleting the previous one) or s9 (by rekeying the IPsec SA and discarding the old one). Importantly, when multiple IPsec SAs are active, StrongSwan only supports ESP communication via the most recently established SA, such as in states s9 and s10. Whether initiating direct ESP creation (transitioning from s2 to s9), rekeying IKE before ESP creation (going from s2 to s6 and then to s10), or creating an ESP prior to IKE rekeying (transitioning from s2 to s9 and then to s10), StrongSwan exclusively facilitates ESP communication through the latest tunnel. Deleting the most recent ESP enables communication through the older, still-active ESP (transitioning from s9 to s11).

2. Libreswan IKEv2

Figure 8 presents a simplified state-machine model of Libreswan operating under the IKEv2 alphabet. Like StrongSwan, the sequence from state s0 to s1 and then to s2 covers interactions involving IKE_SA_INIT and IKE_AUTH. Transitions from s2 to s3 and then to s4 are concerned with deleting the active ESP and IKE, effectively terminating the connection. Regarding the management of multiple ESPs, Libreswan also supports ESP communication through the most recently created instance, as evidenced by the transition from s2 to s5. However, once the latest ESP is deleted, the older ESPs cannot be reactivated for communication, as illustrated by the transition from s5 to s6.

In contrast to StrongSwan, Libreswan demonstrates more flexibility in state s1. Specifically, after completing the IKE_SA_INIT interaction, Libreswan can still complete IKE_AUTH requests after receiving some unexpected request. Moreover, Libreswan allows the creation of an ESP on an older IKE channel, facilitating communication through that newly established ESP, as seen in the transition from s7 to s8. Note that ESPs created on these older IKE channels interfere with ESP communication on the current channel. Even deleting the ESP established on the older IKE channel does not reinstate the communication capabilities of the ESP on the current IKE channel, as shown by the transition from s8 to s9.

Discussion

Based on the comparative analysis between SALearn and various model learning tools, as well as SALearn's performance in dissecting different IPsec implementations, several key observations emerge:

1. SALearn exhibits superior efficiency in both query and learning times when compared to the traditional black-box learning tool, StateLearner. This efficiency advantage becomes increasingly noticeable as the scale and complexity of the learning objects escalate.

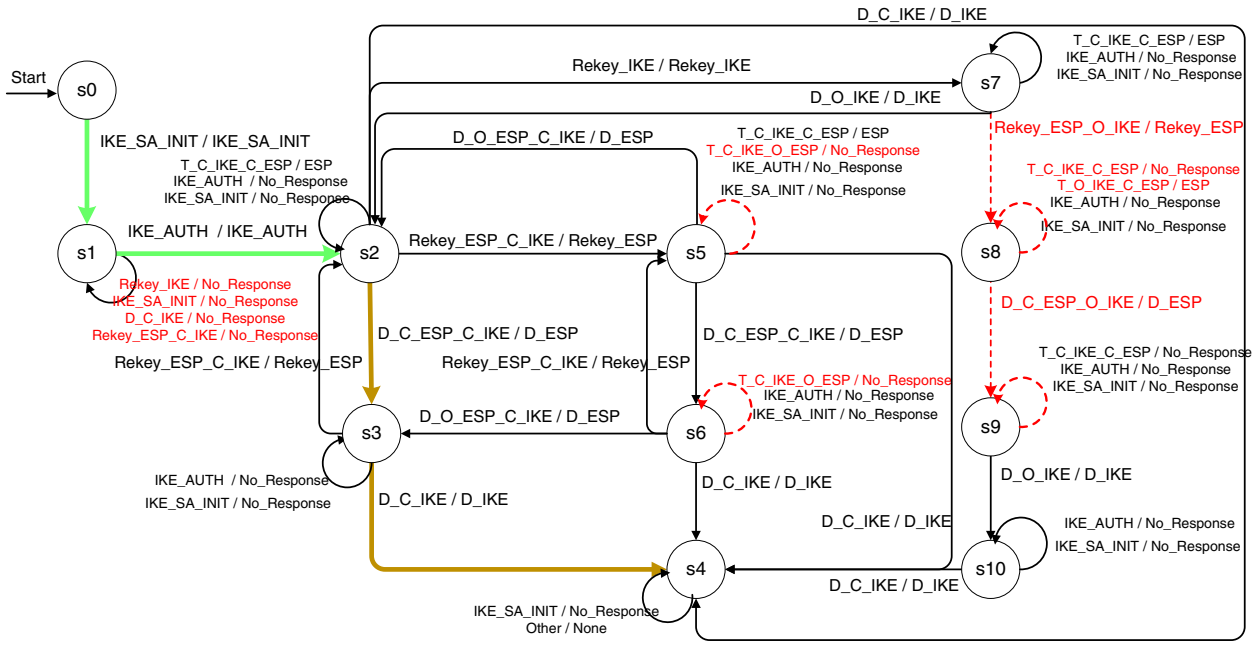

Fig. 8 Simplified state-machine model of Libreswan with IKEv2 alphabet

2. When compared with the similar gray-box model learning tool StateInspector, SALearn offers broader applicability across protocol implementations. It relies solely on the collection of code instrumentation data for learning, thereby circumventing the limitations of StateInspector, such as stack memory constraints and multi-process complexities.
3. SALearn is capable of inferring comprehensive state-machine models based on symbolic alphabets. These models faithfully encapsulate the intricate logic governing message interactions within the protocol implementations.
4. Analysis of the models generated by SALearn enables the identification of not only abnormal paths and logical inconsistencies that contravene protocol specifications but also uncovers potential anomalous states. These findings can catalyze the detection of issues in the protocol's code implementation.

Despite these strengths, SALearn is not without limitations:

1. For SALearn to operate effectively, the protocol code needs to be instrumented, thereby imposing higher learning requirements compared with StateInspector and StateLearner.
2. The learning process in SALearn necessitates the gathering and filtering of state-variable information, introducing computational overhead.
3. The task of collating state-variable information to identify abnormal paths and states in SALearn involves manual effort, adding to the overall complexity of the process.

Further discussion on the implications of identified anomalies:

1. Inconsistent IPsec SA Management in StrongSwan: The StrongSwan state machine under IKEv2 reveals that when multiple IPsec SAs are active, StrongSwan prioritizes communication through the most recently established ESP. Intriguingly, further packet analysis shows that StrongSwan crafts response messages using the latest ESP key, even if a test request is sent using a previously established ESP key. Conversely, upon deleting the newest ESP and sending a test request via the previous ESP, StrongSwan invariably replies using the second most recent ESP. According to Section 2.8 of RFC 7296, "to rekey a child SA within an existing IKE SA, create a new, equivalent SA, and when the new one is established, delete the old one." In contradiction to this specification, StrongSwan neither deletes the old IPsec SA upon establishing a new one, nor prevents communication between the old and new IPsec SAs. This behavior violates the standards and undermines forward secrecy.
2. Flawed Logic in IKE SA Updating in Libreswan: The IKEv2 state machine for Libreswan reveals that the system allows for the creation of ESP on older IKE channels, and such ESPs are functional for regular communication. In contradiction to Section 2.8 of RFC 7296, which states that "after the new equivalent IKE SA is created, the initiator deletes the old IKE SA, and the Delete payload to delete itself MUST be the last request sent over the old IKE SA," Libreswan neither mandates the deletion of the old IKE SA upon establishing a new one, nor restricts ESP creation on the old IKE SA. This lapse contradicts the RFC guidelines and compromises forward secrecy.

Conclusion

In this research, we introduce a state-aware model learning approach aimed at bridging the gap between inferred states and the inner workings of a protocol device. By synchronizing network interaction I/O and filtering dynamic memory variables across their lifetimes, our method refines the inferred state machine to include more relevant elements. Our evaluation, covering two widely used IKEv2 implementations, reveals two instances where these systems violate expected behavior. The results validate the efficacy of our approach; a digital twin paired with a state-aware model can give insights into predicted communication patterns in edge computing scenarios where devices connect with each other in a decentralized way utilizing lightweight protocols. Based on the learnt behaviors, the system may update its security measures in real-time, detecting any abnormalities or potential attacks on lightweight protocol implementations.

Regarding memory information management, the current approach is somewhat simplistic, focusing primarily on the lifespan and values of memory variables for filtering. This approach leaves much to be desired in terms of the precision and interpretability of these variables within the context of the program. To address this issue, future work could involve several improvements. First, utilizing more specific variables like macro definitions or enumerated types could refine state calibration and enhance its interpretability. Second, considering advanced analysis methods such as symbolic execution or taint analysis could provide a more accurate understanding of how variables influence program states, thereby honing the precision of state inference. Additionally, a formal analysis model could be developed based on the inferred state paths and calibrated variables to assess compliance with security requirements. Taken together, this research not only broadens the methodologies available for protocol model learning but also offers valuable insights into enhancing the security analysis of network protocols.

Authors' contributions
J. G. and D. Z. designed the research and wrote the paper. C. G. and X. C. analyzed the proposed research. X. Z. and M. J. made suggestions on research methodology and experimental evaluation. All authors reviewed and edited the manuscript.

Funding
This work was supported by the National Natural Science Foundation of China (Grant No. 61772548), the Science Foundation for the Excellent Youth Scholars of Henan Province (Grant No. 222300420099), and the Major Public Welfare Projects in Henan Province (201300210200).

Availability of data and materials
Not applicable.

Declarations

Ethics approval and consent to participate
This article does not contain any studies with human participants or animals performed by any of the authors.

Competing interests
The authors declare no competing interests.

Received: 26 October 2023 Accepted: 6 January 2024
Published online: 30 January 2024

References

1. Ahmad I, Niazy MS, Ziar RA, Khan S (2021) Survey on iot: security threats and applications. J Robot Control (JRC) 2(1):42–46
2. Li W, Wu J, Cao J, Chen N, Zhang Q, Buyya R (2021) Blockchain-based trust management in cloud computing systems: a taxonomy, review and future directions. J Cloud Comput 10(1):1–34
3. Vaezi M, Azari A, Khosravirad SR, Shirvanimoghaddam M, Azari MM, Chasaki D, Popovski P (2022) Cellular, wide-area, and non-terrestrial iot: A survey on 5g advances and the road toward 6g. IEEE Commun Surv Tutorials 24(2):1117–1174
4. Tao F, Xiao B, Qi Q, Cheng J, Ji P (2022) Digital twin modeling. J Manuf Syst 64:372–389
5. VanDerHorn E, Mahadevan S (2021) Digital twin: Generalization, characterization and implementation. Decis Support Syst 145:113524
6. Rasheed A, San O, Kvamsdal T (2020) Digital twin: Values, challenges and enablers from a modeling perspective. IEEE Access 8:21980–22012
7. Liu Y, Ong S, Nee A (2022) State-of-the-art survey on digital twin implementations. Adv Manuf 10(1):1–23
8. De Ruiter J, Poll E (2015) Protocol state fuzzing of tls implementations. 24th USENIX Security Symposium (USENIX Security 15). USENIX Association, Washington, D.C., pp 193–206
9. McMahon Stone C, Chothia T, De Ruiter J (2018) Extending automated protocol state learning for the 802.11 4-way handshake. In: Computer Security: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I 23. Springer, pp 325–345
10. Fiterău-Broştean P, Lenaerts T, Poll E, de Ruiter J, Vaandrager F, Verleg P (2017) Model learning and model checking of ssh implementations. In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software. Santa Barbara, pp 142–151
11. Fiterau-Brostean P, Jonsson B, Merget R, De Ruiter J, Sagonas K, Somorovsky J (2020) Analysis of dtls implementations using protocol state fuzzing. In: 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, ELECTR NETWORK, pp 2523–2540
12. Fiterau-Brostean P, Jonsson B, Sagonas K, Tåquist F (2023) Automata-based automated detection of state machine bugs in protocol implementations. In: NDSS. Internet Society, San Diego
13. Bordeleau F, Combemale B, Eramo R, van den Brand M, Wimmer M (2020) Towards model-driven digital twin engineering: Current opportunities and future challenges. In: Systems Modelling and Management: First International Conference, ICSMM 2020, Bergen, Norway, June 25–26, 2020, Proceedings 1. Springer, Bergen, pp 43–54
14. Bibow P, Dalibor M, Hopmann C, Mainz B, Rumpe B, Schmalzing D, Schmitz M, Wortmann A (2020) Model-driven development of a digital twin for injection molding. In: International Conference on Advanced Information Systems Engineering. Springer, ELECTR NETWORK, pp 85–100
15. Kirchhof JC, Malcher L, Rumpe B (2021) Understanding and improving model-driven iot systems through accompanying digital twins. In: Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. Association for Computing Machinery, Chicago, pp 197–209
16. Angluin D (1987) Learning regular sets from queries and counterexamples. Inf Comput 75(2):87–106
17. Isberner M, Howar F, Steffen B (2014) The ttt algorithm: a redundancy-free approach to active automata learning. In: Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings 5. Springer, Toronto, pp 307–322
18. Khendek FB, Fujiwara S, Bochmann G, Khendek F, Amalou M, Ghedamsi A (1991) Test selection based on finite state models. IEEE Trans Softw Eng 17(591–603):10–1109
19. Shu Z, Yan G (2022) Iotinfer: Automated blackbox fuzz testing of iot network protocols guided by finite state machine inference. IEEE Internet Things J 9(22):22737–22751
20. Howar F, Jonsson B, Vaandrager F (2019) Combining Black-Box and White-Box Techniques for Learning Register Automata. In: Steffen B, Woeginger G (eds) Computing and Software Science. Lecture Notes in Computer Science, vol 10000. Springer, Cham. https://doi.org/10.1007/978-3-319-91908-9_26
21. Marcovich R, Grumberg O, Nakibly G (2023) Pise: Protocol inference using symbolic execution and automata learning. In: Proceedings 2023 Workshop on Binary Analysis Research. Internet Society, San Diego
22. Pacheco ML, von Hippel M, Weintraub B, Goldwasser D, Nita-Rotaru C (2022) Automated attack synthesis by extracting finite state machines from protocol specification documents. In: 2022 IEEE Symposium on Security and Privacy (SP). IEEE, San Francisco, pp 51–68
23. Kivinen T (2016) Minimal internet key exchange version 2 (ikev2) initiator implementation. Tech. rep
24. Ko M, Kim H, Min SG (2022) An ikev2-based hybrid authentication scheme for simultaneous access network and home network authentication. IEICE Trans Commun 105(2):250–258
25. Rafique W, Qi L, Yaqoob I, Imran M, Rasool RU, Dou W (2020) Complementing iot services through software defined networking and edge computing: A comprehensive survey. IEEE Commun Surv Tutor 22(3):1761–1804
26. Cui Q, Zhu Z, Ni W, Tao X, Zhang P (2021) Edge-intelligence-empowered, unified authentication and trust evaluation for heterogeneous beyond 5g systems. IEEE Wirel Commun 28(2):78–85
27. Fioraldi A, D'Elia DC, Balzarotti D (2021) The use of likely invariants as feedback for fuzzers. In: 30th USENIX Security Symposium (USENIX Security 21). USENIX Association, ELECTR NETWORK, pp 2829–2846
28. Zhao B, Li Z, Qin S, Ma Z, Yuan M, Zhu W, Tian Z, Zhang C (2022) Statefuzz: System call-based state-aware linux driver fuzzing. In: 31st USENIX Security Symposium (USENIX Security 22). USENIX Association, Boston, pp 3273–3289
29. Neele T, Sammartino M (2023) Compositional automata learning of synchronous systems. International Conference on Fundamental Approaches to Software Engineering. Springer Nature Switzerland, Cham, pp 47–66
30. Peled D, Vardi MY, Yannakakis M (1999) Black box checking. In: International Conference on Protocol Specification, Testing and Verification. Springer, Beijing, pp 225–240
31. Zhu X, Wen S, Camtepe S, Xiang Y (2022) Fuzzing: a survey for roadmap. ACM Comput Surv (CSUR) 54(11s):1–36
32. Wang Q, Ji S, Tian Y, Zhang X, Zhao B, Kan Y, Lin Z, Lin C, Deng S, Liu AX, 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). USENIX Association, ELECTR NETWORK, pp 4205–4222
33. Fiterău-Broştean P, Jonsson B, Sagonas K, Tåquist F (2022) Dtls-fuzzer: A dtls protocol state fuzzer. In: 2022 IEEE Conference on Software Testing, Verification and Validation (ICST). IEEE, ELECTR NETWORK, pp 456–458

34. McMahon Stone C, Thomas SL, Vanhoef M, Henderson J, Bailluet N, Chothia T (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. Association for Computing Machinery, Los Angeles, pp 2265–2278
35. Aschermann C, Schumilo S, Abbasi A, Holz T (2020) Ijon: Exploring deep state spaces via fuzzing. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE, ELECTR NETWORK, pp 1597–1612
36. 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, Porto, pp 460–465
37. Natella R (2022) Stateafl: Greybox fuzzing for stateful network servers. Empir Softw Eng 27(7):191
38. Ba J, Böhme M, Mirzamomen Z, Roychoudhury A (2022) Stateful greybox fuzzing. In: 31st USENIX Security Symposium (USENIX Security 22). USENIX Association, Boston, pp 3255–3272
39. Wen C, Wang H, Li Y, Qin S, Liu Y, Xu Z, Chen H, Xie X, Pu G, Liu T (2020) Memlock: Memory usage guided fuzzing. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. Association for Computing Machinery, Seoul, pp 765–777
40. Zhou S, Yang Z, Qiao D, Liu P, Yang M, Wang Z, Wu C (2022) Ferry: State-aware symbolic execution for exploring state-dependent program paths. In: 31st USENIX Security Symposium (USENIX Security 22). USENIX Association, Boston, pp 4365–4382
41. Kent S, Seo K (2005) Rfc 4301: Security architecture for the internet protocol. RFC Editor, p 101. https://doi.org/10.17487/RFC4301. https://www.rfc-editor.org/info/rfc4301
42. Harkins D, Carrel D (1998) Rfc2409: The internet key exchange (ike). RFC Editor, p 41. https://doi.org/10.17487/RFC2409. https://www.rfc-editor.org/info/rfc2409
43. Kaufman C, Hoffman P, Nir Y, Eronen P, Kivinen T (2014) Rfc 7296: Internet key exchange protocol version 2 (ikev2). RFC Editor, p 142. https://doi.org/10.17487/RFC7296. https://www.rfc-editor.org/info/rfc7296

Publisher's Note