Alembic: Automated Model Inference for Stateful Network Functions

Soo-Jin Moon, *Carnegie Mellon University;* Jeffrey Helt, *Princeton University;* Yifei Yuan, *Intentionet;* Yves Bieri, *ETH Zurich;* Sujata Banerjee, *VMware Research;* Vyas Sekar, *Carnegie Mellon University;* Wenfei Wu, *Tsinghua University;* Mihalis Yannakakis, *Columbia University;* Ying Zhang, *Facebook, Inc.*

https://www.usenix.org/conference/nsdi19/presentation/moon

This paper is included in the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19).

February 26–28, 2019 • Boston, MA, USA

ISBN 978-1-931971-49-2

Open access to the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19) is sponsored by

Alembic: Automated Model Inference for Stateful Network Functions*

Soo-Jin Moon[1], Jeffrey Helt[2], Yifei Yuan[3], Yves Bieri[4], Sujata Banerjee[5]

Vyas Sekar[1], Wenfei Wu[6], Mihalis Yannakakis[7], Ying Zhang[8]

[1]*Carnegie Mellon University,* [2]*Princeton University,* [3]*Intentionet,* [4]*ETH Zurich*
[5]*VMware Research,* [6]*Tsinghua University,* [7]*Columbia University,* [8]*Facebook, Inc.*

Abstract

Network operators today deploy a wide range of complex, stateful network functions (NFs). Typically, they only have access to the NFs' binary executables, configuration interfaces, and manuals from vendors. To ensure correct behavior of NFs, operators use network testing and verification tools, which often rely on models of the deployed NFs. The effectiveness of these tools depends on the fidelity of such models. Today, models are handwritten, which can be error prone, tedious, and does not account for implementation-specific artifacts. To address this gap, our goal is to automatically infer behavioral models of stateful NFs for a given configuration. The problem is challenging because NF configurations can contain diverse rule types and the space of dynamic and stateful NF behaviors is large. In this work, we present *Alembic*, which synthesizes NF models viewed as an ensemble of finite-state machines (FSMs). *Alembic* consists of an offline stage that learns symbolic FSM representations for each NF rule type and an online stage that generates a concrete behavioral model for a given configuration using these symbolic FSMs. We demonstrate that *Alembic* is accurate, scalable, and sheds light on subtle differences across NF implementations.

1 Introduction

Modern production networks include a large number of proprietary network functions (NFs), such as firewalls (FWs), load balancers (LBs), and intrusion detection systems (IDSs) [21]. To help debug network problems, ensure correct behavior, and verify security, there are many efforts in network testing and verification [22, 35, 40, 41] as well as "on-boarding" new virtual NFs [32].

Such network management tools rely on *NF models* to create test cases, generate verification proofs, and run compatibility tests. These models are required because NF implementations are often proprietary, leaving operators with only configuration interfaces and vendor manuals. Today, NF models are handcrafted based on manual investigation [22, 40], which is tedious, time-consuming, and error-prone. Further, models do not capture subtle implementation differences across vendors [22, 30, 35]. Using low-fidelity models can affect the correctness and effectiveness of these management tools (§2).

Ideally, we want to automatically synthesize high-fidelity NF models. Synthesizing such models is challenging because: (1) NFs have large state spaces; (2) their state may be mutated by any incoming packet; and (3) in response, the NF may react with any number of diverse and possibly even nondeterministic actions. In this paper, we present *Alembic*, a system that addresses a scoped portion of this open challenge. Specifically, we focus on modeling NFs where their internal states are mutated by incoming TCP packets and their actions are restricted to dropping and forwarding packets, possibly with header modification. Our goal is to synthesize high-fidelity NF models given only the binary executable, vendor manuals, and a specific configuration with which the NF is to be deployed. We adopt this pragmatic approach as vendors may not be willing to share their source code, even with customers. Even this scoped problem presents significant challenges:

- *C1) Modeling and representing stateful NF behaviors*: The behavior of an NF often depends on the history of observed traffic, making it difficult to discover and concisely represent its internal states.
- *C2) Large configuration space*: Concrete configurations (e.g., a FW rule-set) are composed of multiple rules. Fields within a rule (e.g., source IP) can take large sets of values or ranges of values (e.g., IP prefix), making it impractical to infer models for all possible configurations.
- *C3) Large traffic space*: Given the stateful behavior, the input space potentially includes all possible *sequences* of TCP packets. Naively enumerating this large space would be prohibitively expensive.
- *C4) NF actions*: NFs such as NATs can modify packet headers, making model inference more difficult.

*Contributions by Soo-Jin Moon were made in-part during a former internship at Hewlett Packard Labs. Other contributors from former employees at Hewlett Packard Labs include Sujata Banerjee, Ying Zhang and Wenfei Wu.

To tackle these challenges, we leverage the following key insights (§3):

A) Compositional model: Rather than exhaustively modeling an NF under all possible configurations, we consider the NF's behavior as the logical composition of its behavior for individual rules in a configuration.

B) Learning symbolic model: Configurations consist of different rule types, such as a firewall drop rule, where each type is associated with a different runtime behavior of the NF. For a given type, the logical behavior of the NF is the same across different values of the rule's parameters. Hence, we can learn a *symbolic* model for each rule type rather than exhaustively infer a new model for each possible value.

C) Ensemble representation: Even with the above insights, each rule has a large search space as each rule parameter can take a range of values (e.g., a range of ports). Fortunately, we observe that NF behavior is logically independent for subsets of these ranges. For instance, assume a FW contains one rule and we know it keeps per-connection state. We can then model this rule using an *ensemble of independent models* by cloning the model learned using a single connection. However, we must then consider how to infer the specific granularity of state tracked by the NF (e.g., per-connection or per-source). We show in §5 how we can automatically infer this granularity and prove the correctness of this approach in §C.

D) Finite-state machine (FSM) learning: FSMs are a natural abstraction to represent stateful NFs [22, 35], and using them allows us to potentially leverage classical algorithms for FSM inference (e.g., L* [12]). But there are practical challenges in directly applying L* here: First, we need to create suitable mappings between logical inputs (i.e., an input alphabet) that L* uses and the real network packets/configurations that NFs take as inputs (§4). Second, header modifications by NFs make it incompatible with L*, so we need domain-specific ideas to handle such cases (§6).

Having described the high-level insights, we discuss how they specifically address the challenges: Compositional modeling (Insight A) addresses the large configuration space (C2). Both symbolic and ensemble representations (Insights B and C) address the large traffic space (C3) by learning a symbolic model for each rule type and then appropriately cloning it to create an ensemble representation (say for large IP/port ranges). Lastly, extending L* (Insight D) enables us to represent stateful NF behavior (C1 and C4).

Building on these insights, we design and implement *Alembic*.[1] In the *offline stage*, we infer symbolic FSMs for different rule types as defined by an NF's manual. To concisely represent the internal states of an NF, we extend the L* algorithm [12]. We also leverage our L*-based workflow to infer the state granularity tracked by the NF (e.g., per-connection). Since model synthesis need only be done once per NF, we can afford several tens of hours for this stage. Given a concrete configuration (i.e., a set of rules), the *online stage* uses these symbolic models to construct a concrete model within a few seconds. Specifically, the online stage maps each rule in a configuration to a corresponding symbolic FSM which, coupled with the inferred granularity, is used to create an ensemble of FSMs. The ensemble is logically composed together for each rule to construct the final concrete model for the given configuration. The resulting concrete model can then be used as an input to network testing and verification tools.

[1] Alembic is a reference to the tool used in the alchemical process of distillation or extraction, as our system extracts models from NFs.

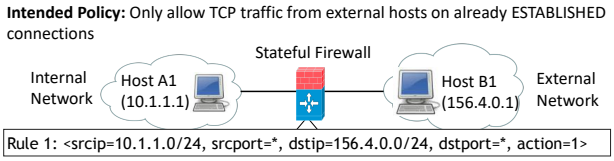


Figure 1: **Network set-up**

We evaluate *Alembic* with a combination of synthetic, open-source, and proprietary NFs: PfSense [5], Untangle [7], ProprietaryNF, Click-based NFs [31], and HAProxy [2]. We show that *Alembic* generates a concrete model for a new configuration in less than 5 seconds, excluding the offline stage. *Alembic* finds implementation-specific behaviors of NFs that would not be easily discovered otherwise, including some that depart significantly from typical high-level handwritten models (§8.4). For instance, we discover: (1) in contrast to a common view of a three-way TCP handshake, for some NFs, the SYN packet from an internal host is sufficient for an external host to send any TCP packets; and (2) the FIN-ACK packet does not cause internal NF state transitions leading to the changes in the NF's behavior. Finally, we show that using *Alembic*-generated models can improve the accuracy of network testing and verification tools (§8.5).

2 Motivation

In this section, we highlight some examples of how inaccuracies in handwritten NF models may affect the correctness of network verification and testing tools. Figure 1 shows an example network, where the operator uses a stateful FW to ensure that external hosts (e.g., B1) cannot initiate TCP traffic to internal hosts (e.g., A1). This intent translates to three concrete policies:

- **Policy 1:** To prevent unwanted traffic from entering the network, A1 must establish a connection with B1 before the FW forwards B1's TCP packets to A1.
- **Policy 2**: When A1 sends a RST or RA (RST-ACK) packet to terminate the connection, the FW should drop all subsequent packets from B1.
- **Policy 3:** To protect against an attacker sending out-of-window packets to de-synchronize the FW state [44], the FW should drop or send a RST when it receives packets with out-of-window sequence (seq) or acknowledgment (ack) numbers.

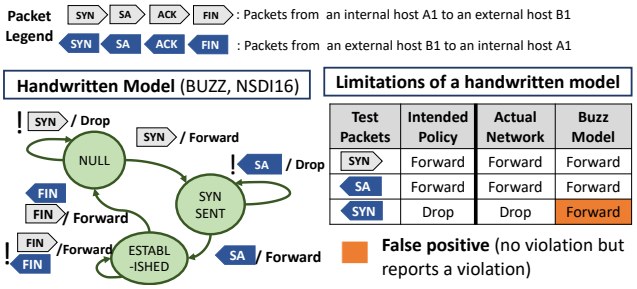


Figure 2: **A handwritten model of a stateful firewall (FW) which incorrectly reports a policy violation**

To implement these policies, the FW is configured with the rule shown in Figure 1. Since many FWs implement a default-drop policy, there is no explicit drop rule for packets originating externally. Note we do not need explicit rules for Policy 2 and 3 as they should be performed by the FW when following the TCP protocol.

To check if the network correctly implements the intended policies, operators use testing and verification tools [22, 35, 40]. These tools use *NF models* to generate test traffic [22, 41] or to verify intended properties [35]. If these models are inaccurate, the results can have any of the following error types: (1) *false positives*, where the tool reports violations when there is no violation; (2) *false negatives*, where the tool fails to discover violations; or (3) *inability to test or verify* where the tool fails completely because the models are not expressive enough. As an example, consider BUZZ [22], a recently-developed network testing tool. BUZZ uses a model-based testing approach to generate test traffic for checking if the network implements a policy, and the original paper includes several handwritten models. In the remainder of this section, we present three examples of how operators can encounter issues while using the BUZZ tool due to *discrepancies* between handwritten models and NF implementations. Our goal is not to pinpoint limitations of the BUZZ tool but to highlight shortcomings of handwritten models. We find that models from other tools lead to similar problems [35, 40].

To control for NF-specific artifacts (for now), we use two custom, Click-based [31] FWs that correctly implement the above policies.[2] Figure 2 shows the handwritten model of a stateful FW used in the BUZZ tool [22]. We use the BUZZ FW model for comparison as it implements a policy similar to our example (i.e., the FW only forwards packets belonging to a TCP connection initiated by an internal host).

Test case (policy 1): The operator uses the BUZZ tool to generate test traffic and check if TCP packets from B1 can reach A1. Figure 2 shows a sample test traffic sequence generated by BUZZ: $\text{SYN}^{\text{Internal}}_{\text{A1}\to\text{B1}}$ (i.e., TCP SYN packet from A1 to B1), $\text{SYN-ACK}^{\text{External}}_{\text{B1}\to\text{A1}}$, and finally $\text{SYN}^{\text{External}}_{\text{B1}\to\text{A1}}$. Our Click-based FW drops the last SYN from B1, which matches the policy intent as the TCP handshake did not complete. However, according to the handwritten model, $\text{SYN}^{\text{External}}_{\text{B1}\to\text{A1}}$ is marked as forwarded. Specifically, the model updates the state to ESTABLISHED on receiving a SYN-ACK (SA in Figure 2) from B1, allowing $\text{SYN}^{\text{External}}_{\text{B1}\to\text{A1}}$ to be forwarded to A1. This discrepancy between the model and the Click-based FW will be flagged as a policy violation, resulting in a *false positive*.

Test case (policy 2): The operator wants to test if a RST from A1 actually resets the connection state of the FW. However, as we see in Figure 2, the handwritten model only checks for FIN packets but not RST packets to reset the connection state. Hence, the test cases generated by the handwritten model will have discrepancies with the Click-based FW, resulting in a *false positive* (similar to policy 1).

Test case (policy 3): The operator wants to test whether the FW correctly handles packets with out-of-window seq and ack numbers. We observe that many FW vendors enable this feature by default (e.g., §8.4). Unfortunately, the handwritten model is not expressive enough to encode the notion of packets with correct and incorrect seq and ack numbers.

To make matters worse, existing tools (e.g., [22, 35, 40]) assume homogeneous models across vendor implementations for a given NF type. However, we found non-trivial differences in implementations (§8.4). Further, NF models fed to testing and verification tools need to be aware of the *impact of specific configurations*, which can easily be missed by handwritten models. For instance, the BUZZ FW model assumes a *default drop* policy from the external interface, which is consistent with many vendors. However, while running model inference using *Alembic*, we found that one specific NF (Untangle FW) *allows* packets by default [7]. To implement a default-drop policy in Untangle, we need an explicit drop-all rule, and a model for Untangle needs to be customized for this configuration.

3 Alembic System Overview

In this section, we state our goals, identify the key challenges, describe our insights to address these challenges, and provide an end-to-end overview of *Alembic*.

Preliminaries: We introduce the terminology related to NF configurations, which describe an NF's runtime behavior. A `configuration schema` contains NF rule types. Each `rule type` has various configuration fields, and the data types these fields accept (e.g., "srcip" takes an IPv4 range). Once we specify the concrete values for the fields (concrete values can be wild-card), we obtain a `concrete rule` of the rule type. A `concrete configuration` consists of multiple concrete rules. Figure 3 shows an example of a firewall (FW) and a network address translation (NAT) configuration schema and their corresponding concrete configurations. In the NAT Rule type, the outsrcip field denotes the possible output IP values used in address translation.

[2] Because BUZZ's included FW model does not encode the notion of out-of-window packets, we wrote a FW that adheres to policies 1 and 2 for a fair comparison, and a separate FW for policy 3.

ProprietaryNF FW

ConfigSchema:
Rule type 1 (Accept): ⟨srcip:IPv4 range, srcport:Port range, dstip:IPv4 range, dstport:Port range, action:1 ⟩
Rule type 2 (Deny): ⟨ srcip:IPv4 range, srcport:Port range, dstip:IPv4 range, dstport:Port range, action:0 ⟩
ConcreteConfig:
Rule 1: ⟨ srcip:10.1.1.1,srcport:*,dstip:156.4.0.1,dstport:*, action:1 ⟩
Rule 2: ⟨ srcip:10.8.0.0/16,srcport:*,dstip:151.0.0.0/8,dstport:*,action:0⟩

PfSense outbound NAT

ConfigSchema:
Rule type 1: ⟨srcip: IPv4 range, srcport: Port range, dstip: IPv4 range, dstport: Port range, outsrcip: IPv4 range, outsrcport: Port range⟩
ConcreteConfig:
Rule 1: ⟨srcip:10.1.0.0/16,srcport:*,dstip:156.4.0.0/16,dstport:*, outsrcip:126.2.0.0/16,outsrcport=* ⟩
Rule 2: ⟨srcip:10.0.0.0/8,srcport:*,dstip:162.4.0.0/16,dstport:*, outsrcip:192.1.0.0/16,outsrcport=* ⟩

Figure 3: **Example of a simplified ConfigSchema and ConcreteConfig for a FW and a NAT**

3.1 Problem formulation

Given an NF with a concrete configuration, *Alembic*'s goal is to automatically synthesize a *high-fidelity* behavioral model of the NF. Since NF implementations do not change often, we can afford several tens of hours of offline profiling per NF. However, since concrete configurations (e.g., a FW rule-set) can change often, we need to generate a new model given a new configuration quickly, within a few seconds.

Alembic takes five inputs: (1) the NF executable binary, (2) the *configuration schema* (ConfigSchema), (3) the high-level rule *processing semantics* of parsing the configuration (e.g., first match), (4) a list of network interfaces, and (5) the set of input packet types (e.g., TCP SYN or ACK) the model needs to cover. For (1), we assume no visibility into the internal implementation or source code and only have access to its manual describing configuration. For (2), the ConfigSchema is typically already available from vendor documentation.[3] The ConfigSchema in Figure 3 assumes we are explicitly given a set of rule types (e.g., accept or deny), where each rule type is associated with a different runtime behavior. In practice, the vendor documentation may only specify a set of fields and their types. For instance, a FW ConfigSchema provides one rule type with an action field that takes a binary value, in which each value leads to a rule type with different runtime behaviors. We show how we generate a set of all rule types in such a case (§7). For (3), we assume the rule processing semantics are available from the vendor documentation. Our design can handle any NF that applies a single rule per packet. Our implementation currently supports first-match semantics but can be easily extended to handle others (e.g., last-match). For (4), we need to know a list of interfaces that the NF is configured with. In this work, we assume that we are given two interfaces (e.g., internal and external-facing interfaces).

[3] *Alembic* requires a one-time, manual effort to translate this documentation into a format compatible with our current workflow.

Lastly, given packet types (5), *Alembic* will automatically configure each packet type with appropriate field values.

Here, we focus on modeling TCP-relevant behavior for NFs that forward, drop, or modify headers (e.g., FWs, NATs, and LBs). We provide default packet types for TCP, but *Alembic* can be extended with additional packet types. We scope the types of NFs and their actions that *Alembic* can handle in §3.3 and discuss how to extend *Alembic* to handle more complex NFs in §10.

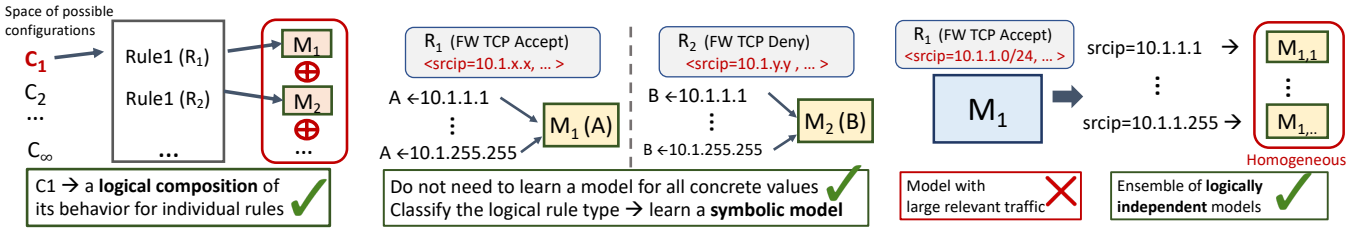
3.2 Key ideas

To highlight our main insights to address challenges *C1* through *C4* from §1, suppose we want to model an NF with a concrete configuration C_1 composed of N concrete rules $\{R_1 \cdots R_N\}$. Figure 4 illustrates our ideas to make this modeling problem tractable.

A) Compositional model (Fig. 4a): The concrete configuration C_1 can be logically *decomposed* into individual rules. As seen in Figure 4a, suppose we have models M_1 for R_1 and M_2 for R_2. Then, we can create a *compositional model* for the NF given the processing semantics defined by the ConfigSchema (e.g., first-match). If the packet matches $Rule_1$, then apply $Model_1$, else if it matches $Rule_2$, then apply $Model_2$. Otherwise, apply $Model_{default}$.

B) Symbolic model (Fig. 4b): To start, we make two simplifying assumptions, which we relax below: (1) the IP and port fields in a concrete rule take a single value from a range (e.g., 10.1.1.1 for srcip); and (2) the NF keeps per-connection state. Suppose the srcip field in R_1 (Figure 4b) takes a single IP from 10.1.0.0/16. It is infeasible to exhaustively infer the model for all possible values. Fortunately, we observe that the logical behavior of the NF for a particular rule type (e.g., FW accept rule) is *homogeneous* across different values for the IPs and ports in this range. Thus, we can efficiently generate a model by representing each IP and port field in a rule with a symbolic value. Hence, for each logical rule type (e.g., FW accept rule), we can learn a symbolic model (e.g., $M_1(A)$).

C) Ensemble representation (Fig. 4c): We relax the assumption that IPs and ports take single values and discuss how we handle ranges within a rule (i.e., R_1 in Figure 4b takes a /16 prefix for a srcip). We observe that NF behavior is *logically independent* for subsets of this large traffic space. Consider a stateful firewall that keeps per-connection state. Rather than viewing M_1 as a monolithic model that captures the behavior of all relevant connections, we can view the model as a *collection of independent models*, one per connection (i.e., $M_{1,1}$ for connection 1, $M_{1,2}$ for connection 2, etc.). Combining this idea with B above, we learn a *symbolic model* for each rule type and *logically clone the model* to represent IP and port ranges (henceforth, an ensemble of models). However, to leverage this idea, we need to infer the granularity at which an NF keeps independent states (e.g., per-connection or per-source). We show in §5 how to automatically infer this.



(a) **Compositional model for a config** (b) **Symbolic model (e.g., $M_1(A)$ for TCP accept)** (c) **Ensemble of independent models per rule**

Figure 4: ***Alembic* Key Insights**

Algorithm 1 NF operational model for processing incoming packets

```
1:  function NF(locatedPkt p, Config c, ProcessingSemantic ps,
    Map[rule, Map[key, state]] stateMap)
2:      poutList = []
3:      rule = FindRuleToApply(p, c, ps)
4:      if rule is None then
5:          rule = GetDefaultRule( )
6:      keyType = GetKeyType(rule)
7:      key = ExtractHeader(keyType, p)
8:      FSM = GetModel(key, rule)
9:      curState = GetState(stateMap, rule, key)
10:     poutList, nextState = Transition(FSM, p, curState)
11:     UpdateState(stateMap, rule, key, nextState)
12:     return poutList
```

D) FSM inference: The remaining question is how to represent and infer a symbolic model. Following prior work in stateful network analysis, we adopt the FSM as a natural abstraction [22, 35]. To this end, we develop a workflow that leverages L* for FSM inference [12]. At a high-level, given a set of relevant inputs, L* adaptively constructs sequences, probes the blackbox, and infers the FSM. However, directly applying L* for an NF entails significant challenges: First, L* requires the set of inputs a priori. Hence, we need to generate inputs from a large input space, and create suitable mappings between inputs that L* takes and real packets for the NF. Second, L* is not suitable for learning a FSM for a header-modifying NF because it assumes: (1) we know the input alphabet a priori, and (2) the underlying system is deterministic. As an example violation of (2), a NAT may nondeterministically choose the outgoing ports. We leverage a domain-specific idea to extend L* for such cases (§6).

3.3 Operational model and limitations

Having described our key insights, we scope the types of NFs for which *Alembic* is applicable. We use an abstract NF (Algo. 1) to describe how incoming packets are processed (a more detailed description can be found in §B). Our goal is to handle NFs with logic stated in Algo. 1.

NF operational model: We start by describing the inputs and outputs of the abstract NF. The NF receives or transmits *a*

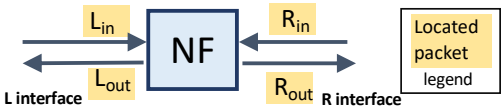


Figure 5: **An NF with located packets**

located packet [34] (i.e., a packet associated with an interface). Figure 5 shows a setup for an NF with 4 located packets. The NF is configured with two interfaces, L (e.g., internal) and R (e.g., external). As an example, L_{in} is a packet entering the NF via L, and L_{out} is an outgoing packet from the NF via L.

The abstract NF is configured with a *concrete configuration*, composed of a set of *rules*. Each rule maintains a mapping between keys and concrete FSMs. For instance, if the NF uses a per-connection key, then it will keep a concrete FSM for each unique 5-tuple. The concrete FSMs describe the appropriate action (i.e., L_{out} or R_{out}) for an incoming located packet (i.e., L_{in} or R_{in}). As shown in Algo. 1, when a located packet arrives, the NF searches the configuration for the correct rule to apply based on the processing semantics. If no rule is found, the NF uses the default (i.e., empty) rule. Then, it uses the relevant packet headers determined by the rule's key to find the concrete FSM and current state associated with that key. Finally, the NF processes the packet according to the FSM and updates the current state (Lines 10 and 11). *Alembic* aims to synthesize models for NFs following Algo. 1.

Assumptions on configurations: We make the following assumptions about NF configurations:

- Rules in a concrete configuration are independent. For instance, we do not consider NFs that share the same state across different rules. At most one rule in a configuration can be applied to an incoming packet.
- Within a concrete rule, the states across different *keys* (i.e., state granularity tracked by an NF) are *independent*. For a per-connection FW with a rule that takes IP and port ranges, states across connections are independent.
- When IPs and ports in a concrete rule take ranges (e.g., ports=*), NFs treat each value in the range *homogeneously* such that we can pick a representative sample and learn a symbolic model (i.e., the symbolic model obtained using port 80 or port 5000 for an outsrcip is identical).

Assumptions on NF actions: We now scope the NF actions that *Alembic* can handle:

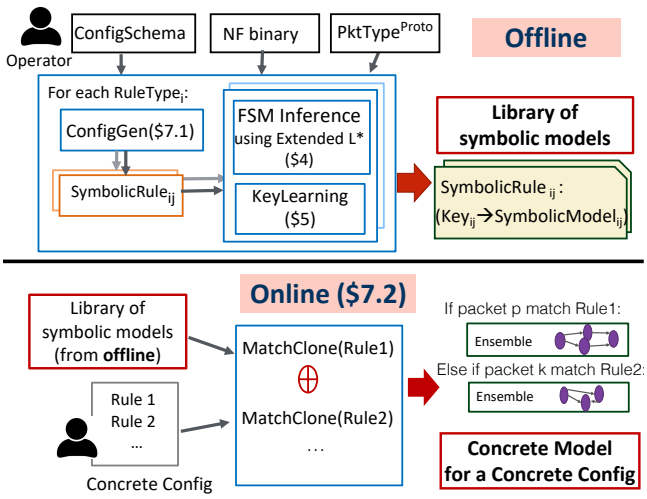


Figure 6: ***Alembic* Workflow**

- For simplicity, we only consider single-function NFs, excluding cases such as combined NFs processing FW rules and then NAT rules.
- To make learning tractable, we only look at IP and port modifications. Our implementation does not consider seq/ack numbers, ToS, or other fields (§4.3). We only handle header modifications for connection-oriented NFs (§6).[4] We tackle header modification for an NF that initially modifies IP/port of a packet, p1, entering from a particular interface *before* modifying a packet, p2 (that belongs to the same connection as p1) entering from the other interface. Lastly, we cannot infer context-sensitive relations such as how the modified IP or port (e.g., NAT ports) is chosen.
- We do not explicitly model temporal effects, such as connection timeouts. When we inject input packets into the NF, we collect outputs for Δ_{wait} (e.g., 100 ms) before injecting the next input packet. *Alembic* cannot handle cases where output packets are results of prior input packets (e.g., retries after 1 second).
- We support five types of state granularity: per-connection, per-source (e.g., a scan detector which counts a number of SYN packets), per-destination (e.g., DDoS detector), cross-connection, and stateless.

3.4 Alembic workflow

Having described our key insights and scope, we now present our workflow (Figure 6) consisting of two stages:

Offline stage: From the ConfigSchema, we generate a set of rule types (§7). Given each rule type, the **ConfigGen** module generates a SymbolicRule, R^{symb}, and a corresponding ConcreteRule. For instance, given a FW ConfigSchema, it generates two SymbolicRules and ConcreteRules (e.g., FW accept and deny rule as shown in Figure 7).

[4]Most header modifying NFs we are aware of are connection-oriented.

R_1^{symb} : ⟨src:A,srcport:Ap1,dst:B,dstport:Bp1, action:1⟩ **FW TCP Accept**
R_1^{conc}: ⟨src:10.1.1.1,srcport:2000,dst:156.4.0.1,dstport:5000,action:1⟩

R_2^{symb}: ⟨src:A,srcport:Ap1,dst:B,dstport:Bp1, action:0⟩ **FW TCP Deny**
R_2^{conc}: ⟨src:10.1.1.1,srcport:2000,dst:156.4.0.1,dstport:5000,action:0⟩

Figure 7: **SymbolicRules and ConcreteRules for a FW**

For each SymbolicRule, we use the **FSMInference** module, which leverages L*-based workflow to infer a symbolic model where IPs and ports are symbolic (§4) and handles header modifications (§6). This module uses our version of L* (i.e., **Extended L***). We also design the **KeyLearning** module, which leverages the FSMInference module and infers the state granularity (i.e., key type) tracked by the NF (e.g., per-connection). Using the key type, we can identify the **key**, a set of header field values that identifies logically independent states (e.g., a 5-tuple for per-connection NF). The offline stage produces a set of symbolic models, mapping each SymbolicRule to a symbolic model and its key type.

Online stage: Given a new configuration, each rule is matched to a corresponding SymbolicRule, mapped to a key type and a symbolic model. Based on the key type, we logically clone the symbolic model to represent concrete IP and port ranges (collectively, an ensemble of FSMs). Given the processing semantics, we logically compose each ensemble to create the final model for this configuration. Network management tools can then use the resulting model.

Roadmap: In the interest of clarity, §4 describes the FSMInference module of *Alembic* for a given SymbolicRule with the following simplifying assumptions: NFs keep per-connection state and do not modify headers. In subsequent sections, we relax these assumptions and show how we infer the state granularity (§5) and handle header-modifying NFs (§6). §7 discusses how we generate a set of rule types and the corresponding SymbolicRule and the *Alembic* online stage.

4 Extended L* for FSM Inference

We now present the FSMInference module, which leverages the Extended L* for inferring a symbolic model given a SymbolicRule, R^{symb} (e.g., in Figure 7). Recall that we are also given a corresponding ConcreteRule, R^{conc}, to configure the NF. For clarity, we start with two simplifying assumptions: (1) NFs keep per-connection state, and (2) NFs do not modify packet headers. We relax these assumptions in §5 and §6.

4.1 Background on L* algorithm

Before discussing the challenges of directly applying L*, we provide a high-level description of the L* algorithm [12], which infers a FSM for a given blackbox. Given the input alphabet, Σ (e.g., $\{a,b\}$ where a, b are input symbols), L* generates sequences (e.g., *a*, *aa*, *aba*), and probes the blackbox, resetting the box between sequences. For each input

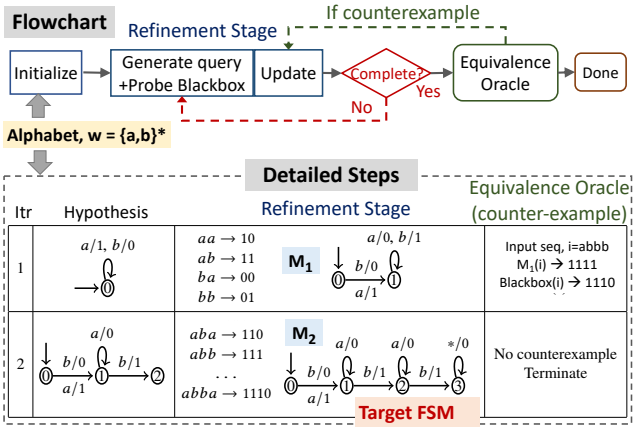


Figure 8: **L* overview and example**

sequence, L* builds a *hypothesis* FSM consistent with the input-output pairs seen so far. Specifically, it builds a Mealy machine whose outputs are a function of its current state and inputs. As shown in Figure 8, L* iteratively refines the hypothesis FSM until it is complete (i.e., the set of probing sequences cover the state space of this hypothesis). After the hypothesis converges, L* queries an Equivalence Oracle (EO), which checks if the inferred FSM is identical to the blackbox and provides a counterexample if they are not. If the EO reports that the hypothesis is identical to the blackbox, the algorithm terminates. Otherwise, L* uses the counterexample to further refine the hypothesis. The process repeats until the EO reports no counterexamples. L*'s runtime complexity is polynomial in the number of states and transitions of a minimal FSM representing the target FSM as well as the length of the longest counterexample used to refine the hypothesis [12].

Example: Figure 8 illustrates an example of the steps in L* for the target FSM shown with $\Sigma = \{a, b\}$. Initially, L* starts with the inputs, a and b, and a single-state FSM. It generates four sequences to refine the model and converges to M_1 as shown. It then queries the EO and finds a counterexample where Blackbox($abbb$)=1110 but $M_1(abbb)$=1111, which is used to update the model. To explore the state space of the new hypothesis, L* generates longer sequences. After this second iteration, the EO finds no counterexamples (as M_2 is identical to the blackbox), and the algorithm terminates.

4.2 Challenges in using L* for NFs

While L* is a natural starting point, there are practical challenges in applying it directly to NFs. We will describe these challenges using Figure 9 and discuss our solutions.

1) Generating input alphabet (§4.3): L* assumes the input alphabet (Σ) is known. As discussed in §3, we can set Σ for *Alembic* to be a set of located symbolic packets, which are packets with symbolic IPs and ports associated to interfaces. From now on, when we say packets, we refer to located packets. The main disconnect here is that the NF (i.e., the blackbox

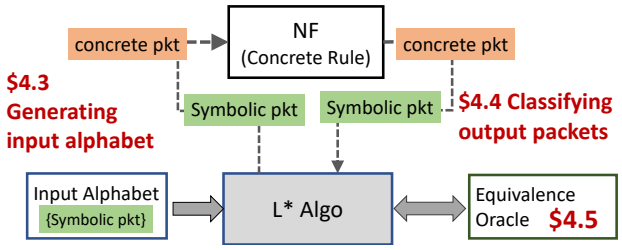


Figure 9: **Key challenges in adopting the L* workflow for NF model inference**

in the L* workflow) takes in concrete packets and not symbolic packets. Thus, we need to map a symbolic packet to a concrete packet. Two challenges exist here: First, the possible header space for concrete packets is large (i.e., all IPs and ports), and second, the concrete packets need to exercise the internal states of the NF (e.g., trigger the NF behavior).

2) Classifying output packets (§4.4): Next, for each symbolic packet suggested by L*, we need to map it to an NF action. The practical challenge is that NFs may require an unpredictable delay. If we assume a processing delay that is too short and classify the action as a drop, we might learn a spurious model. While a delay that is too long will lead to our inferences taking a long time. Thus, we need a robust way to map an input to the observed output.

3) Building an equivalence oracle (§4.5): L* assumes access to an EO (Figure 9). In cases where we do not have access to the ground truth, we can only approximate the oracle via input-output observations. There are two practical issues. First, existing approaches (e.g., [17, 25]) to building an EO generate a large number of equivalence queries, creating a scalability bottleneck. Second, different approaches for building an EO may affect the soundness of *Alembic* (§4.5).

4.3 Generating input alphabet

We now describe how we generate a set of *located symbolic packets* for the input alphabet and how we map each located symbolic packet to a concrete packet. As discussed in §3, we are given the representative packet types of interest $\text{PktType}^{\text{Proto}}$ (e.g., TCP handshake) as an input.

To illustrate these challenges, consider two straw-man solutions that generate packets for: (1) every possible combinations of header fields, and (2) randomly generated header fields. (1) is prohibitively expensive, and (2) may not exercise the relevant stateful behaviors. Our idea is to use the symbolic and concrete rules to identify relevant header fields and their values. Specifically, we observe that the header fields and their values (e.g., IP-port) in R^{conc} will trigger relevant NF behaviors. Thus, we generate all combinations of these relevant IP-port pairs using their concrete values from R^{conc}. Using a pair of R_1^{symb} and R_1^{conc} as an example (Figure 7), we identify A=10.1.1.1 as a possible candidate for both source and des-

tination IPs across *all interfaces* (i.e., A can be a source or destination IP on packets entering from internal or external interfaces). We consider all interfaces, as a packet entering different interfaces can be treated differently.

We also consider the scenario where the packet does not match any rules. One approach is to pick concrete header values that do not appear in the concrete rule and generate a corresponding symbolic packet (e.g., not A=12.1.0.1). However, this would double the size of Σ. Instead, we leverage our insight regarding the compositional behavior of NFs and view this as composing the action with the *default* behavior of the NF when no concrete rule is installed. We separately infer a model, $M_{default}$, with an empty configuration (e.g., a FW without any rules).[5]

Example: From R^{symb}, we mark A:Ap1 and B:Bp1 as possible IP:port pairs, where A:Ap1 and B:Bp1 refer to srcip:srcport and dstip:dstport pairs from R^{symb}. Then, we generate all possible combinations across source and destination IP/ports and network interfaces: (1) $\text{TCP}^{\text{Internal}}_{\text{A:Ap1}\rightarrow\text{B:Bp1}}$ (corresponding to a TCP packet with srcip:port=A1:Ap1 and dstip:port=B1:Bp1 on the internal interface), (2) $\text{TCP}^{\text{External}}_{\text{A:Ap1}\rightarrow\text{B:Bp1}}$, (3) $\text{TCP}^{\text{Internal}}_{\text{B:Bp1}\rightarrow\text{A:Ap1}}$, ..., etc. Suppose the packet types of interest are: $\{\text{SYN}, \text{SYN-ACK}, \text{ACK}\}$. Then, for (1), we obtain $\text{SYN}^{\text{Internal}}_{\text{A:Ap1}\rightarrow\text{B:Bp1}}$, $\text{SYN-ACK}^{\text{Internal}}_{\text{A:Ap1}\rightarrow\text{B:Bp1}}$, $\cdots$. We follow the similar procedure for (2) and (3). Essentially, $\text{SYN}^{\text{Internal}}_{\text{A:Ap1}\rightarrow\text{B:Bp1}}$ is a symbolic packet which maps to a concrete SYN packet with A=10.1.1.1 and Ap1=2000 that is injected from the internal interface. *Alembic* internally tracks the symbolic-to-concrete map (i.e., A=10.1.1.1) to connect the symbolic packet used by L* to the concrete packets into the NF. Finally, we (optionally) prune out packets that are infeasible given the known reachability properties of the network. For instance, it is infeasible for a packet with srcip=10.1.1.1 to enter from the external interface.

4.4 Classifying output packets

To classify the output from the NF, we monitor for output packets at all interfaces of the NF and map them to their symbolic representations. For instance, after detecting a SYN on the external interface with source IP:port, 10.1.1.1:2000, and destination IP:port, 156.4.0.1:5000, we assign the output symbols as $\text{SYN}^{\text{External}}_{\text{A:Ap1}\rightarrow\text{B:Bp1}}$. Specifically, *Alembic* monitors all interfaces for Δ_{wait} and reports the set of observed packets (e.g., L_{out} and R_{in}). Δ_{wait} is critical for classifying dropped packets and we cannot have an arbitrarily assigned values. Unfortunately, an NF sometimes introduces unexpectedly long delays in packets ($\geq$ 200ms). For instance, Untangle performs connection setup steps with variable latency upon receiving SYN packets, and ProprietaryNF experiences periodic spikes in CPU usage leading to delayed packets. Such delays can result in misclassifying a packet as a drop and affect the learning process. For these NFs, Δ_{wait} is determined by injecting the TCP packets and measuring the maximum observed delay. Further, we extended L* with an option to probe the same sequence multiple times and pick the action that occurs in the majority of test sequences.

4.5 Building an equivalence oracle

Building an efficient oracle is difficult with just black-box access [17,25]. Any EO will be incomplete as it cannot generate all sequences. Our goal is to achieve soundness with respect to the generated Σ without sacrificing scalability.

We tested three standard approaches for generating EOs that LearnLib [38], an open-source tool for FSM learning, supports: (1) *Complete Oracle* (CO), which exhaustively searches sequences to a specified length; (2) *Random Oracle* (RO), which randomly generates sequences; and (3) *Partial W-method* (*Wp-method*) [25], which takes d as an input parameter which is an upper bound on the number of additional states from its current estimate at each iteration.[6] We discarded the CO as it simply performs an exhaustive search and the RO as it is not systematic in exploring the state space. Instead, we use the Wp-method, a variant of the W-method [17] that uses fewer test sequences without sacrificing W-method's coverage guarantees. Briefly, the W-method uses a characterization set, the W-set, which is a set of sequences that distinguish every pair of states in the hypothesis FSM. The W-method searches for new states that are within d additional inputs of the current hypothesis and uses the W-set to confirm the new states. In theory, one can set d to be large but increases the runtime by a factor of $|\Sigma|^d$. For this reason, we set $d = 1$ in *Alembic*. *Alembic* can only discover additional NF states that are discoverable by the Wp-method with $d = 1$; i.e., *Alembic* with Wp-method ($d = 1$) is sound. Even with $d = 1$, *Alembic* synthesizes models that are more expressive than many handwritten models and discovers implementation-specific differences (§8).

Distributed learning: Both L* and Wp-method for $d = 1$ are polynomial in runtime. However, the Wp-method is the bottleneck as the number of sequences generated by Wp-method is approximately $|\Sigma|$ factor higher than that of the L*. Fortunately, the equivalence queries can be parallelized. In our system implementation (§8), we run equivalence queries in parallel across multiple workers until we find a counterexample. Using this technique, we can significantly reduce the time for learning a complex behavioral models (§8.3).

5 KeyLearning: Learning State Granularity

Thus far, we assumed that the NF maintains per-connection state. We now relax this assumption and show how we tackle

[5] We acknowledge an assumption that rule matching is correctly implemented by the NF. If the NF has a rule for src=A and dst=B but a buggy implementation that matches A' and B', we will not uncover this behavior.

[6] In practice, the number of states can grow by $> d$ at each iteration.

NFs that maintains other key types (e.g., per-source). Specifically, we implement a KeyLearning module. Given a SymbolicRule, the module outputs the **key type**, a set of header fields that identify a relevant model in an ensemble representation. Note that here we still assume that the NF does not modify packet headers, which we will relax next in §6.

High-level intuition: Consider a FW configured with a rule that keeps per-connection state. A packet from one connection only affects its own FSM and is unaffected by packets that belong to other connections. Now, consider an NF which keeps per-source state, and packets, p1 and p2, with the same srcip, but with different dstip. The arrival of p1 affects not only the state for processing p1, but also the state associated with p2 because they share the same srcip. The KeyLearning algorithm builds on the above intuition; if two connections are independent with respect to an NF's processing logic, then the packet corresponding to one connection only affects the state of its FSM. Thus, to infer the key type, we construct test cases using *multiple connections* to validate the independence assumptions across these connections. We show how we can validate independence by inspecting two connections using carefully constructed source and destination values.

The KeyLearning algorithm is composed of test cases to distinguish between different key types. As a concrete example of a test case, suppose we have a SymbolicRule, which takes ⟨srcip=A, dstip=B⟩ where A and B are ranges of IPs (e.g., A=10.1.0.0/16 and B=156.4.0.0/16). First, we infer two models with two separate ConcreteRules, where we configure each IP using a concrete singleton (e.g., R_1^{conc}, with ⟨srcip=10.1.1.1, dstip=156.4.0.1⟩ to learn $Model_1$, and R_2^{conc} with ⟨srcip=10.1.1.1, dstip=156.4.0.2⟩ to learn $Model_2$). Note that these two have the *same srcip*. We leverage the FSMInference module in §4. We first generate Σ_1 for R_1^{conc} and use the FSMInference in §4 to obtain $Model_1$, and then repeat for $Model_2$. Assuming these models are independent, we run a logical *FSM composition* operation to construct $Model_{composite}$ (Def.7 in §C). This is what the hypothetical model will be if these two connections are *independent*. As a second step, we now learn a joint model $Model_{joint}$, where we combine input alphabets from both connections. Specifically, we configure a *ConcreteRule*, where the dstip takes a range of IPs (e.g., 156.4.0.1-156.4.0.2).

For example, consider a scan detector, that keeps per-source state. As the above two connections have the same srcip, $Model_{joint}$ will reflect that the packets affect each other's state (i.e., $Model_{joint}$ is not equivalent to $Model_{composite}$, which assumes independence across two connections). But, for a per-connection model, the two connections are independent (i.e., $Model_{joint}$ would be equivalent to $Model_{composite}$). Thus, we now have a simple logical test to distinguish between per-connection and per-source.

Inference Algorithm: Our inference algorithm generalizes the basic test described above. By crafting different ConcreteRules (i.e., changing the overlap on srcip or dstip) and running the equivalence tests between $Model_{composite}$ and $Model_{joint}$ for each case, we create a decision tree to identify the *key type* maintained by the NF, which are: (1) per-connection, (2) per-source (e.g., a scan detector), (3) per-destination, (4) cross-connection, or (5) stateless. [7]

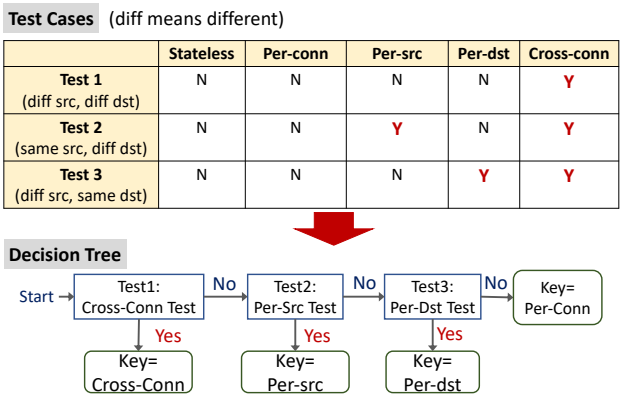


Figure 10: **KeyLearning Decision Tree**

Figure 10 shows the result of test cases for these key types. For instance, Test 1 configures two connections to have different sources and destinations, to check whether the NF keeps cross-connection state. Test 2 configures two connections to have the same sources, but with different destinations. If Test 2 outputs that two connections affect the states relevant for each other, then the NF is maintaining either a cross-connection or per-source state. The decision tree (Figure 10) uniquely distinguishes the key and the correctness naturally follows from our carefully constructed test cases. We formally prove the correctness of this approach in §C.

6 Handling NF Header Modifications

Now, we extend our FSMInference in §4 to handle header modifications, such as a NAT rewriting a private IP-port pair to a public IP-port pair. We currently only handle NFs that maintain per-connection state while modifying IPs and ports. We consider two cases of possible header modifications: (1) *static* (e.g., a source NAT modifies a private port to a static public port), and (2) *dynamic* (e.g., a source NAT or LB randomly generates port mappings across resets). We first describe how we handle each case individually, then present our combined workflow to handle both cases. Our workflow does not require knowing a priori that an NF modifies header fields, which field it modifies, or how it modifies packet headers (i.e., static or dynamic).

Static header modifications: Consider a source NAT that deterministically maps a source IP-port pair (e.g., A:Ap1) to a public source IP-port pair (e.g., X:Xp1). To discover the NAT's behavior that rewrites the public IP-port back to the private IP-port, we need to generate a symbolic packet using

[7] The key for a stateless NF is a 5-tuple. We can view a stateless NF as an FSM with a single state, which is identical to each 5-tuple keeping one state.

the public (modified) IP-port (i.e., X:Xp1). However, we may not know the concrete value of X:Xp1 a priori. Hence, we cannot generate a complete set of $|\Sigma|$. Our idea is to first run the inference module (§4) and check whether a symbolic model has additional symbolic IPs and ports. If so, we append the new IP-port pairs to the Σ and re-run the inference. We repeat this step until the output FSM contains no new IP-port pairs. Given that the static modification maps an IP-port to the same IP-port pair, this approach converges.

Dynamic header modification: The above approach of updating the input alphabet will not converge for NFs that dynamically modify packet headers, however. Consider a NAT that *randomly* picks one of the available ports for the same 5-tuple (e.g., a private IP-port (e.g., A:Ap1 first maps to X:Xp1 but then to X:Xp2 after L* resets the NF). Since L* assumes a deterministic FSM, it will crash as a result of this nondeterminism. Our idea is simple. If L* crashes, then we identify the IP-port pair that caused the nondeterministic behavior. Next, we *mask* this nondeterministic behavior of the NF from L* by explicitly mapping such IP-port pairs to consistent symbolic values (e.g., *Alembic* maps $\text{SYN}^{\text{Internal}}_{\text{A}\to\text{B}}$ to $\text{SYN}^{\text{Internal}}_{\text{X}\to\text{B}}$ regardless of the concrete value of the rewritten source IP). Since the concrete value of X will change across resets, the extended L* uses the most-recently observed concrete value of X when playing sequences.

Combining both cases, we first run the FSMInference module (§4). If L* completes but discovers new symbols (i.e., static modification), then we re-run the workflow with new symbols. However, if L* crashes due to a nondetermistic FSM (i.e., dynamic modification), we mask the non-deterministic behavior as discussed. After the required modifications are applied, the L* is repeated until it converges. As we only handle modification for per-connection NF, we assume the key is per-connection for an NF that modifies packet headers.

7 Handling an Arbitrary Config

We now discuss how we generate a set of SymbolicRules (§7.1) and then how the *online* stage constructs a concrete model given a concrete configuration (§7.2).

7.1 Generating SymbolicRules

The ConfigGen module generates a set of SymbolicRules. As discussed in §3.1, the vendor documentation may not clearly give a set of rule types where each type is associated with a different runtime behavior (e.g., FW accept vs. deny). Suppose the FW ConfigSchema specifies a rule types as ⟨srcip, srcport, dstip, dstport, action⟩ where "action" takes a binary value. To obtain a set of logical rule types, we use a set of conservative heuristics. Typically, we observe that fields which take a large set of values (e.g., IPs and ports) demonstrate similar behaviors across values within the set. For fields that only take a small set of values (e.g., action), each value typically carries a distinct runtime behavior. Based on this observation, the ConfigGen module first assigns a new symbol (i.e., A for srcip) to each field that takes a large set of values. Then for each combination of other small fields (e.g., action), this module generates a SymbolicRule (for each rule type). We also generate a corresponding *ConcreteRules* by sampling a value for each field. For the example above, ConfigGen generates two rule types, accept and deny.

7.2 Alembic online

We now describe *Alembic*'s *online* stage, which constructs a concrete model for a given a configuration. The concrete model then uses our operational model (Algo. 1) to model how an NF processes incoming packets.

Constructing a concrete model: For each concrete rule, R, in a concrete configuration, we first fetch the corresponding by SymbolicRule by substituting fields that were made symbolic with concrete values from the rule, R (e.g., ⟨srcip=10.1.0.1 ⋯ action=1⟩ matches a SymbolicRule, ⟨srcip=A ⋯ action =1⟩). Then, we fetch the corresponding symbolic FSM and the key type, and use the key type (e.g., srcip-port for per-source NF) to appropriately clone the symbolic model to create an ensemble representation. There is one additional step when the key type is not per-connection; we must substitute any ranges based upon the key type. For example, for a per-source NF, dstip-port in a concrete model refers to a range of concrete values specified in R for dstip and dstport. The output is an ensemble of concrete models for each rule in a configuration.

Processing incoming packets: Upon receiving a packet, the NF fetches the corresponding rule in a configuration using the processing semantics (e.g., first-match). The NF then uses the *key* to access the relevant concrete FSM in an ensemble of FSMs and the current state associated with the packet (Line 9 in Algo. 1). Finally, the NF applies the appropriate action and updates the current state associated with that packet. We present a more detailed description of how we instantiate an ensemble of FSMs in §B.

8 Implementation & Evaluation

System Implementation: We implemented *Alembic* using Java for the extended L*, C for monitoring NF actions, and Python for the rest. We create packet templates using Scapy [6]. Then, *Alembic* feeds the output of prior modules into the Extended L* built atop LearnLib [38]. We re-architected the Learnlib framework to enable distributed learning where queries are distributed to workers via JSON-RPC [4].[8] Our L* implementation tracks the symbol-concrete mapping of IPs and ports to translate between symbolic and concrete packets. The symbolic FSM output is stored in DOT format, which is then consumed by the online stage.

[8] Due to some unhandled edge cases, our current implementation requires using only one worker for NFs with dynamic header modifications.

Table 1: **Coverage of models over input packet types**

	FW			staticNAT		randNAT	LB	
PktType	pf	ut	pNF	pf	pNF	pf	pf	hp
correct-seq	●	○	●	●	●	◐	◐	○
combined-seq	●		●					

pf: PfSense, ut: Untangle, pNF: ProprietaryNF, hp: HAProxy
◐: TCP-handshake pkts, {SYN_C,SYN-ACK_C,ACK_C}, for both interfaces
○: ◐ set excluding SYN_C from the *external* interface

Table 2: **Results of stress testing**

NF (pkt type)	accuracy	NF (pkt type)	accuracy
PfSense FW (C)	98.8-100%	ProprietaryNF FW (C)	99.9-100%
PfSense FW (CI)	94.8-100%	ProprietaryNF FW (CI)	98-100%
PfSense NAT (C)	99.1-100%	PfSense randNAT (C)	98.2-100%
PfSense LB (C)	96.4-97.4%	ProprietaryNF NAT (C)	98.8-100%

C : correct-seq CI : combined-seq

L* assumes that we have the ability to reliably reset the NF between every sequences. For *Alembic*, we need to reset the connection states. For some NFs, this can be performed using a single command (e.g., `pfctl -k` in PfSense). However, other NFs required that the VM be rebooted (e.g., Untangle). In such cases, we take a snapshot of the initial state of the VM and restore the state to emulate a reset. This does cost up to tens of seconds but is a practical alternative to rebooting.

Experimental Setup: We used *Alembic* to model a variety of synthetic, open-source, and proprietary NFs. First, we created synthetic NFs using Click [31] to validate the correctness of *Alembic*. Each Click NF takes an FSM as input and processes packets accordingly, so we know NF's ground-truth FSM. To validate against real NFs, we generated models of PfSense [5] (FW, static NAT, NAT that randomizes the port mappings, and LB), ProprietaryNF (FW, static NAT), Untangle [7] (FW), HAProxy [2] (LB). We now use NAT to refer to a static NAT and a randNAT to refer to a NAT that randomizes the IP-port mappings. Our experiments were performed using CloudLab [1]. We ran PfSense, Untangle, ProprietaryNF, HAProxy, and Click in VMs running on VirtualBox [8]. Recall that Δ_{wait} needs to be customized for each NF. We used Δ_{wait} of 100 for PfSense and Click-based NFs, 250 ms for ProprietaryNF, 200 ms for Untangle, and 300 ms for HAProxy. For NFs that incur unexpected delays (e.g., HAProxy, ProprietaryNF, Untangle), we took a majority vote of 3.

Packet types: We use two TCP packet types. First, the `correct-seq` set consists of standard TCP packets, {SYN_C, SYN-ACK_C, ACK_C, RST-ACK_C, FIN-ACK_C}, where the handling of seq and ack are under-the-hood. Instead of introducing seq and ack numbers in Σ, we introduce additional logic in the Extended L* to track seq and ack of the transmitted packets and rewrite them during the inference to adhere to the correct semantics (i.e., update the ack of SYN-ACK_C after we observed an output of SYN_C).[9] Second, we introduce `combined-seq` set to model the interaction of TCP packets in the presence of out-of-window packets. We extend the correct-seq set with packets with randomly-chosen, incorrect seq and ack values, {SYN-ACK_I, ACK_I, RST-ACK_I, FIN-ACK_I}.

8.1 Validation using synthetic NFs

A) Inferring the ground-truth model: We provide Click [31] with a 4-state FSM that describes a stateful FW that only accepts packets from external hosts after a valid three-way handshake. We also constructed another 18-state FSM that describes a similar FW and a 3-state FSM that describes a source NAT (SNAT). In all three cases, *Alembic* inferred the ground-truth FSM.

B) Finding intent violations: We used a red-team exercise to evaluate the effectiveness of *Alembic* in finding intent violations in NF implementations. In each scenario, we modified the FSM from A to introduce violations and verified that the *Alembic*-generated model captured the behavior for all of the following four cases. A and B refer to an internal and external host, respectively: (1) a FW prevents the connection from being established by dropping SYN-ACK packets; (2) a FW proactively sends SYN-ACK upon receiving SYN from A to B; (3) a SNAT rewrites the packet to unspecified srcip-port; and (4) a SNAT rewrites a dstip-port. Some of these scenarios are inspired by real-world NFs.

C) Validating key learning: We wrote additional Click [31] NFs that track the number of TCP connections based on different keys. We applied the key learning algorithm to each and confirmed it identifies the correct key (Table 5 in §C).

8.2 Correctness with real NFs

As summarized in Table 1, we generated models for PfSense and ProprietaryNF FWs using both correct-seq and combined-seq sets. For the other NF types, we used only the correct-seq set because the FW models for these NFs already modeled the interaction of TCP packets in the presence of out-of-window packets. For an NF that uses dynamic modification (e.g., randNAT), we cannot correctly instantiate the model in the presence of RST-ACK and FIN-ACK packets (§B). Hence, we only showcased how this NF handles connection establishment. Untangle and HAProxy have SYN retries and spurious resets (i.e., temporal effects) that are beyond our current scope (§3.3) and could not be disabled. Thus, we again only model how these NFs handle connection establishment. Further, during our attempts to infer models, we discovered these two NFs are connection-terminating, where an external SYN_C packet interfered with the connection initiation attempt from the internal host, which violates our independence assumption. To make the learning tractable, we removed the SYN_C from the external interface for these connection-terminating NFs.

Complementary testing methodologies: Since we do not know the ground truth models and thus cannot report the coverage of *code paths* inside the NF, we used three approaches to validate the correctness of our models: (1) *iperf [3] testing*,

[9] The seq number is incremented by 1 for packets with a SYN or FIN flag set and otherwise, by the data size. T. The ack number for a side of a connection is 1 greater than any received packet's sequence number.

Table 3: **Time to infer a symbolic model (h: hours, m: min)**

NF (pkt type)	time	NF (pkt type)	time
PfSense FW (C)	11 m	ProprietaryNF FW (C)	48 h
PfSense FW (CI)	16 h	ProprietaryNF FW (CI)	25 h 18 m
PfSense NAT (C)	28 m	PfSense randNAT (C)	14 m
PfSense LB (C)	14 m	ProprietaryNF NAT (C)	48 h
Untangle FW (C)	37 m	HAProxy LB (C)	20 m

Table 4: **Scalability benefits of our design choices**

Runtime ($\|\Sigma\|$)	1 connection (Σ=6)	2 connections (Σ=12)	3 connection (Σ=18)
	26 min	10 hr	$>$ 3 days
Runtime (d in Wp-method)	d=1	d=2	d=3
	13 min	1 hr 10 min	7 hr

generating valid sequences of TCP packets; (2) *fuzz testing*, randomly picking a packet type and a concrete IP and port; and (3) *stress testing*, generating packets by first picking a packet type and selecting concrete IP and port values to activate at least one rule.

For each test run, we generated an arbitrary configuration. For NFs that take multiple rules (e.g., FW and NAT), we varied the number of rules between 1, 5, 20, and 100. For each concrete rule, we randomly sampled a field from the field type defined by the ConfigSchema. We ensured that we picked concrete configurations different from the ones used during the inference (§4). For FWs and NATs, the generated configurations were installed on one interface (i.e., internal). Further, as *Alembic* cannot compose models for multi-function NFs (i.e., a FW with NAT), we set allow rules on the FWs when we inferred models for NATs and LBs. For iperf [3] testing, we set up a client and a server and collect the traces on each interface. Because iperf [3] generates a deterministic sequence of packets, we only tested with 1 accept rule. For stress and fuzz testing, we generated sequences of 20, 50, 100, and 300 packets. In each setting, we measured model accuracy by calculating the fraction of packets for which the model produced the exact same output as the NF. Each setting is a combination of the NF vendor and type (e.g., PfSense FW with the correct-seq set), input packet type (e.g., 300 packets), and the number of rules (e.g., 20 rules).

Iperf testing: Our models predicted the behavior of all NFs with 100% accuracy.

Fuzz testing: Across all settings for ProprietaryNF and PfSense FWs (both combined-seq and correct-seq set), the accuracy was 100%. For PfSense and ProprietaryNF NATs, the accuracy was 99.8% to 100%.

Stress testing: We summarize the results in Table 2. For many NFs (e.g., ProprietaryNF and PfSense FWs), we see the lowest accuracy (e.g., 98%) for 1 rule with 300 packets. This is expected because our testing generates a long sequence of packets that the Wp-method with $d = 1$ did not probe. Also, given the same FW (e.g., PfSense FW), we observe higher accuracy for an NF modeled with the correct-seq set compared to that modeled using the combined-seq set. We confirm that the model learned using the combined-seq set is rather large ($>$ 100 states) resulting from the many ways in which the correct and incorrect packets can interact. Note that ProprietaryNF NAT correct-seq took 49 hours to model and ProprietaryNF FW combined-seq took 5 days to infer the model. Going back to our earlier requirements that we can afford several tens of hours (i.e., a couple days) for the offline stage, we ran the accuracy testing on an intermediate model inferred after 48 hours, which still achieved high accuracy. We did not perform fuzz or stress testing for Untangle FW and HAProxy LB. These NFs have temporal effects that result in mis-attribution, which is outside our scope (§3.3). We see that *Alembic* achieves high accuracy even with large configurations.

8.3 Scalability

We now evaluate the runtime of *Alembic*'s components.

Time to learn symbolic models: For each NF, we report the longest time to model one SymbolicRule as learning can be parallelized across symbolic rules. In all cases, we use 20 servers setup, except for with PfSense random NAT which used one. [8] The results are summarized in Table 3. In summary, we achieved our goal of inferring high-fidelity models in less than 48 hours. We find that the runtime depends on: (1) the size of the FSM and $|\Sigma|$, and (2) *Alembic* or NF-specific details (e.g., rebooting). For (1), as the size of $|\Sigma|$ was double for the combined-seq set, it took more than 48 hours to discover 72-state FSM (ProprietaryNF FW, combined-seq) but less than 26 hours for 79-state with the correct-seq set. For (2), discovering the NAT model ProprietaryNF NAT (correct-seq) took longer than the FW as the NAT inference ran in two phases (§6). Lastly, PfSense models take less time to infer as PfSense does not require rebooting, and has shorter Δ_{wait}.

Time to validate the key: We use PfSense FW (correct-seq) to report the time to infer the key. It took 6 hours to infer the key (e.g., 2 hrs for each test).

Time for the online stage: For ProprietaryNF FW, the time to compose a concrete model is 75 ms for 10 rules, 0.6 s for 100 rules, and 5 seconds for 1,000 rules. The result generalizes to other NFs.

Scalability benefits of our design choices: The insights to leverage compositional modeling and KeyLearning allow *Alembic* are critical in achieving reasonable runtime by reducing the size of Σ. Suppose one FW rule takes a source IP field takes a /16 prefix. Without KeyLearning, we need to infer a model with all 2^{16} connections. Similarly, for a configuration with 20 rules, we need to infer a model with all relevant connections. The top half of the Table 4 shows how the runtime drastically increases as we increase the number of connections using a Click-based [31] FWs from §8.1 (using just one worker). Further, we measured the runtime as a function of d in Wp-method (bottom of Table 4). Using $d = 1$, we

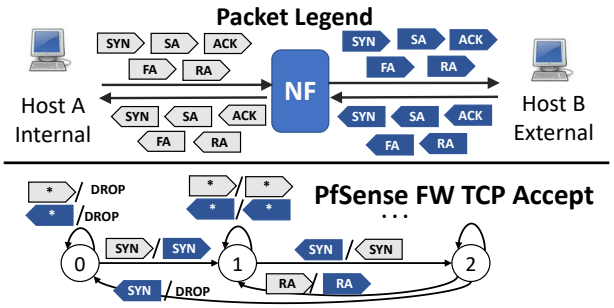


Figure 11: **The light/dark coloring indicates packets on host A/B's interface, respectively. The figure below shows the 3 states for PfSense FW accept rule**

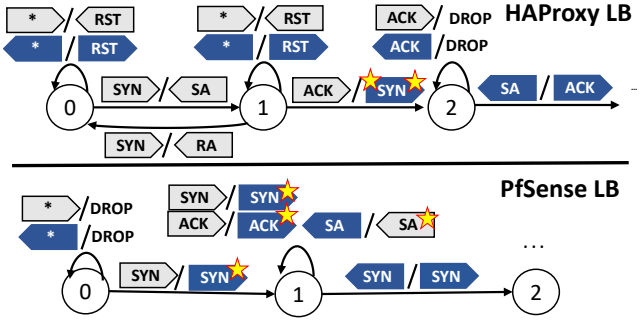


Figure 12: **First 3 states of the HAProxy and PfSense LB. Stars on head/tail of packets indicate src/dst modification**

were still able to infer the ground truth while reducing the runtime. These results demonstrate how reducing the size of $|\Sigma|$ is critical to obtain a reasonable runtime. Lastly, distributed learning helps scalability. The Click-based [31] FW with 18 states takes 1.6 hours with 1 worker but only 16 minutes with 19 workers (and 1 controller).

8.4 Case studies

We now highlight vendor-specific differences found using *Alembic*. For clarity, we present and discuss only partial representations of the inferred FSMs (as some FSMs are large).

Firewall (correct-seq): A common view of stateful FWs in many tools is a three-state abstraction (SYN, SYN-ACK, ACK) of the TCP handshake. Using *Alembic*, however, we discovered that the reality is significantly more complex. With a single FW accept rule, the inferred PfSense model (correct-seq) shows that a TCP SYN from an internal host, A, is sufficient for an external host, B, to send any TCP packets (Figure 11). Furthermore, FIN-ACK, which signals termination of the connection, does not cause a state transition. We find that ProprietaryNF FW has 79 states for a FW accept rule in contrast to 3 states for PfSense FW. ProprietaryNF, too, does not check for entire three-way handshake (e.g., only SYN, SYN-ACK). We find that the complexity of the FSM (i.e., 79 states) results from the number of ways in which the two TCP handshakes (from A and B) can interfere with each other. Such behavior could not have been exposed through handwritten models. Untangle FW actually behaves like a connection-terminating NF (Figure 13 in §A for partial model). The FW lets the first SYN from A through, but when B replies with a SYN-ACK, Untangle forwards it but preemptively replies with an ACK. When the A replies with ACK, Untangle drops it to prevent a duplicate.

Firewall (combined-seq): Surprisingly, for PfSense, we learned 257 states with combined-seq. The complexity is a result of packets with incorrect seq and ack causing state transitions, where many are forwarded. We learned a 72-state FSM for the ProprietaryNF FW after 48 hours and the full model (104-state) after 5 days. The cause for the larger FSM for PfSense is that the incorrect seq and ack packets often cause state transitions more frequently than ProprietaryNF FWs. Further, it is interesting to see how PfSense only had 3 states for the correct-seq set but 257 states with combined-seq, in contrast to ProprietaryNF where the number of states for both sets are similar. At a high-level, we find that obtaining such model is useful as it could possibly be used to generate a sequence of packets to bypass the firewall, but this is beyond the scope of our work.

Load balancer: HAProxy (Figure 12) follows the NF's connection-terminating semantics. It completes the TCP handshake with the client before sending packets to the server. After the handshake, the source of outgoing packets is modified to server-facing IP of LB, and destination is modified to the server (i.e., star on both-ends of TCP SYN in Fig. 12). In contrast, PfSense LB behaves like a NAT. When a client sends a SYN to the LB, the destination is modified to the server's IP (i.e., star in state 1 in Figure 12). Then, the LB modifies destination of packets from both client and server. We confirmed that PfSense indeed implements load balancing this way [37]. *Alembic* automatically discovered this without prior assumptions of any connection-termination behavior. Further, the connection-termination semantics of HAProxy differ from those of Untangle FW. Unlike HAProxy, Untangle lets SYN packets through and preemptively completes the connection with the external host. This is yet another example of non-uniformity across NF implementations.

8.5 Implications for network verification

We use two existing tools, BUZZ [22] and VMN [35], to demonstrate how *Alembic* can aid in network testing and verification. Using a Click-based [31] FW which adheres policy 1 and 2 (§2), we compare the test output using: (1) $M_{Alembic}$ inferred using *Alembic*, and (2) existing M_{hand} for FW. Using $M_{Alembic}$, BUZZ did not report a violation. Using M_{hand}, BUZZ reported a violation (*false-positive*) as 1 of 6 test traces did not match (trace in §2). Similarly, for policy 2, BUZZ reported a violation using M_{hand}. The failed test case is: $\text{SYN}^{\text{Internal}}_{\text{A}\to\text{B}}$, $\text{RST}^{\text{External}}_{\text{B}\to\text{A}}$, $\text{RST}^{\text{External}}_{\text{B}\to\text{A}}$. M_{hand} predicts that both RST packets are dropped, as the model does not check for RST flags. However, Click NF allows the first RST packet to

reset the NF state. We also plugged the model for PfSense into a network verification tool, VMN [35]. The existing model in VMN does not check TCP flags. Using VMN, we verified the property: "TCP packets from an external host, B, can reach A even if no SYN packet is sent from A." Recall that in PfSense, $SYN^{Internal}_{A \to B}$ needs to be sent for B to send TCP packets to A. Hence, the property is NOT SATISFIED. Using $Model_{hand}$, the tool returned that the property is SATISFIED whereas using $Model_{Alembic}$ indicated that it is not (i.e., *false-negative* for $Model_{hand}$).

9 Related Work

NF modeling: There is a large body of work on understanding and modeling NFs [19, 26, 30, 35, 39]. Joseph and Stoica [30] propose a language to model *stateless* NFs to ease the NF deployment process. NFactor [45] uses code analysis techniques to extract packet forwarding models in the form of a match-action table. While this can be complementary, it may be difficult to obtain source-code for proprietary NFs. Some works focus on the NF internal states and how to manage them [26, 39]. Our work is orthogonal as we focus on NF behavioral models of externally-visible actions.

FSM inference: L* algorithm by Angluin lays the foundation for learning the FSM [12]. The techniques of learning FSMs has been used for model checking blackbox systems (e.g., [28,36]). Symbolic finite automata (SFA) [42] are FSMs where the alphabet is given by a Boolean algebra with an infinite domain. While *Alembic* does not directly formulate the problem to infer SFAs, we use the homogeneity assumption in the IP and port ranges to learn a *symbolic model*. Hence, using abstractions like SFA may help us to naturally embed symbolic encodings. We could potentially leverage a tool (e.g., [20]) that extends L* to infer the SFA. However, using SFA does not address the NF-specific challenges (e.g., inferring the key, handling modifications) but may serve as the basis for interesting future work.

NF model use cases: Many network testing and verification tools need NF models [22,35,43]. Some [35,40] proposed new modeling languages to represent NFs. However, it is unclear how to represent existing NFs using these languages. Symnet [40] wrote parsers to automatically generate NF models using their language, SEFL. Again, it is unclear whether the parser generalizes to other FWs or to arbitrary configurations. However, not all network verification tools require models. Vigor [46] uses the C code of a NAT to verify properties such as memory-safety, which are orthogonal to our approach.

Application of L*: L* has been used to discover protocol vulnerabilities (e.g., [15, 16]) or specific attacks (e.g., cross-site scripting) against web-application firewalls [13]. However, these approaches do not tackle the NF-specific challenges (e.g., handling large configuration space and header modifications). Other works also use L* to infer models of various protocols (e.g., TLS [18]). While Fiterau-Brostean et al. [23, 24] inferred the behavior of TCP/ IP implementations in an operating system, these tools leverage a simple extension of L* and cannot model NFs with a large configuration space.

10 Discussion

Before we conclude, we discuss outstanding issues.

Handling more protocols: NFs such as layer-7 load balancers (LB), transparent proxy, or deep packet inspection (DPI) operate at the application layer. To model these cases, *Alembic* needs to generate relevant input packet types for these protocols (e.g., GET, POST, PUT for HTTP). However, the main challenge is to model the multi-layer interactions.

Representing complex NFs: Some NFs exhibit complex actions that cannot be captured with "packet in and packet out" semantics. For instance, to represent quantitative properties (e.g., rate-limiting), we need to incorporate them as part of the input alphabet (e.g., "sessions sent at a certain rate") and monitor relevant properties to classify NF actions. Similarly, to handle temporal effects (e.g., timeout), we need to add the passage of time (e.g., wait 30 s) to the input alphabet. While we could extend our current infrastructure to handle these NFs, it may be worthwhile to consider more native abstractions other than deterministic FSMs. For instance, many have proposed different abstractions to represent quantitative properties (e.g., [9, 11, 29, 33]) and timing properties (e.g., [10]). Once we pick the abstraction, we can find relevant techniques that extend L* (i.e., [14, 27]). It is difficult to find one abstraction to model multiple properties at once, and we need to pick the abstraction based on the properties of interest.

Handling more complex ConfigSchema: To handle more complex configuration semantics such as "if condition, do X," and "go to rule X", we still need to model a rule type (e.g., both X in the above example) similar to our workflow. To incorporate new processing semantics, we need to change how we compose individual models in the online stage.

11 Conclusions

We proposed *Alembic*, a system to automatically synthesize NF models. To tackle the challenges stemming from large configuration spaces, we synthesize NF models viewed as an ensemble of FSMs. *Alembic* consists of an offline stage that learns symbolic models and an online stage to compose concrete models given a configuration. Our evaluation shows that *Alembic* is accurate, scalable, and enables more accurate network verification. While *Alembic* demonstrates the promise of NF model synthesis, there remain some open challenges (§3.3 and §10) that present interesting avenues for future work.

12 Acknowledgments

We thank our shepherd, Aurojit Panda, and the anonymous reviewers for their suggestions. We also thank Tianhan Hu for his help in implementing a distributed version of *Alembic*, and Bryan Parno, Swarun Kumar, Matthew Mccormack, and Adwait Dongare for providing feedback on this paper. This work was funded in part by NSF awards CNS-1440065, CNS-1552481, CCF-1703925, and CCF-1763970.

References

[1] Cloudlab. https://www.cloudlab.us/.

[2] Haproxy. http://www.haproxy.org/.

[3] iPerf Performance Tool. https://iperf.fr/.

[4] jsonrpc. https://github.com/briandilley/jsonrpc4j.

[5] pfsense. https://www.pfsense.org/.

[6] Scapy. http://www.secdev.org/projects/scapy/.

[7] untangle. https://www.untangle.com/.

[8] Virtualbox. https://www.virtualbox.org/.

[9] ALUR, R., DANTONI, L., DESHMUKH, J., RAGHOTHAMAN, M., AND YUAN, Y. Regular functions and cost register automata. In *Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on* (2013), IEEE, pp. 13–22.

[10] ALUR, R., AND DILL, D. L. A theory of timed automata. *Theor. Comput. Sci. 126*, 2 (April 1994), 183–235.

[11] ALUR, R., FISMAN, D., AND RAGHOTHAMAN, M. Regular programming for quantitative properties of data streams. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632* (New York, NY, USA, 2016), Springer-Verlag New York, Inc., pp. 15–40.

[12] ANGLUIN, D. Learning regular sets from queries and counterexamples. *Inf. Comput. 75*, 2 (November 1987), 87–106.

[13] ARGYROS, G., STAIS, I., JANA, S., KEROMYTIS, A. D., AND KIAYIAS, A. Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 1690–1701.

[14] BALLE, B., AND MOHRI, M. Learning weighted automata. In *Algebraic Informatics* (Cham, 2015), A. Maletti, Ed., Springer International Publishing, pp. 1–21.

[15] CHO, C. Y., BABI Ć, D., SHIN, E. C. R., AND SONG, D. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 426–439.

[16] CHO, C. Y., BABIĆ, D., POOSANKAM, P., CHEN, K. Z., WU, E. X., AND SONG, D. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 10–10.

[17] CHOW, T. S. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng. 4*, 3 (May 1978), 178–187.

[18] DE RUITER, J., AND POLL, E. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., 2015), USENIX Association, pp. 193–206.

[19] DETAL, G., HESMANS, B., BONAVENTURE, O., VANAUBEL, Y., AND DONNET, B. Revealing middlebox interference with tracebox. In *Proceedings of the 2013 Conference on Internet Measurement Conference* (New York, NY, USA, 2013), IMC '13, ACM, pp. 1–8.

[20] DREWS, S., AND D'ANTONI, L. Learning symbolic automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2017), Springer, pp. 173–189.

[21] EDELINE, K., AND DONNET, B. On a middlebox classification, 2017.

[22] FAYAZ, S. K., YU, T., TOBIOKA, Y., CHAKI, S., AND SEKAR, V. Buzz: Testing context-dependent policies in stateful networks. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2016), NSDI'16, USENIX Association, pp. 275–289.

[23] FITERĂU-BROŞTEAN, P., JANSSEN, R., AND VAANDRAGER, F. Learning fragments of the tcp network protocol. In *International Workshop on Formal Methods for Industrial Critical Systems* (2014), Springer, pp. 78–93.

[24] FITERĂU-BROŞTEAN, P., JANSSEN, R., AND VAANDRAGER, F. Combining model learning and model checking to analyze tcp implementations. In *International Conference on Computer Aided Verification* (2016), Springer, pp. 454–471.

[25] FUJIWARA, S., VON BOCHMANN, G., KHENDEK, F., AMALOU, M., AND GHEDAMSI, A. Test selection based on finite state models. *IEEE Trans. Softw. Eng. 17*, 6 (June 1991), 591–603.

[26] GEMBER-JACOBSON, A., VISWANATHAN, R., PRAKASH, C., GRANDL, R., KHALID, J., DAS, S., AND AKELLA, A. Opennf: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 163–174.

[27] GRINCHTEIN, O. *Learning of timed systems*. PhD thesis, Acta Universitatis Upsaliensis, 2008.

[28] GROCE, A., PELED, D., AND YANNAKAKIS, M. Adaptive model checking. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (London, UK, UK, 2002), TACAS '02, Springer-Verlag, pp. 357–370.

[29] HENZINGER, T. A. *The Theory of Hybrid Automata*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 265–292.

[30] JOSEPH, D., AND STOICA, I. Modeling middleboxes. *Netwrk. Mag. of Global Internetwkg.* (2008).

[31] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The click modular router. *ACM Trans. Comput. Syst. 18*, 3 (August 2000), 263–297.

[32] MAKAYA, C., AND FREIMUTH, D. Automated virtual network functions onboarding. In *IEEE SDN-NFV Conference* (2016).

[33] MOHRI, M. Weighted automata algorithms. In *Handbook of weighted automata*. Springer, 2009, pp. 213–254.

[34] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., AND WALKER, D. Composing software defined networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX Association, pp. 1–13.

[35] PANDA, A., LAHAV, O., ARGYRAKI, K., SAGIV, M., AND SHENKER, S. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association.

[36] PELED, D., VARDI, M. Y., AND YANNAKAKIS, M. Black box checking. *J. Autom. Lang. Comb. 7*, 2 (November 2001), 225–246.

[37] PFSENSE. Inbound Load Balancing. https://doc.pfsense.org/index.php/Inbound_Load_Balancing.

[38] RAFFELT, H., STEFFEN, B., AND BERG, T. Learnlib: A library for automata learning and experimentation. In *Proc. ACM FMICS 2005*.

[39] RAJAGOPALAN, S., WILLIAMS, D., JAMJOOM, H., AND WARFIELD, A. Split/merge: System support for elastic execution in virtual middleboxes. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013).

[40] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. Symnet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 314–327.

[41] TSCHAEN, B., ZHANG, Y., BENSON, T., BENERJEE, S., LEE, J., AND KANG, J.-M. SFC-Checker: Checking the Correct Forwarding Behavior of Service Function Chaining. In *IEEE SDN-NFV Conference* (2016).

[42] VEANES, M., HALLEUX, P. D., AND TILLMANN, N. Rex: Symbolic regular expression explorer. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation* (Washington, DC, USA, 2010), ICST '10, IEEE Computer Society, pp. 498–507.

[43] VELNER, Y., ALPERNAS, K., PANDA, A., RABINOVICH, A., SAGIV, M., SHENKER, S., AND SHOHAM, S. Some complexity results for stateful network verification. In *Proceedings of the 22Nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636* (New York, NY, USA, 2016), Springer-Verlag New York, Inc., pp. 811–830.

[44] WANG, Z., CAO, Y., QIAN, Z., SONG, C., AND KRISHNAMURTHY, S. V. Your state is not mine: A closer look at evading stateful internet censorship. In *Proceedings of the 2017 Internet Measurement Conference* (New York, NY, USA, 2017), IMC '17, ACM, pp. 114–127.

[45] WU, W., ZHANG, Y., AND BANERJEE, S. Automatic synthesis of nf models by program analysis. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2016), HotNets '16, ACM, pp. 29–35.

[46] ZAOSTROVNYKH, A., PIRELLI, S., PEDROSA, L., ARGYRAKI, K., AND CANDEA, G. A formally verified nat. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 141–154.

A Partial FSM for Use Cases

Figure 13 shows partial FSM for Untangle FW accept, drop, default rule, and ProprietaryNF accept rule.

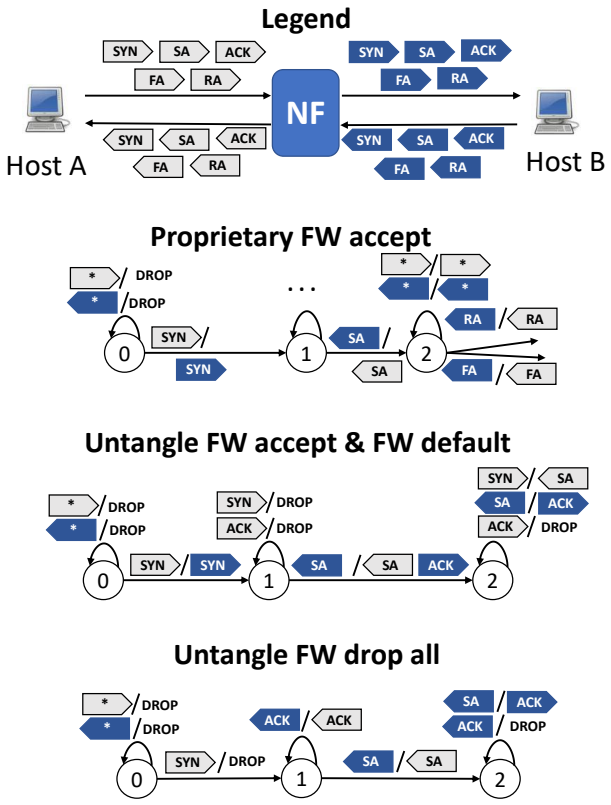


Figure 13: Partial FSM for Untangle FW accept, drop, default rule, and ProprietaryNF accept rule

B Instantiating a Concrete Model

We present a detailed description of how we instantiate a concrete model in our *online* stage. We consider three cases: (1) NFs that keep per-connection state but do not modify headers, (2) NFs that keep per-connection state and do, and (3) NFs that keep state according to other keys but do not modify headers. We do not consider header-modifying NFs that keep state according to other keys (e.g., per-source) as they are outside our current scope. For simplicity, we assume a perfect Equivalence Oracle such that the generated symbolic model from the *offline* stage is identical to the ground truth.

Case 1) NFs that keep per-connection state but do not modify headers

For NFs that do not modify packet headers, we define a key with (A:Ap1, B:Bp1) where A:Ap1 is a srcip-port and B:Bp1 is a dstip-port. Note that matches for a per-connection key are bi-directional; a packet with srcip-port, B:Bp1, and dstip-port,

A:Ap1, would also match the key, (A:Ap1, B:Bp1). Then, for each concrete value of the key in a rule, we instantiate a concrete FSM.

We posit that our instantiation logic is correct for an input packet type with *all TCP packet* types (e.g., SYN, SYN-ACK, ACK, RST-ACK) for the following reasons:

1. A model learned using one connection from the offline stage represents the ground truth (assuming a perfect Equivalence Oracle).
2. Because we assume each connection is independent and shares the same logical behavior (from §3.3 and Def 4 in §C), cloning a model learned from one connection to represent other connections does not introduce errors.

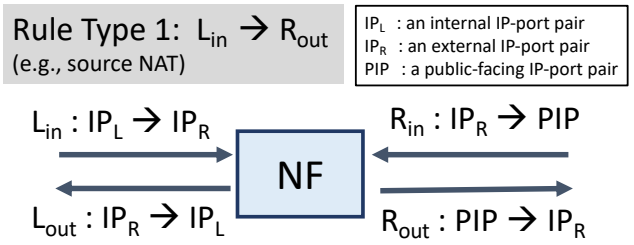


Figure 14: **NAT example**

Case 2) NFs that keep per-connection state and do modify headers

We extend the NF operational model presented in Alg 1 to instantiate a concrete model for header-modifying NFs. Recall that in the *Alembic*'s *offline* stage, we learn a model using a range, where we infer a model using a symbolic IP and port in a range. For header-modifying NFs, even though the learned model contains symbolic IPs and ports, our instantiation logic is correct because each concrete model is indexed with a concrete IP and port (Algo. 2).

Consider a NAT with two rule types defined in its ConfigSchema.

1. Rule Type 1: $L_{in} \rightarrow R_{out}$ where the initial modification for a new connections happens for L_{in} (e.g., modifying the source IP of an internal IP to a public-facing IP).

2. Rule Type 2 : $R_{in} \rightarrow L_{out}$ where the initial modification for a new connections happens for R_{in} (e.g., port forwarding where the port 8080 from the R interface is forwarded to port 80 on the internal server).

For ease of explanation, we first show how we instantiate a concrete model for a model inferred for *rule type 1* and later describe how we can easily extend our design to handle rule type 2. Figure 14 shows the ranges of valid source and destination IPs and ports for located packets for a NAT configured with a concrete rule for rule type 1 (e.g., a valid ranges for L_{in} is IP_L for a srcip pair and IP_R for a dstip-port pair).

Algorithm 2 Instantiating a model for a per-connection NF with header modifications

```
1:  function OnlineForModification(locatedPkt p, Rule r,
    Map[rule, Map[key, state]] stateMap, T_{L→R}, T_{R→L})
2:      if p.interface == L then
3:          p_out = FwdDirection(p, r, stateMap, T_{L→R}, T_{R→L})
4:      else
5:          p_out = ReverseDirection(p, r, stateMap, T_{L→R},
        T_{R→L})
6:      return p_out

7:  function FwdDirection(locatedPkt p, Rule r, Map[rule,
    Map[key, state]] stateMap, T_{L→R}, T_{R→L})
8:      if NewConnection then
9:          Update T_{L→R}, T_{R→L}
10:     Extract FSM, currentState
11:     p_out, nextState ← Get action from the FSM
12:     Update currentState with nextState
13:     return p_out

14: function ReverseDirection(locatedPkt p, Rule r, Map[rule,
    Map[key, state]] stateMap, T_{L→R}, T_{R→L})
15:     if p ∈ T_{R→L} then
16:         Extract FSM, currentState
17:         p_out, nextState ← Get action from the FSM
18:         Update currentState with nextState
19:     else
20:         Extract default FSM, currentState
21:         p_out, nextState ← Get action from default FSM
22:         Update currentState with nextState
23:     return p_out
```

To tackle the challenge above, we introduce two maps to associate an output (or modified) packet's 5-tuple to the corresponding input packet's 5-tuple for both interfaces. Specifically, we use $T_{L\rightarrow R}$ to map L_{in} to R_{out}, and $T_{R\rightarrow L}$ to map R_{in} to L_{out} (Algo 2). Algo 2 is a detailed description after Line 3 in the operational model (Algo 1 in §3). Note that for each of presentation, we assume we found a rule to apply (Line 3 in Algo 1) for the incoming packet.

If an NF receives a packet from the L interface, the algorithm checks whether the packet is a new connection by performing a lookup in the map (in FwdDirection). If the connection does not already exist in the map, we update the $T_{L\rightarrow R}$ with $(IP_L, IP_R) \rightarrow (PIP, IP_R)$ and $T_{R\rightarrow L}$ with $(IP_R, PIP) \rightarrow (IP_R, IP_L)$. Then, we extract the corresponding FSM and the current state (or the initial state if a new connection) to apply the appropriate action (i.e., determine p_{out}). If the incoming packet is from the R interface, we look up the corresponding map, $T_{R\rightarrow L}$, to fetch the original IP-port (e.g., IP_L). Then, it uses the key to fetch the FSM and determine the appropriate action for the incoming packet. If the entry does not exist in the map, our concrete model instead uses the FSM associated with the NF's default behavior. Note

that in the case of static header modification, such as a NAT configured with a list of static mappings between internal and external IP-port pairs, we prepopulate $T_{L\to R}$ and $T_{R\to L}$ with these static mappings. Hence, for an NF that statically modify packet headers, we will not reach Line 20 as these mapping already exist.

Extending for Rule Type 2 : We now discuss how to adapt the above framework to handle *rule type 2* where the initial modification happens for packet entering the other interface (e.g., R_{in}). In contrast to rule type 1, an NF configured with a concrete rule for rule type 2 initially modifies packet header for R_{in} (i.e., not L_{in}). We need to make two changes in Algo 2:

1. Line 2 must change to call FWDDIRECTION (Line 7) if the packet comes via the R interface.
2. For the corresponding packet coming from the reverse direction (i.e., L_{in} for rule type 2), we need to perform a look up in $T_{L\to R}$ to check if the reverse mapping exists instead of $T_{R\to L}$ (i.e., change Line 15).

Note that our approach does not need a priori knowledge of which rule type the NF is configured with. We just need to infer at which interface the initial modification happens by parsing the generated model. For instance, if the initial modification happens for L_{in} (i.e., rule type 1), then we follow the original algorithm shown in Algo. 2. If the initial modification happens for R_{in} (i.e., rule type 2), then we follow the algorithm in Algo. 2 with two changes mentioned above.

The above algorithm describes how we instantiate a concrete FSM. Now, there are two types of modifications. In the case of static modification, we know the value of the modified packet a priori for a given incoming packet, so we can prepopulate the concrete FSMs with all the known IPs and ports. However, in the case of dynamic modification where we cannot predict the modified values in advance, we initialize an ensemble of concrete FSMs with *symbolic* IP and port (for the modified values) and bind them to concrete IPs and ports as they are revealed (i.e., after injecting packets and observing outputs).

Given this context, we posit the correctness of these instantiated models (formal proof is outside our current scope). For per-connection NFs with static header modifications, our instantiation of FSMs is correct with an input packet type of *all TCP packet types*, for the same two reasons described for case 1. We now state additional reasonings:

1. The same 5-tuple for an input packet maps to the same 5-tuple for the output packet, and $T_{L\to R}$ and $T_{R\to L}$ store these mappings. Thus, we will correctly discover the reverse mapping during the instantiation.
2. Even in the presence of connection resets, the same 5-tuple will be mapped to the same output (i.e., 5-tuple). Hence, the model for each connection is correct even in the presence of packets that reset the connection state (i.e., we can reuse the previous mappings stored).

Table 5: **Validating the correctness of KeyLearning using Click-based NFs (§8.1)**

Ground Truth	Test1	Test2	Test3	Result
Cross-conn	Y			Cross-conn
Per-src	N	Y		Per-src
Per-dst	N	N	Y	Per-dst
Per-conn	N	N	N	Per-conn

For NFs that dynamically modify packet headers, we posit that for the input set of *TCP-handshake packets* (i.e., SYN, SYN-ACK, ACK). However, when we receive a TCP packet that resets a connections (e.g., RST-ACK), the concrete IP and port that was bound to a symbolic IP and port will change (i.e., after a reset, srcip-port maps to P:Pp2 instead of P:Pp1). Hence, the generated model will continue to use the mapping already stored in $T_{L\to R}$ and $T_{R\to L}$, resulting in inaccurate model.

Case 3: NFs that do not keep per-connection state

We now consider NFs that do not modify packet headers but have keys other than per-connection. Recall the following key types and their corresponding header fields:

1. *Per-source key*, defined by a source IP
2. *Per-destination key*, defined by a destination IP
3. *Cross-connection key*, defined by *any* packet (i.e., all IP and ports with the range)
4. *Stateless key*, defined by srcip-port and dstip-port. Note that we view the stateless NF as keeping a per-connection state but the FSM is always just a single state.

When we instantiate an ensemble of concrete FSMs for an NF that keeps per-source state, the IPs and ports that do not define the key (i.e., srcport, dstip, and dstport) refer to ranges of values. Hence, the model for a srcip should accept *ANY* srcport, dstip, and dstport within the specified range.

We *posit* that our instantiation logic outputs a correct model for an input packet type, with all TCP-relevant symbols (i.e., All TCP-relevant symbols as there are no modifications) if the per-source NF adheres to Def. 2 (§C):

1. Our definition for per-source NF assumes that all destinations given the same source IP are treated homogeneously. Hence, it is correct to use the model learned from one connection and simply replace the symbolic destination in the model to any destination IP that appears in the configuration.
2. As we assume no header modification, the instantiated model is correct for all TCP-relevant symbols.

We omit the cases for per-destination, cross-connection, and stateless for brevity. The correctness arguments for these cases are similar to that of per-source NFs.

C Correctness of KeyLearning

We formalize the definition of the granularities of states maintained by NFs (i.e., keys) and prove the correctness of our

KeyLearning algorithm in §5.

Recall that each NF SymbolicRule (1 rule) consists of multiple configuration fields (e.g., FW needs to be configured to allow packets from a subnet X to Y). To simplify the presentation, let us consider a rule r in an NF that takes two configuration fields, namely source and destination, and thus also omit configuration fields that do not affect the key (e.g., an action and a load-balancing algorithm that do not affect the key). We use $NF_r^{\langle X,Y\rangle}$ to refer to an NF instance only with the targeted rule r that is configured with source X and destination Y. Given such an NF instance, we use L* to learn a model from it. Particularly, let $L^{\Gamma}(NF_r^{\langle X,Y\rangle})$ refer to the FSM learned by L* for the NF instance $NF_r^{\langle X,Y\rangle}$ using packets only from the set $\Gamma \subset X \times Y \cup Y \times X$. We assume that the FSM learned by L* is correct with respect to the NF instance. That is, given any sequence of packets with source a and destination b, running $L^{\Gamma}(NF_r^{\langle X,Y\rangle})$ on it obtains the same output sequence as running $NF_r^{\langle X,Y\rangle}$ on it, provided that $(a,b) \in X \times Y$ or $(a,b) \in Y \times X$.

Definition of keys: To prove the correctness of our KeyLearning algorithm, we first formalize the definition of NF keys. The following table summarizes the notations we use.

Term	Definition
Σ	the set of packets (symbol for FSMInference)
$\Sigma^{(X,Y)}$	the set of packets with source (destination, resp.) IP from X (Y, resp.)
$\sigma\|_{(a,b)}$	Given σ and a source-destination pair) (a,b), $\sigma\|_{(a,b)}$ is the sequence of packets obtained from σ by removing all packets that are not with source a and destination b.
$\sigma\|_{(a,b),(b,a)}$	Similar to above, but also keeps packets with source b and destination a.
$\sigma{+}{+}(a,b)$	The sequence obtained by appending (a,b) to the sequence σ
$NF_r^{\langle X,Y\rangle}(\sigma)$	the output of the last packet given σ to the NF configured with $r^{\langle X,Y\rangle}$

The definition of keys is given as follows.

Definition 1 (Cross-connection NF)**.** *A rule r in an NF keeps* cross-connection *state iff for all NF instances $NF_r^{\langle X,Y\rangle}$, all pairs of connections (a,b) and (c,d) such that $a,c \in X$, $b,d \in Y$, and $(a,b) \neq (c,d)$, there exists a sequence $\sigma \in \Sigma^{(a,b)}$, such that $NF_r^{\langle X,Y\rangle}(\sigma{+}{+}(c,d)) \neq NF_r^{\langle X,Y\rangle}((c,d))$.*

Definition 2 (Per-source NF)**.** *A rule r in an NF keeps per-source state if all its instance $NF_r^{\langle X,Y\rangle}$ satisfies the three conditions:*

1. for all $a \in X$ and $b \in Y$, there exits a σ over $\Sigma^{(\{a\},Y)}$, such that $NF_r^{\langle X,Y\rangle}(\sigma{+}{+}(a,b)) \neq NF_r^{\langle X,Y\rangle}((a,b))$.

2. for all $a \in X$, $b \in Y$, and σ_1, σ_2 over $\Sigma^{(\{a\},Y)}$ such that σ_1 and σ_2 have the same length, $NF_r^{\langle X,Y\rangle}(\sigma_1{+}{+}(a,b)) = NF_r^{\langle X,Y\rangle}(\sigma_2{+}{+}(a,b))$.

3. for all $a \in X$, $b \in Y$, and σ over $\Sigma^{(X,Y)}$, $NF_r^{\langle X,Y\rangle}(\sigma{+}{+}(a,b)) = NF_r^{\langle X,Y\rangle}(\sigma|_{(a,\_)}{+}{+}(a,b))$.

Definition 3 (Per-destination NF)**.** *A rule r in an NF keeps per-destination state if all its instance $NF_r^{\langle X,Y\rangle}$ satisfies the three conditions:*

1. for all $a \in X$ and $b \in Y$, there exits a σ over $\Sigma^{(X,\{b\})}$, such that $NF_r^{\langle X,Y\rangle}(\sigma{+}{+}(a,b)) \neq NF_r^{\langle X,Y\rangle}((a,b))$.

2. for all $a \in X$, $b \in Y$, and σ_1, σ_2 over $\Sigma^{(X,\{b\})}$ such that σ_1 and σ_2 have the same length, $NF_r^{\langle X,Y\rangle}(\sigma_1{+}{+}(a,b)) = NF_r^{\langle X,Y\rangle}(\sigma_2{+}{+}(a,b))$.

3. for all $a \in X$, $b \in Y$, and σ over $\Sigma^{(X,Y)}$, $NF_r^{\langle X,Y\rangle}(\sigma{+}{+}(a,b)) = NF_r^{\langle X,Y\rangle}(\sigma|_{(\_,b)}{+}{+}(a,b))$.

Definition 4 (Per-connection NF)**.** *A rule r in an NF keeps per-connection state if all its instance $NF_r^{\langle X,Y\rangle}$ satisfies the two conditions:*

1. for all $(a,b) \in X \times Y \cup Y \times X$, there exits a σ over $\Sigma^{(\{a\},\{b\})} \cup \Sigma^{(\{b\},\{a\})}$, such that $NF_r^{\langle X,Y\rangle}(\sigma{+}{+}(a,b)) \neq NF_r^{\langle X,Y\rangle}((a,b))$.

2. for all $(a,b) \in X \times Y \cup Y \times X$, and σ over $\Sigma^{(X,Y)} \cup \Sigma^{(Y,X)}$, $NF_r^{\langle X,Y\rangle}(\sigma{+}{+}(a,b)) = NF_r^{\langle X,Y\rangle}(\sigma|_{(a,b),(b,a)}{+}{+}(a,b))$.

Definition 5 (stateless)**.** *A rule r in an NF is called a stateless NF iff for all NF instance $NF_r^{\langle X,Y\rangle}$, packet $p \in \Sigma^{(X,Y)}$, and sequence σ over $\Sigma^{(X,Y)}$, $NF_r^{\langle X,Y\rangle}(\sigma{+}{+}p) = NF_r^{\langle X,Y\rangle}(p)$.*

In addition, we assume all NFs satisfy the following consistency in the configuration space:

Definition 6 (Consistency in the configuration space)**.** *For all A,B,X,Y,σ such that $A \subset X$, $B \subset Y$ and σ is a sequence over $\Sigma^{(A,B)}$, $NF_r^{\langle X,Y\rangle}(\sigma) = NF_r^{\langle A,B\rangle}(\sigma)$.*

FSM composition: The definition of FSM composition is given below.

Definition 7 (FSM composition for key learning)**.** *Given two FSMs $FSM_i = (S_i, \Sigma_i, \Delta_i, \delta_i, s_i^0)$, where S_i is the state space, Σ_i is the space of possible input symbols such that $\Sigma_1 \cap \Sigma_2 = \emptyset$, Δ_i is the set of output symbols, $\delta_i : S_i \times \Sigma_i \to S_i \times \Delta_i$ is the transition function, and $s_i^0 \in S_i$ is the initial state of FSM_i, the composite FSM of FSM_1 and FSM_2 is $FSM_{composite} = (S_1 \times S_2, \Sigma_1 \cup \Sigma_2, \Delta_1 \cup \Delta_2, \delta, s_1^0 \times s_2^0)$, where $\delta((s_1,s_2),p) = ((s_1',s_2'),p')$ if and only if 1) $\delta_1(s_1,p) = (s_1',p')$ and $s_2 = s_2'$; or 2) $\delta_1(s_2,p) = (s_2',p')$ and $s_1 = s_1'$.*

Proof of KeyLearning algorithm: The correctness of our KeyLearning algorithm is given in the following theorem.

Theorem 1 (Correctness of KeyLearning)**.** *Figure 10 is correct.*

Proof Sketch. For brevity, we only prove the column for the per-source NF; proofs of other columns are similar. The proof for per-source NF follows from the three lemmas below. □

Lemma 1. *All NFs that keep per-source state cannot pass Test 1.*

Proof. Let A_1 and A_2 be the FSM learned for $NF_r^{\langle\{a\},\{b\}\rangle}$ and $NF_r^{\langle\{c\},\{d\}\rangle}$ respectively (i.e., $A_1 = L^{\{(a,b)\}}(NF_r^{\langle\{a\},\{b\}\rangle})$, similarly for A_2), B be the FSM learned for $NF_r^{\langle\{a,c\},\{b,d\}\rangle}$ using packets from (a,b) and (c,d) (i.e., $B = L^{\{(a,b),(c,d)\}}(NF_r^{\langle\{a,c\},\{b,d\}\rangle})$), and C be the FSM composed of A_1 and A_2. We only need to prove that for any sequence σ consisting of packets over $\{(a,b),(c,d)\}$, $B(\sigma) = C(\sigma)$. W.L.O.G., suppose σ ends with (a,b). Then $B(\sigma) = NF_r^{\langle\{a,c\},\{b,d\}\rangle}(\sigma) = NF_r^{\langle\{a,c\},\{b,d\}\rangle}(\sigma|_{(a,b)}) = B(\sigma|_{(a,b)})$ (condition 3), $C(\sigma) = C(\sigma|_{(a,b)}) = A_1(\sigma|_{(a,b)})$ (the first equality is by condition 3 and the second is by FSM composition). But by homogeneity in the config space, $A_1(\sigma|_{(a,b)}) = B(\sigma|_{(a,b)})$. Thus, $B(\sigma) = C(\sigma)$. In other words, B is equivalent to C. □

Lemma 2. *All NFs that keep per-source state can pass Test 2.*

Proof. Let A_1 and A_2 be the FSM learned for $NF_r^{\langle\{a\},\{b\}\rangle}$ and $NF_r^{\langle\{a\},\{c\}\rangle}$ respectively, B be the FSM learned for $NF_r^{\langle\{a\},\{b,c\}\rangle}$, and C be the FSM composed of A_1 and A_2. By the first condition of per-source NF, there exists a σ over $\Sigma^{\langle\{a\},\{b,c\}\rangle}$, such that $B(\sigma++(a,b)) \neq B((a,b))$. By the second condition, $B(\sigma++(a,b)) = B(\sigma'++(a,b))$, where σ' is a sequence consisting of only (a,c). Since C is composed of A_1 and A_2, $C(\sigma'++(a,b)) = A_1((a,b))$. But by homogeneity in the config space, $A_1((a,b)) = B((a,b))$. Thus, $C(\sigma'++(a,b)) \neq B(\sigma'++(a,b))$. In other words, B is not equivalent to the composite FSM of A_1 and A_2. □

Lemma 3. *All NFs that keep per-source state cannot pass Test 3.*

Proof. Let A_1 and A_2 be the FSM learned for $NF_r^{\langle\{a\},\{b\}\rangle}$ and $NF_r^{\langle\{c\},\{b\}\rangle}$ respectively, B be the FSM learned for $NF_r^{\langle\{a,c\},\{b\}\rangle}$, and C be the FSM composed of A_1 and A_2. Consider any sequence σ over $\Sigma^{\langle\{a,c\},\{b\}\rangle}$. W.L.O.G., suppose σ ends with (a,b). Then by condition 3, $B(\sigma) = B(\sigma|_{(a,b)})$. By definition of composition, $C(\sigma) = A_1(\sigma|_{(a,b)})$. But by homogeneity in the config space, $A_1(\sigma|_{(a,b)}) = B(\sigma|_{(a,b)})$. Thus, $C(\sigma) = B(\sigma)$. In order words, B and C are equivalent. □