Pryde: A Modular Generalizable Workflow for Uncovering Evasion Attacks Against Stateful Firewall Deployments

Soo-Jin Moon†*, Milind Srivastava†*, Yves Bieri§, Ruben Martins†, Vyas Sekar†
†Carnegie Mellon University, §Compass Security

Abstract—**Stateful firewalls (SFW) play a critical role in securing our network infrastructure. Incorrect implementation of the intended stateful semantics can lead to evasion opportunities, even if firewall rules are configured correctly. Uncovering these opportunities is challenging due to the (1) black-box and proprietary nature of firewalls; (2) diversity of deployments; and (3) complex stateful semantics. To tackle these challenges, we present *Pryde*. *Pryde* uses a modular model-guided workflow that generalizes across black-box firewall implementations and deployment-specific settings to generate evasion attacks. *Pryde* infers a behavioral model of the stateful firewall in the presence of potentially non-TCP-compliant packet sequences. It uses this model in conjunction with attacker capabilities and victim behavior to synthesize custom evasion attacks. Using *Pryde*, we identify more than 6,000 unique attacks against 4 popular firewalls and 4 host networking stacks, many of which cannot be uncovered by prior work on censorship circumvention and black-box fuzzing.**

Index Terms—**network security, stateful firewalls, evasion attacks, black-box modeling, TCP**

1. Introduction

"Well, let's just say I know a little girl who can walk through walls." – Charles Xavier [1]

Stateful firewalls (SFWs) play a critical role in securing network infrastructure across enterprise [2], industrial [3], and cloud networks [4], [5]. Unlike simple access control lists, SFWs track individual TCP connections to determine which packets are forwarded. For instance, a canonical policy is to allow packets only on connections that have been previously established by hosts inside the network.

Semantic errors in the SFW implementation open the door for external attackers to evade the SFW and attack internal hosts. For instance, some SFW implementations incorrectly process specific packet sequences (e.g., a SYN followed by an ACK followed by a RST) and forward malicious traffic to the internal network even when a connection has not been established by an internal host; thus evading the SFW. More generally, there could be many such semantic errors in an SFW implementation that introduce vulnerabilities. Such errors fall outside the scope of prior work on firewall rule checking [6], [7], [8] as evasion can occur even if rules are correctly configured.

Given the diversity of SFW implementations and deployments, operators need tools to automatically uncover semantic vulnerabilities, satisfying two key requirements:

- *Generalizable across black-box SFWs implementations:* Many SFW implementations are proprietary and acquired as closed packages (e.g., hardware boxes, closed VMs or containers). There is no access to code or internal details. Thus, we can only assume black-box access to firewalls and prior work on firewall code analysis (e.g., [9], [10]) does not apply.
- *Generalizable across deployments:* The configurations and capabilities of various network hosts impact the security posture of the deployment. For e.g., attackers controlling an internal host could use it to spoof packets, or a victim stack with a legacy OS could accept malformed TCP packet sequences. Thus, we need a framework to handle the diversity of deployments and attacker capabilities, in reasoning about potential SFW evasion opportunities. This rules out prior work on protocol fuzzers (e.g., [11], [12]), automata learning (e.g., [13], [14]), and censorship circumvention (e.g., [15], [16]) that assume a fixed deployment.

Given such a framework, operators can use the findings to work with vendors and take appropriate measures. For instance, they can update firewall versions or configurations to quarantine devices, block specific packet sequences, or drop spoofed packets.

Building such a framework entails addressing fundamental state-space explosion challenges [17] across multiple dimensions including the complexity of SFW implementations and the large space of input packet sequences. Together, with the diversity of attack capabilities and victim stack implementations, uncovering evasion attacks is challenging. We elaborate on this in §2.

To tackle these challenges, we design *Pryde*[1]: a modular model-guided workflow to generate evasion attacks. We explicitly decouple SFW- and deployment-specific sources of state-space complexity to reduce the search space. Given the diversity of an attacker's downstream goals (e.g., lateral movement, remote code execution), *Pryde* supports the flexibility to define attack success criteria such as sending a

. * These authors contributed equally to this work.

1. The name *Pryde* is inspired by the Marvel superhero Kitty Pryde who can walk through walls [1].

malicious payload to a victim vs. receiving an ACK to sent malicious data. Our design contributions include:

- A scalable black-box model inference approach to reason about the behavior of stateful firewalls that considers non-compliant TCP packet sequences relevant for evasion.
- A practical and extensible model-guided workflow to specify deployment scenarios and SFW models in a model checker [18]. We design compact SMT representations and refinement heuristics to efficiently uncover semantically-distinct attacks.

Findings: We evaluate *Pryde* against 4 popular SFWs[2] (anonymized as FW-1 to FW-4) and 4 victim stacks. Our choice of SFWs includes 3 commercial-grade vendors (2 available from the AWS EC2 marketplace [19]) and a popular open-source vendor. Our choice of victim stacks (§6) is representative of real-world vulnerabilities (e.g., [20], [21]). Our key findings are:

- *Pryde* generates >6,000 successful and unique evasion attacks, 2-3 orders of magnitude higher than black-box fuzzing and censorship circumvention algorithms [15] (Finding **1**).
- *Pryde*'s generalizable workflow generates successful attacks customized to a firewall and deployment setting, as attacks don't translate well across different settings (Finding 7). *Pryde* finds attacks against diverse firewalls and victims, with and without insider threats (details in §3.1), and against multiple attacker success criteria (Figure 3).
- *Pryde* helped uncover many counter-intuitive and subtle firewall-specific behaviors leading to successful evasion (as well as resilience). For instance, we discovered that FW-3 forwards ACKs before a TCP handshake is even initiated; a behavior that led to subtle attack patterns (Finding **4**). In another example, FW-1 actively spoofs RSTs in response to unwanted packets, to thwart many attacks (Finding **3**).

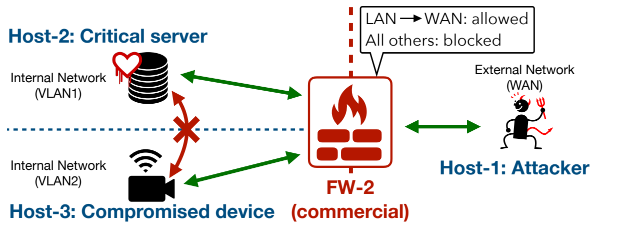
Vendor disclosure: We have disclosed our findings through multiple iterations of *Pryde* to all 4 vendors. We shared video demos of the attacks and scripts for reproducing them. All vendors acknowledged our initial report. FW-1 and FW-4 vendors asked to replicate our study on newer versions and we found successful attacks against them as well. FW-3 vendors mentioned that later firewall releases enabled stricter checks by default, which might thwart our attacks[3]. However, we found successful attacks even with the checks enabled (more details in Appendix A).

2. Background and motivation

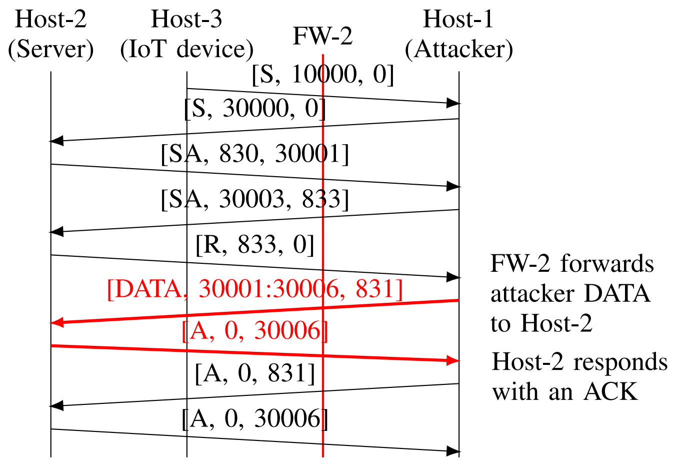
In this section, we provide background on stateful network firewalls and their typical deployment. Then, we motivate the need for uncovering evasion and discuss why prior work is ineffective in our problem setting.

2. Due to the sensitive nature of the findings and vendor conversations, we anonymize these vendor names.

3. This check was enabled after our first disclosure in Sep 2020.



(a) Network setup showing an external attacker (Host-1), a stateful firewall (FW-2), a critical server (Host-2), and a compromised IoT device (Host-3). Host-3 cannot observe or talk to Host-2.



(b) An attack uncovered by *Pryde* shown as a timing diagram. This diagram shows an exchange of packets between different hosts across FW-2, with the format [TCP flags, seq, ack].

Figure 1: Motivating attack scenario showing that an internal host can accept malicious data sent by an external attacker even with a correctly configured firewall

2.1. Background on stateful firewalls

Stateful firewalls (SFWs) are critical to blocking unwanted access to protected networks from untrusted network segments. In this work, we consider an SFW operating at the network layer. An SFW decides to drop or forward a network packet based on the 5-tuple defining a connection, *(src-ip, src-port, dst-ip, dst-port, protocol)*, and the configured rules. Typically, an SFW is configured with rules of the form: {*interface, 5-tuple*}→*action* where *interface* refers to the network interface where incoming TCP packets are captured. The SFW tracks TCP connection states and uses the configured rules to determine the actions to take.

A common configuration is to drop packets from a less trusted network segment that do not belong to a connection already initiated and established by an internal host [22]. For instance, we can allow internal hosts to initiate a connection to `cmu.edu` and also allow the return "data" packets from `cmu.edu` to arrive at the host that initiated the connection. However, other unsolicited TCP packets from `cmu.edu` should be dropped. This logic is the "stateful" part; where the SFW needs to check if a packet belongs to an existing connection. To this end, the SFW logic considers

TCP/IP header fields such as source/destination IPs and ports, TCP flags, and sequence/acknowledgement numbers. For instance, it may check whether a given TCP packet has seq/ack numbers set within an expected window. Based on this stateful logic, the packet will be forwarded or dropped.

2.2. Motivating scenario

We now describe a motivating scenario using a subtle attack we uncovered using *Pryde*. The attack demonstrates how an attacker can evade SFWs and attack an internal host even if best practices are followed.

Consider the setting in Figure 1a with 3 hosts and a commercial firewall FW-2 (anonymized). Host-1 is an external attacker. Host-2 is an internal server behind FW-2 that is running Ubuntu 11.04. Host-3 is a low-security IoT device that has been compromised out of band [23] (details in §3.1). The network operator follows best practices and separates critical servers, like Host-2, from low-security hosts, like Host-3, using network segmentation (e.g., VLANs). Thus, Host-3 cannot observe or communicate with Host-2. To protect internal hosts like Host-2, FW-2's rules are set up to only allow TCP connections from internal hosts and drop all other packets (e.g., from Host-1).

Figure 1b shows an evasion attack uncovered by *Pryde*. Each vertical line represents a host. An arrow between 2 hosts is a TCP packet represented in the format [flags, seq, ack]. The attack starts with the compromised Host-3 spoofing a SYN packet as Host-2's and sending it to Host-1. The spoofed SYN punches a hole in FW-2, allowing Host-1 to send packets to Host-2. Despite Host-2's RST, Host-1's packets evade FW-2's protections and ultimately attack Host-2.

Figure 1b shows one attack using a specific SFW and network setting. More generally, we want to help network operators reason about evasion scenarios across different SFWs, victims, and attacker capabilities. Taken together, we need *generalizability* along two key dimensions:

- *Black-box SFWs implementations:* Different SFW vendors have critical differences in their stateful connection tracking logic that can, in turn, lead to subtle and distinct evasion attacks.
- *Deployment specifics:* Attackers may have different capabilities such as sending arbitrary packets to the firewall or colluding with a weak insider that is capable of spoofing. Further, some victim hosts may be more accepting of malformed packet sequences than others.

2.3. Related work and limitations

Prior work on firewall testing, censorship circumvention, code analysis, protocol fuzzing, and automata learning is insufficient in meeting our requirements (Table 1).

Firewall rule testing: Many tools test the correctness of SFW configurations (e.g., [6], [7], [8]). However, they only focus on verifying the rule-sets and cannot uncover semantic attacks on SFW implementations. They cannot uncover the attack in Figure 1 as FW-2's rules are configured correctly.

	Generalizable across blackbox SFWs	Generalizable across deployments
Firewall rule testing [6], [7]	◐	○
Censorship circumvention [15], [16]	●	○
Source code analysis [9], [10]	◑	○
Protocol fuzzers [11], [12]	●	○
Automata learning [14], [26], [27], [13]	●	○
Pryde	●	●

◐: Generalizable but require whitebox access to SFW rules
◑: Generalizable but require whitebox access to SFW/end-host code

TABLE 1: Comparing Pryde against prior work in satisfying two key requirements.

Censorship circumvention: Censorship circumvention tools can create packets to punch "holes" in censors [15], [16]. While successful against censors, there are two fundamental issues. First, they only uncover shallow evasion sequences and do not systematically explore the attack space to achieve semantic coverage. For instance, the attack sequence given in Figure 1b exercises different states of FW-2 to achieve evasion. Second, these tools cannot be customized to diverse deployment settings with many dimensions such as SFW, victim implementation, and attacker capabilities. Finding **1** shows how state-of-the-art tools [15] are ineffective in our problem setting.

Source code analysis: Many tools use the end host's source code or instrumented binary to uncover evasion attacks against the end host itself [9], [10] or a NIDS [24]. Analyzing the SFW's code to generate evasion attacks against it is infeasible in a black-box setting[4]. While SymTCP [24] does not access the NIDS' code, it and other works do not account for different deployments (such as Host-3 being an insider threat).

Protocol fuzzers: Protocol fuzzers [11], [12], [25] perform stateful analysis on network protocols to find semantic errors. However, they only consider simple client-server models and do not model deployments with multiple network segments or SFWs that forward and drop packets. These fuzzers cannot analyze deployments in the presence of an insider (e.g., Host-3 in Figure 1a) or model packets entering an SFW from multiple interfaces.

Automata learning and model checking: There are prior efforts to analyze protocol implementations [14], [26] and check them for non-compliance [27], [13]. For e.g., BLEDiff [13] focuses on differences in BLE protocol implementations using black-box model inference techniques. These works focus on an orthogonal problem and do not capture the behavior of a system in the presence of attack packets. They also do not model deployments (such as one with insiders) and thus, cannot be used to generate deployment-aware evasion attacks. For e.g., given the setup in Figure 1a, these tools would not be able to reason about leveraging the compromised Host-3 to evade FW-2.

4. 3 out of the 4 firewalls we test are closed source.

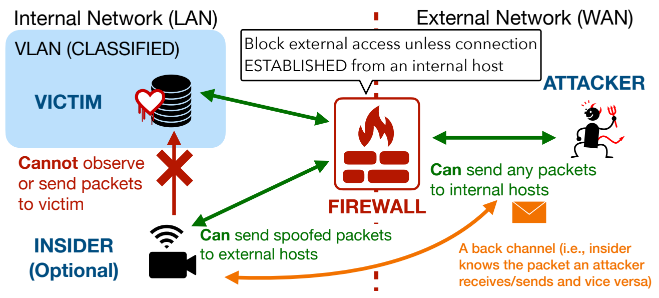

Internal Network (LAN)
External Network (WAN)
VLAN (CLASSIFIED)
Block external access unless connection ESTABLISHED from an internal host
VICTIM
ATTACKER
Cannot observe or send packets to victim
FIREWALL
Can send any packets to internal hosts
INSIDER (Optional)
Can send spoofed packets to external hosts
A back channel (i.e., insider knows the packet an attacker receives/sends and vice versa)

Figure 2: Deployment model with the attacker, victim, stateful firewall, and (optional) insider in a network.

3. Overview

We begin by scoping our problem and discussing the design space of options. We show why strawman solutions are intractable and give an overview of our design choices. We then discuss *Pryde*'s end-to-end workflow.

3.1. Problem Setup and Threat Model

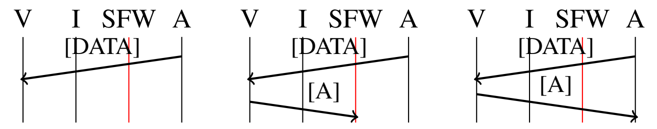
Figure 2 generalizes the problem setup from Figure 1a. We consider an SFW that aims to protect a victim from an attacker. The victim can implement any network stack. The attacker and the victim interact in the (physical or virtual) network by sending TCP packets. We also consider an optional compromised internal host (insider).

Attacker goal: Given a deployment and an SFW configured to drop traffic on unsolicited connections from an untrusted host, the attacker's goal is to evade the SFW and send a DATA packet (TCP packet with a malicious payload) to the seemingly unreachable victim. The notion of the victim accepting a DATA packet is dependent on the downstream goal. If the attacker's goal is to only evade the firewall, any DATA packet reaching the victim would count as a successful attack even if the victim responds with a RST. If the attacker has a stronger goal of attacking the victim, say using a buffer overflow or Heartbleed [28], acceptance means that the victim needs to respond to the received DATA packet with an ACK.

Pryde is flexible and admits different attack success criteria. As a concrete starting point, we consider 3 success criteria with progressively stringent requirements (Figure 3):

- **(S1) Victim receives DATA:** Firewall forwards DATA from the attacker to the victim.
- **(S2) Victim sends ACK:** Firewall forwards attacker DATA to the victim and the victim responds with an ACK.
- **(S3) Attacker receives ACK:** Firewall forwards attacker DATA to the victim, the victim responds with an ACK and the firewall forwards this ACK to the attacker.

The attacks that *Pryde* uncovers can be building blocks to enable future attacks such as lateral movement, remote code execution, and data exfiltration. These tasks are outside the scope of this paper. Given that S1 is a necessary step for S2 and S3 (which are stricter criteria), we primarily focus on S2 and S3 in our evaluation (§6).



S1: Victim receives DATA **S2: Victim sends ACK** **S3: Attacker receives ACK**

Figure 3: *Pryde* supports flexible attack success criteria.

Attacker capabilities and constraints: The attacker can craft and send *any* TCP packets to internal hosts (Figure 2). However, the SFW, based on its own logic, determines the action to take (e.g., drop, forward, send another packet) on the packet.

We assume that the attacker does not know the SFW rules but knows the SFW vendor and version. This can be obtained by fingerprinting techniques from prior work such as banner grabbing [29]. The attacker has no visibility into the SFW implementation or code but has offline black-box access, e.g. by obtaining a virtual SFW appliance [19]. The attacker does not know the victim's implementation and cannot control or observe the victim's actions.

Victim: The victim is an end host or IoT device running an OS that could be susceptible to attacks. For instance, there are multiple CVEs [30] for the Linux 2.6 kernel used in many IoT devices (e.g., smart TV [20]) and networking hardware (e.g., Cisco, Asus, Netgear routers [21]). In our evaluation (§6), we consider 4 diverse victim stacks popular in consumer IoT devices [31], [32].

Insider: Optionally, the attacker may collude with a weak *insider* in the internal network, that cannot directly attack the victim. The insider is in an isolated network segment (e.g., a separate VLAN). It cannot observe the victim's behavior and communication with the victim is explicitly blocked via network segmentation. It has limited capabilities such as spoofing the source of TCP packets. Spoofing is a reasonable capability as deployments may not enable source address validation by default[5] even across VLANs [33], [34].

A concrete example of an insider is a compromised IoT device residing in a different network segment [35], [36] from the victim. Attack campaigns like Mirai [37] and multiple reports on spear-phishing [38], [39], [40], [41] and breaches in IoT networks [42], [23], [43], [44] show the prevalence of insider threats.

3.2. High-level ideas

We present the space of design options (Figure 4) and our high-level ideas here. We discuss detailed design choices in subsequent sections (§4 - §5.2).

Deployment aware vs. agnostic: Given an SFW, there are two approaches to generate evasion attacks. A *deployment-agnostic* approach focuses only on the behavior of the SFW to try and generate attacks, ignoring the behavior of

5. FW-3 vendors acknowledged that this check was disabled by default and would remain disabled for fear of breaking existing deployments!

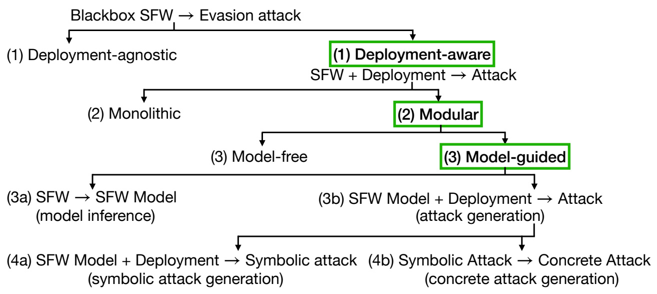

Blackbox SFW → Evasion attack
(1) Deployment-agnostic
(1) Deployment-aware
SFW + Deployment → Attack
(2) Monolithic
(2) Modular
(3) Model-free
(3) Model-guided
(3a) SFW → SFW Model (model inference)
(3b) SFW Model + Deployment → Attack (attack generation)
(4a) SFW Model + Deployment → Symbolic attack (symbolic attack generation)
(4b) Symbolic Attack → Concrete Attack (concrete attack generation)

Figure 4: Design space taxonomy and *Pryde*'s design choices highlighted with green boxes in this taxonomy.

other hosts in the deployment. Strawman examples of this include fuzzing [11], [12], [25] and prior work on censorship circumvention [15], [16]. For instance, one could use a Finite State Machine (FSM) model of an SFW and attempt to craft attacks that drive the SFW to a vulnerable state. Censorship circumvention works use genetic algorithms to craft strategies to evade censors. However, these approaches cannot reason about the behavior of other hosts or constraints imposed by the network. They are ineffective at generating attacks that can evade an SFW and attack an internal host (refer to Finding **1**). Thus, we argue for a **deployment-aware** approach in designing *Pryde*.

Modular vs. monolithic: Within deployment-aware approaches, a monolithic approach takes a bird's eye view and reasons about the entire deployment (with all hosts and network constraints) as a whole. An example of such an approach is stateless deployment-aware fuzzing (Finding **1** compares *Pryde* to this). A monolithic approach is neither scalable nor extensible as assumptions about the deployment and victim implementations may change over time. For instance, introducing a second insider or changing the attack success criteria in Figure 2 would require us to re-run the entire workflow for each possible scenario. Instead, we use a **modular** approach where we decouple reasoning about the SFW, from the deployment setting and attacker capabilities. §3.4 discusses how this modular workflow allows *Pryde* to generalize to different SFWs, deployments, and success criteria.

Model guided vs. model free: Given an SFW and a deployment, there are two natural approaches to generate evasion attacks – model-guided and model-free. Model-free approaches leverage techniques like fuzzing or genetic algorithms and do not learn an explicit model of the hosts in a deployment. Evasion attacks require triggering specific state transitions in the SFW and thus require carefully crafted TCP packet sequences. We argue for a **model-guided approach** where we learn a sufficiently-precise model of the SFW's stateful semantics (§4). We reason about this in conjunction with a model of the deployment to uncover evasion attacks that exploit deep semantic errors (§5).

3.3. Pryde's workflow

Figure 5 provides an overview of *Pryde*'s modular workflow with three steps.

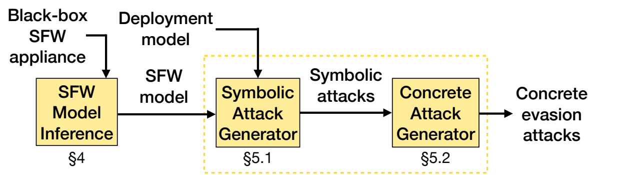


Figure 5: High-level workflow of *Pryde*.

SFW Model Inference (§4): Given a black-box SFW appliance, this module outputs an FSM model. Our key contribution here over prior work Alembic [45] is creating a model inference workflow targeted to evasion scenarios such as reasoning about out-of-window TCP packets.

We decouple *Pryde*'s attack generation into the following two modules to ensure scalability and extensibility.

Symbolic Attack Generator (§5.1): This module reasons about the interactions of the SFW and other hosts in the network to output evasion attacks. The key contribution here is in succinctly encoding these interactions using SMT constraints atop Z3 [18]. We delay binding of TCP seq/ack fields to the Concrete Attack Generator. This allows us to reason about the semantics of the SFW and the deployment without considering the large space of seq/ack numbers, thus making analysis scalable. Each outputted symbolic attack is a sequence of a sender (s_i) and a symbolic packet (σ_i).

Concrete Attack Generator (§5.2): This module binds concrete seq/ack values to each symbolic packet and outputs concrete TCP packet sequences. It uses the generated symbolic attacks and an ensemble of *strategies* to synthesize evasion attacks with concrete seq/ack numbers (i.e., σ_i with concrete seq/ack values). To apply strategies, we devise simple heuristics to reason about the relationships between packets' seq/ack numbers, which include both TCP-compliant and non-TCP-compliant semantics.

Our modular approach is *extensible* to future deployments and reduces the need for running the entire workflow for a new scenario. Consider an operator who wants to generate evasion attacks for a given SFW in n different deployments. Instead of running the entire workflow n times, the operator can run Model Inference once and reuse the inferred SFW model for each of the n deployments. Similarly, if an operator wants to try a new *strategy* (defined in §5.2) in the Concrete Attack Generator, they do not have to re-run Model Inference or the Symbolic Attack Generator.

3.4. How an operator would use *Pryde*

To use *Pryde*, an operator needs to set up the firewall appliance and provide these inputs: (1) network configuration; (2) deployment model; (3) attacker goal; and (4) (optional) approximate victim model. These inputs are typically already available or easy to obtain.

Figure 6 shows examples of these inputs and where they are used in *Pryde*'s workflow. (1) The network configuration specifies the IPs of the firewall interfaces and various subnets. (2) The deployment model specifies the hosts present in the network, their capabilities (e.g., spoofing), and their

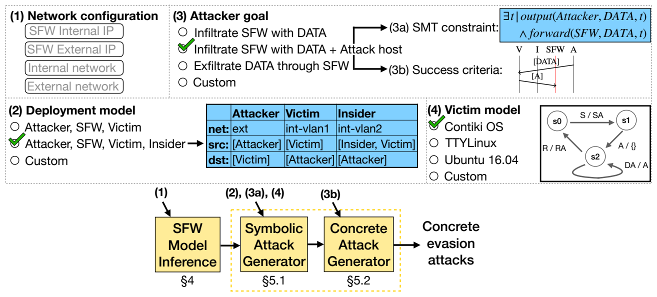

(1) Network configuration
SFW Internal IP
SFW External IP
Internal network
External network
(3) Attacker goal
Infiltrate SFW with DATA
✓ Infiltrate SFW with DATA + Attack host
Exfiltrate DATA through SFW
Custom
(3a) SMT constraint:
$\exists t \mid output(Attacker, DATA, t) \wedge forward(SFW, DATA, t)$
(3b) Success criteria:
V I SFW A
[DATA]
[A]
(2) Deployment model
Attacker, SFW, Victim
✓ Attacker, SFW, Victim, Insider
Custom
Attacker | Victim | Insider
net: ext | int-vlan1 | int-vlan2
src: [Attacker] | [Victim] | [Insider, Victim]
dst: [Victim] | [Attacker] | [Attacker]
(4) Victim model
✓ Contiki OS
TTYLinux
Ubuntu 16.04
Custom
s0
s1
s2
S / SA
A / {}
R / RA
DA / A
(1)
(2), (3a), (4)
(3b)
SFW Model Inference
§4
Symbolic Attack Generator
§5.1
Concrete Attack Generator
§5.2
Concrete evasion attacks

Figure 6: Interface to configure and run *Pryde*.

constraints (e.g., can the insider talk to the victim). (3) The attacker goal has 2 parts: (3a) an SMT constraint used during symbolic attack generation as a goal, and (3b) an observable success criteria used to classify attacks as successful when validating them on a live testbed. For instance, the SMT constraint could represent the goal that the attacker must send a DATA packet at timestep t which is forwarded by the firewall. The observable success criteria could be one from Figure 3. (4) The victim model is an approximate low-fidelity Mealy machine[6], used to guide the exploration of the attack space. If unspecified, *Pryde* uses a default victim model which accepts all packets.

Pryde's workflow is generalizable and does not depend on a specific firewall or deployment. Given a firewall and the attacker's goal, *Pryde* runs automatically and outputs attacks that evade the given firewall. By changing the inputs, an operator can use *Pryde* to reason about different deployments. An operator can reason about a different firewall binary or configuration using the Model Inference module (§4) They can reason about evasion with multiple insiders by modifying the deployment model and running the Symbolic Attack Generator (§ 5.1) They can also use *Pryde* to reason about other attacker goals, such as data exfiltration, by changing the SMT constraint and observable success criteria.

Pryde's modular approach reduces the need for running the entire workflow for a new scenario. Consider an operator who wants to generate evasion attacks for a given SFW in n different deployments. Instead of running the entire workflow n times, the operator can run Model Inference once and re-use the inferred SFW model for each of the n deployments. Similarly, if an operator wants to try a new *strategy* (defined in §5.2) in the Concrete Attack Generator, they do not have to re-run Model Inference or the Symbolic Attack Generator. We discuss the details of *Pryde*'s modular generalizable design in §4 and §5.

4. Evasion-Aware Model Inference

Given the SFW binary, our aim is to generate a black-box model to reason about evasion scenarios. Prior work on black-box model inference [45] only considers benign TCP-compliant scenarios and cannot reason about SFW behavior with non-TCP-compliant packets, which are crucial for evasion. If we only considered TCP compliance, we would not

6. A Mealy machine is an FSM where the output depends on the input and the current FSM state

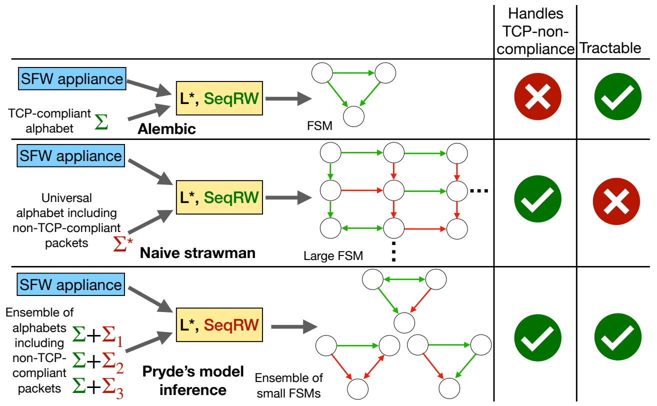


Figure 7: Overview of prior work *Alembic* [45], a naive strawman, and *Pryde*'s model inference module.

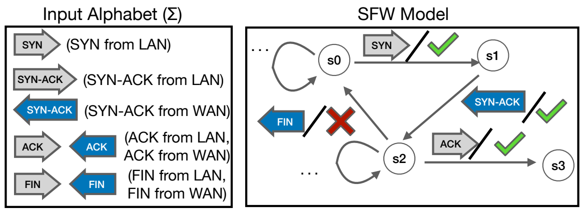


Figure 8: Example of an input alphabet and SFW model.

uncover the type of attack described in §2.2, where a SYN packet spoofed by the insider and a SYN-ACK sent by the attacker led to the attacker sending DATA to the victim.

We begin by providing background on prior work and discuss why strawman extensions could make model inference intractable. We then present our key insights to solve this problem. Figure 7 provides an overview of prior work *Alembic* [45], a strawman, and our approach.

4.1. Prior work and limitations

An SFW's behavior can be represented as an FSM [46], [45]. Prior work *Alembic* [45] extends classical black-box FSM inference techniques (L* [47]) to model stateful network functions. *Alembic* takes 2 inputs: an SFW binary and an input alphabet (Σ) describing packets of interest (e.g., SYN from host A to B). *Alembic*'s output is a Mealy machine where transitions between states are based on symbolic representations of TCP packets.

Figure 8 shows an example input alphabet (Σ) and an inferred SFW model. Σ specifies packets of interest used to infer the model – SYN from LAN to WAN, and SYN-ACK, ACK and FIN in both directions. The directions of the arrows represent packet source/destination networks. The SFW model represents its behavior when it receives a packet of interest. For e.g., if the SFW is in state s_0 and receives a SYN from the LAN, it forwards the SYN (denoted by a green check mark) and transitions to state s_1. If the SFW is in state s_2 and receives a FIN from the WAN, it drops the packet (denoted by a red X) and transitions to state s_0.

Alembic uses the L* algorithm to adaptively construct packet sequences of varying lengths using the symbols in the input alphabet (Σ). Packet sequences are played against the SFW and the SFW's output is used to update the estimate of its model. This process repeats until the model converges or a specified timeout is reached.

Alembic considers two types of input alphabets: (1) $\Sigma^{\text{Setup}}_{\text{Alembic}}$ that includes connection setup packets (SYN from LAN, SYN-ACK from WAN, ACK from LAN and WAN, DATA from WAN); and (2) $\Sigma^{\text{Teardown}}_{\text{Alembic}}$ which includes connection teardown packets; RST, RST-ACK, FIN, FIN-ACK in both directions.[7] We define $\Sigma_{\text{Alembic}} = \Sigma^{\text{Setup}}_{\text{Alembic}} \cup \Sigma^{\text{Teardown}}_{\text{Alembic}}$. *Alembic* needs to convert a sequence of symbolic packets (from Σ^*) to TCP packets, which entails setting seq/ack numbers for each packet. Exploring the entire 2^{64} space of seq/ack numbers makes model inference intractable. *Alembic* restricts these values to adhere to TCP semantics. Consider a candidate packet sequence of length 2 generated by L*: SYN from LAN to WAN and SYN-ACK from WAN to LAN. If the SFW forwards the SYN to the WAN, the WAN host sets its SYN-ACK's ack based on the received SYN's seq. More generally, the seq/ack of packets may be dynamically modified based on previous packets observed by a host. We refer to this dynamic modification of seq/ack values as $\text{SeqRW}(\text{seq} = \text{X})$ (short for Sequence Re-Writing) where X is the initial seq value.

Limitations in our context: *Alembic* [45] was designed to handle non-adversarial TCP-compliant traffic. Our ablation study in Finding **5** shows that *Alembic*'s models are ineffective at generating evasion attacks. This is due to two fundamental limitations:

- *Input alphabet*: *Alembic*'s alphabet set (Σ_{Alembic}) only includes TCP-compliant packets. This alphabet is not expressive enough to consider evasion which requires non-TCP-compliant packets; such as out-of-window seq/ack numbers or packets that deviate from the TCP handshake protocol. To reason about evasion, we need to model both TCP-compliant and non-TCP-compliant traffic and their interactions. This creates new scalability challenges as it increases the packet header space and the space of packet sequences to consider for model inference.
- *Rewriting logic*: *Alembic*'s SeqRW assumes TCP compliance. Reusing this logic for non-TCP-compliant packets defeats the purpose of considering these packets. That is, even with an input alphabet expressive enough to capture evasion scenarios, this logic would not generate an SFW model that leads to evasion attacks (such as the example in §2.2). Hence, we need to extend the SeqRW logic while still avoiding state-space explosion caused due to the large space of concrete seq/ack values.

Next, we discuss our contributions to (1) extend the input alphabet to handle non-TCP-compliant traffic (§4.2); and (2) extend the rewriting logic to be more flexible (§4.3).

4.2. Evasion-aware input alphabet selection

A strawman solution is to consider a universal alphabet containing all possible combinations of seq/ack, TCP flags,

7. These are referred to as `correct-seq` in the *Alembic* paper [45].

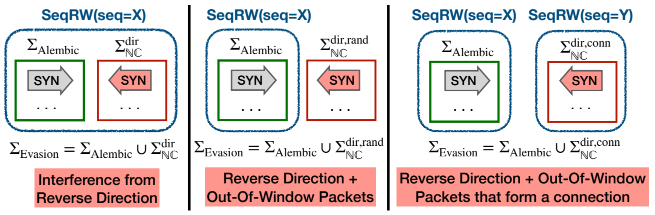

SeqRW(seq=X)
Σ_{Alembic}
$\Sigma^{\text{dir}}_{\mathbb{NC}}$
SYN
SYN
$\Sigma_{\text{Evasion}} = \Sigma_{\text{Alembic}} \cup \Sigma^{\text{dir}}_{\mathbb{NC}}$
Interference from Reverse Direction
SeqRW(seq=X)
Σ_{Alembic}
$\Sigma^{\text{dir,rand}}_{\mathbb{NC}}$
SYN
SYN
$\Sigma_{\text{Evasion}} = \Sigma_{\text{Alembic}} \cup \Sigma^{\text{dir,rand}}_{\mathbb{NC}}$
Reverse Direction + Out-Of-Window Packets
SeqRW(seq=X)
SeqRW(seq=Y)
Σ_{Alembic}
$\Sigma^{\text{dir,conn}}_{\mathbb{NC}}$
SYN
SYN
$\Sigma_{\text{Evasion}} = \Sigma_{\text{Alembic}} \cup \Sigma^{\text{dir,conn}}_{\mathbb{NC}}$
Reverse Direction + Out-Of-Window Packets that form a connection

Figure 9: Rewriting logic to handle Σ_{Evasion} with non-compliant set ($\Sigma_{\mathbb{NC}}$).

and IP/port values. Unfortunately, this results in an explosion in the alphabet size and makes model inference intractable.

Key insight: Rather than learning a "large" model for all possible non-compliant scenarios, our insight is to decompose the problem and design an *ensemble* of *independent* Σs where each Σ relates to a potential type of non-compliance. Given this ensemble of Σs, we can infer an *ensemble of SFW models*, where each model can be used to guide the generation of evasion attacks. Our approach naturally lends itself to an *extensible* way to reason about evasion. We introduce Σ_{Evasion} to consider the potential interference of TCP states from non-compliant input alphabets. This involves reasoning about the interaction between the TCP-compliant packets, Σ_{Alembic}, and non-TCP-compliant packets, $\Sigma_{\mathbb{NC}}$; i.e., $\Sigma_{\text{Evasion}} = \Sigma_{\text{Alembic}} \cup \Sigma_{\mathbb{NC}}$.

Scenarios for TCP non-compliance: We discuss these input alphabet templates below:

1) *Flipped direction* ($\Sigma^{\text{dir}}_{\mathbb{NC}}$): These TCP 3-way handshake packets have flipped directions only (e.g., SYN from the WAN). The seq and ack numbers are *not* out-of-window.
2) *Flipped direction and random seq/ack numbers* ($\Sigma^{\text{dir,rand}}_{\mathbb{NC}}$): These packets have flipped directions and are out of window (with seq/ack randomly initialized).
3) *Flipped direction and in-window amongst themselves* ($\Sigma^{\text{dir,conn}}_{\mathbb{NC}}$): This case is similar to the previous case except the out-of-window packets form a TCP connection amongst themselves.
4) *Out-of-window only* ($\Sigma^{\text{rand}}_{\mathbb{NC}}$): These packets have the same direction but are out-of-window (seq/ack randomly initialized).[8]

For Σ_{Alembic}, we consider both $\Sigma^{\text{Setup}}_{\text{Alembic}}$ and $\Sigma^{\text{Setup}}_{\text{Alembic}} \cup \Sigma^{\text{Teardown}}_{\text{Alembic}}$. For $\Sigma_{\mathbb{NC}}$, we consider each of the 4 templates given above and a null set (as a baseline). We combine each of the 2 Σ_{Alembic} with each of the 5 $\Sigma_{\mathbb{NC}}$ to get 10 input alphabets for each SFW. *Alembic* assumes that the underlying model is deterministic and fails to converge if this is not the case.[9] While using an ensemble of independent Σs may result in some missed evasion opportunities compared to using a universal Σ, this is a pragmatic trade-off. We sacrifice completeness for soundness and tractability, similar to many large-scale testing and verification systems [17].

8. Alembic considers this alphabet too (referred to as `combined-seq`) but this alphabet does not yield any attacks. Refer to Finding **5**.
9. We do not consider alphabets for which convergence is not reached.

4.3. Evasion-aware rewriting logic

A strawman solution is to reuse *Alembic*'s seq/ack rewriting logic, which rewrites all seq/ack numbers in Σ_{Alembic} to adhere to exact TCP semantics. However, this prevents us from finding attacks that involve the firewall and victim accepting out-of-window packets. Another strawman solution is to allow all possible values. However, this makes model inference intractable.

Key insight: Our insight is to exploit the semantic structure per non-compliant Σ and customize the rewriting logic independently for each group (Figure 9). For e.g., in $\Sigma^{\text{dir}}_{\mathbb{NC}}$, seq/ack of all packets in Σ_{Alembic} and $\Sigma^{\text{dir}}_{\mathbb{NC}}$ are rewritten to adhere to TCP semantics (first column of Figure 9). For $\Sigma^{\text{dir,rand}}_{\mathbb{NC}}$ (reverse direction and random seq/ack), packets in Σ_{Alembic} are subject to $\text{SeqRW}(X)$ and packets in $\Sigma^{\text{dir,rand}}_{\mathbb{NC}}$ are initialized randomly and not re-written.

Our modified implementation of the *Alembic* algorithm tracks which alphabet each packet belongs to and decides the appropriate rewriting logic to be applied. Our extensions to the input alphabets and SeqRW allow *Pryde* to generalize to diverse SFWs, SFW configurations, and evasion opportunities. For instance, an operator can add or modify the input alphabets to capture the behavior of the SFW under a different setting. Finding **5** evaluates the importance of *Pryde*'s extensions in discovering attacks.

5. Extensible Evasion Attack Generator

Given the inferred SFW model from §4, the goal is to generate evasion attacks where the attacker evades the SFW as defined by the user-specified success criteria (§3.1). Attack generation can be done in a deployment-agnostic or deployment-aware manner (§3.2).

Deployment-agnostic approaches focus solely on the SFW behavior. A strawman example is model-guided fuzzing, which generates packet sequences to force the SFW to a vulnerable state (e.g., states where the SFW forwards DATA inadvertently). This approach ignores the capabilities and constraints of other hosts in the deployment. We show the ineffectiveness of such an approach in Finding **1** and discuss its role in an ablation study of *Pryde* in Finding **5**.

We design *Pryde*'s attack generation framework to be deployment aware. A strawman deployment-aware workflow would search the large space of interactions amongst various hosts in the deployment and directly generate concrete TCP attacks. However, this approach does not scale. Therefore, we decouple attack generation (refer §3.3) into 2 stages: (1) the *Symbolic Attack Generator* (§5.1) reasons about the interactions of the hosts on the network to generate symbolic attacks; and (2) the *Concrete Attack Generator* (§5.2) uses these symbolic attacks to synthesize concrete evasion attacks.

5.1. Symbolic Attack Generator

Strawman solution: A strawman solution is to naively encode the SFW and the deployment model into a model checker (like Z3 [18]) and output all paths that satisfy the attack requirement. However, this method outputs many semantically-identical packet sequences that provide no additional value to the operator using *Pryde*.

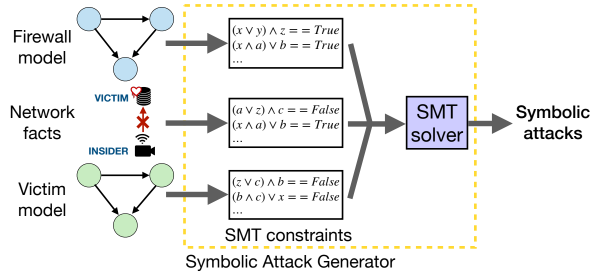

Firewall model
Network facts
VICTIM
INSIDER
Victim model
$(x \vee y) \wedge z == True$
$(x \wedge a) \vee b == True$
...
$(a \vee z) \wedge c == False$
$(x \wedge a) \vee b == True$
...
$(z \vee c) \wedge b == False$
$(b \wedge c) \vee x == False$
...
SMT constraints
SMT solver
Symbolic attacks
Symbolic Attack Generator

Figure 10: Overview of Symbolic Attack Generator.

Key insights: We use two main ideas (Figure 10) to tackle this problem. Firstly, we encode the SFW and deployment model using SMT constraints. This allows us to achieve semantic coverage over the space of all possible attacks using an SMT solver. Secondly, we introduce constraints to make SMT solving tractable and to identify semantically-distinct symbolic attacks. Our contribution is a practical encoding of the SFW and deployment model to identify semantically-distinct symbolic attacks using an SMT solver. While we represent these using Z3 [18] (§6), these could be extended to other model checking tools.

Inputs and outputs: The input is an inferred SFW model and a deployment model. The deployment model includes the attacker model with its capabilities and constraints (such as an insider), an optional victim model representing its network behavior, and a model of the network representing where the hosts are present in the network (e.g., the attacker in the external network, the insider in VLAN 2). We use an approximate victim model to guide the search. However, we use a real victim appliance while testing our attacks (§6). The output is a list of symbolic attacks applicable to the given deployment and victim models. Each symbolic attack is an ordered sequence of symbolic *located packets* [48], which we define below.

Transitions in the SFW model are defined using *located packets* [48]. Since considering all possible fields of the TCP/IP headers is intractable, we consider a subset of these fields and refer to each such packet as σ. σ is defined by the following tuple: $(\text{interface, src-ip, src-port, dst-ip, dst-port, flag, data, prefix, seq, ack})$. interface is the incoming interface (LAN or WAN) where the packet is received at the SFW. data is a Boolean indicating the presence of a payload. prefix indicates whether the seq/ack fields were rewritten during model inference to follow the TCP semantics (Σ_{Alembic} or $\Sigma_{\mathbb{NC}}$). For now, seq/ack fields in the packet are unused and become relevant in the Concrete Attack Generator (§5.2) when binding seq/ack to concrete integers. Input packets to the victim model are also defined by this tuple (σ) but seq/ack-related fields (prefix, seq, ack) are unused. This is a conscious choice given that the attacker may manually approximate victim models for attack generation and will not have enough information to model seq/ack behavior. This does not compromise our success in attack generation. (Extending the victim model to include seq/ack-related fields is trivial.) Next, we discuss how to use SMT constraints to generate symbolic attacks from the above inputs.

Modeling temporal sequence: Given that a symbolic attack is an ordered sequence of symbolic packets, we need to model the progression of time. When an SFW receives a located input packet (an event), the timestep T advances to $T + 1$, changing the system state. While we use the SFW as an example, this modeling of temporal behavior is general across other hosts in the deployment. We use the following functions to represent the state of the SFW at a given T: (1) $\mathrm{State} : \mathrm{Q} \times \mathrm{T} \to \mathrm{Bool}$, a Boolean function indicating if a a given state of the SFW, $s \in \mathrm{Q}$, occurs at timestep T; (2) $\mathrm{Input} : \Delta \times \mathrm{T} \to \mathrm{Bool}$, indicating if a *located packet*, $\sigma \in \Delta$, occurs as an input at timestep T; (3) $\mathrm{Output} : \mathrm{O} \times \mathrm{T} \to \mathrm{Bool}$, indicating if an output packet, $o \in \mathrm{O}$, occurs at timestep T; and (4) $\mathrm{Trs} : \Phi \times \mathrm{T} \to \mathrm{Bool}$, indicating whether a specific transition, $\phi \in \Phi$, occurs at timestep T. A transition is determined by an input symbolic packet (σ), an output symbolic packet (o), and the current state (s).

Modeling senders: We determine a possible sending host based on the given deployment model; e.g., an insider can spoof the source IP. Our encoding is extensible to add more attributes. We encode the SFW behavior using propositional logic. At any given timestep T, we encode the following constraints: (1) exactly one state occurs, (2) at most one input packet is received, (3) at most one output packet is sent, and (4) exactly one transition happens in the model. To encode the SFW model (a Mealy machine), we add pre- and post-conditions to specify the state transition. Intuitively, a transition ϕ_j at a timestep T implies the occurrence of a specific state and an arrival of a located input packet at the same timestep T. After a transition ϕ_j, the state of the SFW changes and we observe a corresponding output packet (defined by the SFW model). We represent this as:

$$(\mathrm{State}(s_i, \mathrm{T}) \ \wedge \ \mathrm{Input}(\sigma_i, \mathrm{T})) \implies \bigvee_j \mathrm{Trs}(\phi_j, \mathrm{T}))$$

$$\mathrm{Trs}(\phi_i, \mathrm{T})) \implies (\mathrm{State}(s_{i+1}, \mathrm{T}+1) \ \wedge \ \mathrm{Input}(\sigma_{i+1}, \mathrm{T}+1) \\ \wedge \ \mathrm{Output}(o_{i+1}, \mathrm{T}+1))$$

We encode the victim using a similar logic. If an attacker's packet reaches the victim, then the next input is dictated by the victim's model. The above encoding allows us to easily generalize to diverse deployments (e.g., multiple insiders, insiders with different capabilities).

Modeling attacker goal: The goal is to find an ordered sequence of σ (located packets) that leads to the SFW forwarding a DATA packet to an internal victim at timestep T. Additionally, we can specify the victim's final state at timestep T, when it receives the DATA packet. Specifying the victim state prunes the search space and generates attacks that are more likely to succeed, especially against criteria S2 and S3 (§3.1). This is because these criteria base attack success on the victim's behavior, and not just the firewall's. We evaluate attack success only when we validate our attacks (§6).

Assumptions in modeling: In modeling host behaviors, we make the following assumptions:

1) Similar to prior work [49], we assume that the SFW receives only 1 packet at a time.
2) At any timestep, any host (attacker, victim, insider, SFW) receives at most 1 packet and outputs at most 1 packet. If a host receives more than 1 packet, those packets can be represented as being received at consecutive timesteps. We do not model a host that sends multiple packets on receiving a single packet. We generate successful attacks (Finding **1**), even with this simplifying assumption.
3) At any timestep, the SFW and the victim occupy exactly 1 state of their corresponding Mealy machines and perform exactly 1 transition on receiving a packet.
4) The victim outputs a packet only when it receives a packet. In our system model (§3.1), the victim is a passive entity that does not attempt to start communication with any host and thus does not send packets on its own.

Improving diversity of attack pathways: Given the above encoding, we now discuss how to generate symbolic attacks. Our aim is to find as many semantically-distinct evasion attacks as possible, given a bounded sequence length. We model the problem similar to bounded model checking (BMC) [50] to find counterexamples with a bounded length, that can be identified within a few iterations/timesteps. By default, the SMT solver terminates upon finding one counterexample, where each counterexample is an evasion attack. To find a new counterexample, we need to add a constraint to block this counterexample. Naively doing so finds many semantically-identical attacks which is not useful and time-consuming. Thus, we need to block not only the counterexample, but also other attacks that exhibit similar semantics/patterns. Our insight is to encode additional constraints to block semantically-identical attacks during the search, and thus only discover semantically-distinct attacks. A natural question is how to (1) define semantically-distinct attacks, and (2) enable the discovery of these distinct attacks. For (1), we define the invariant of the attacks we output and use that invariant to define semantically-distinct attacks. To efficiently search over the SFW model's state space, we want to avoid attacks that can be generated from another attack by simply adding packets that loop in the SFW's state space. Thus, an outputted attack should not have any loops and use the minimum number of packets to traverse the SFW's state space. Thus, we encode a loop-free invariant into the solver.

Definition 5.1 (Loop-free invariant)**.** *A state, s_i, can appear at most once in a state sequence $s_1 \cdots s_n$ transitioned by an attack sequence, $p_1 \cdots p_n$, where p_i is a located packet.*

Given this loop-free invariant, when we discover an attack packet sequence, a=$\{p_1 \cdots p_n\}$, our refinement strategy excludes the exact packet string match. For question (2)

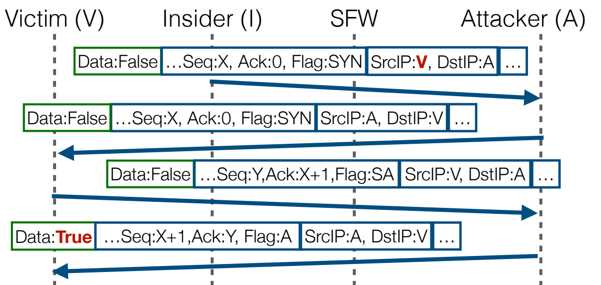

Victim (V)
Insider (I)
SFW
Attacker (A)
Data:False …Seq:X, Ack:0, Flag:SYN SrcIP:V, DstIP:A …
Data:False …Seq:X, Ack:0, Flag:SYN SrcIP:A, DstIP:V …
Data:False …Seq:Y,Ack:X+1,Flag:SA SrcIP:V, DstIP:A …
Data:True …Seq:X+1,Ack:Y, Flag:A SrcIP:A, DstIP:V …

Figure 11: An example of a generated symbolic attack with symbolic seq/ack numbers (src/dst ports omitted for brevity). The start of an arrow denotes a packet sender.

above, we encode this invariant into the model checker to output semantically-distinct attacks.

Definition 5.2 (Semantically-distinct attacks)**.** *Loop-free attack strings, a and a′, are semantically distinct iff* $a \neq a'$.

By construction, the generated attack sequences are loop-free (from Def. 5.1). Additionally, as we do an exact string matching as a refinement strategy, all sequences we generate are distinct as defined above.

Concrete example: Figure 11 shows a symbolic attack sequence. Each vertical line is a host. An edge from host A to B represents a *located packet* sent from A to B. While not shown in this timing diagram, this packet sequence causes state transitions in the SFW model while ensuring that a state is not repeated (Def. 5.1)

5.2. Concrete Attack Generator

The generated symbolic attacks (§5.1) do not have seq/ack fields set in the *located packets*. To get a concrete packet sequence that can be injected into the network, we set concrete seq/ack values for each packet in this module.

Strawman solutions: To see why setting concrete seq/ack values is non-trivial, consider three strawman solutions: (1) randomly assigning seq/ack numbers in all packets, (2) reusing the restricted values from firewall model inference (e.g., using X=1000, Y=2000 for all symbolic packets) and (3) exploring the entire space of seq/ack numbers. Unfortunately, (1) is likely to be dropped by the SFW or the victim. (2) is too restrictive and cannot find potential vulnerabilities when the victim or SFW are more permissive. (3) ensures coverage but is intractable due to the large space (2^{64}) of seq/ack numbers to explore. We show the ineffectiveness of strawmen (1) and (2) in Finding **5**.

Key insight: We observe that, while SFWs tend to check for some relations between seq/ack numbers of TCP packets in a connection, they do not enforce completely strict checking [51]. A strict dependence between packets' seq/ack results in low coverage of the attack space. On the other hand, using random seq/ack results in the SFW dropping packets. Our insight is to maintain a *loose dependence* between packets' seq/ack numbers to trade off coverage and tractability.

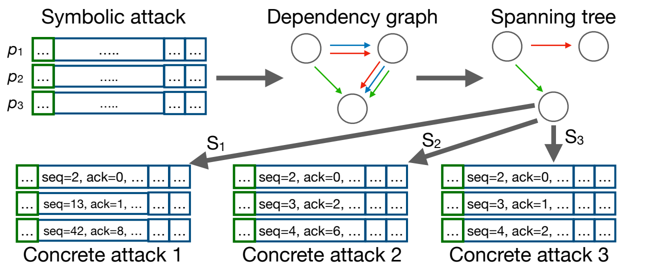


Figure 12: Generating concrete attacks from a symbolic attack by (1) creating a dependency graph, (2) pruning to a spanning tree, and (3) applying *strategies*.

	Random	TCP Compliant	Delta Step
Y.seq	rand()	$\begin{cases} X.seq & X.dir = Y.dir \\ X.ack & X.dir \neq Y.dir \end{cases}$	$\begin{cases} X.seq + \Delta_{seq} & X.dir = Y.dir \\ X.ack + \Delta_{seq} & X.dir \neq Y.dir \end{cases}$
Y.ack	rand()	$\begin{cases} X.ack & X.dir{=}Y.dir \\ X.seq + 1 & X.dir \neq Y.dir \end{cases}$	$\begin{cases} X.ack + \Delta_{ack} & X.dir = Y.dir \\ X.seq + \Delta_{ack} & X.dir \neq Y.dir \end{cases}$

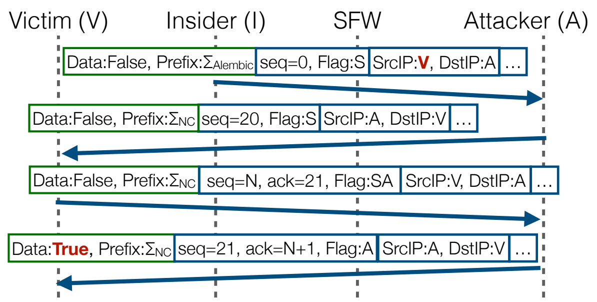
TABLE 2: `update` **function for different strategies.**

Based on this insight, we design a practical 3-step module to generate multiple concrete attacks from a symbolic attack (Figure 12), using an *ensemble of strategies*. Given a symbolic attack, we generate a dependency graph, prune the graph to a spanning tree, and then apply a *strategy* (defined shortly) to set concrete seq/ack values. Each concrete attack is then evaluated for success on a live deployment of a firewall and victim. We describe these 3 steps below:

Generating seq/ack dependency graph: Consider a symbolic packet sequence $p_1 \cdots p_n$. In a TCP-compliant non-evasion context, the seq/ack numbers of packets are defined by the TCP RFC [51]; e.g., in a 3-way handshake SYN-ACK.*ack* == SYN.*seq* + 1. However, our goal is to generate evasion attacks, not to adhere to the TCP protocol. Thus, we relax these constraints and consider loose dependence among the seq/ack numbers of various packets. For instance, there could be a sequence of packets: $\{p_1$: SYN-ACK, p_2: FIN-ACK, p_3: ACK$\}$ and setting p_3's ack depending on p_2's seq may result in an attack. While we allow loose dependencies between seq/ack numbers of various packets, we do not know what seq/ack numbers may result in a successful attack. Hence, we devise multiple heuristics based on diverse packet properties (Appendix B) to add dependencies between packets. Using these heuristics, we generate a graph where each node represents a symbolic packet and edges represent potential dependencies between the packets' seq/ack numbers.

Pruning to a spanning tree: The dependency graph may have nodes that depend on multiple packets (i.e., have multiple incoming edges). We use heuristics to select one incoming edge for each node (details in Appendix C). This converts the dependency graph to a spanning tree.

Setting concrete values: Given an ensemble of *strategies* and a spanning tree, we assign concrete seq/ack values to each packet. Each strategy consists of (a) an `init` function for initializing seq/ack values for packets with no dependencies (no incoming edges), and (b) an `update` function for


Victim (V)
Insider (I)
SFW
Attacker (A)
Data:False, Prefix:Σ_Alembic | seq=0, Flag:S | SrcIP:V, DstIP:A | …
Data:False, Prefix:Σ_NC | seq=20, Flag:S | SrcIP:A, DstIP:V | …
Data:False, Prefix:Σ_NC | seq=N, ack=21, Flag:SA | SrcIP:V, DstIP:A | …
Data:True, Prefix:Σ_NC | seq=21, ack=N+1, Flag:A | SrcIP:A, DstIP:V | …
Figure 13: Example of a generated concrete attack where symbolic seq/ack are substituted with integer values.

updating seq/ack values for other packets. These functions are applied in topological order (details in Appendix D). [10]

We implement three strategies: (i) Random, (ii) TCP compliant, and (iii) Delta step. The seq/ack fields are initialized to random values for the Random strategy and to 0 for the other two strategies. Table 2 shows the `update` functions for these strategies.

- *Random*: This strategy randomly assigns seq/ack numbers and is useful to explore if SFWs and victims accept packets with random, out-of-window seq/ack numbers. We use this as a baseline strategy.
- *TCP compliant*: This strategy attempts to maintain correct TCP semantics. Such a strategy is useful to explore if firewalls and victims accept packets with unexpected TCP flags that still follow correct TCP seq/ack semantics.
- *Delta step*: This strategy gives control over increments in the seq/ack numbers using parameters Δ_{seq} and Δ_{ack}. This is useful when firewalls and/or victims accept TCP packets with seq/ack numbers within a range of window sizes.

Concrete example: Figure 13 shows the concrete attack generated from the symbolic attack in Figure 11 and the *Delta step* strategy with parameters $\Delta_{seq} = 1$, $\Delta_{ack} = 2$. Note that in a concrete attack, symbolic values for seq/ack values are substituted with concrete values but some concrete values (e.g., the seq number of a victim's SYN-ACK) are outside the control of an attacker. These are left as-it-is by *Pryde*.

In this module, our design contribution is the 3-step workflow for concrete attack generation, and not the heuristic recipes used within. As long as an operator follows *Pryde*'s workflow, *Pryde* can admit a variety of heuristics and still generate successful evasion attacks as its success is not sensitive to the specific heuristics an operator may want to employ. For instance, the strawman solutions described earlier (randomly assigning seq/ack, reusing the restricted seq/ack from model inference) do not follow the Concrete Attack Generator workflow and cannot generate successful concrete attacks (Finding **5**) In contrast, *Pryde* generates successful evasion attacks even if the specific heuristics change. We evaluate *Pryde*'s robustness to heuristic changes in Finding **5**. Using an ensemble of strategies, *Pryde* can generate attacks against diverse firewalls and victim stacks. The operator can specify a custom strategy by writing a function to represent its mathematical formulation (Table 2). Finding **5** shows how *Pryde* benefits from using an ensemble of strategies.

10. A subtle issue is that the seq/ack rewriting logic (§4.3) is input template-aware but seq/ack assignment here is template agnostic. This is a pragmatic choice we made. §4.3 is firewall-centric; i.e., making inference tractable in the presence of non-compliance. However, to generate concrete attacks, we also need to take the victim's behavior into account and thus, need more flexibility. Hence, we pick a constrained template-specific approach in §4.3 but adopt a looser approach in this section.

6. Implementation and Evaluation Setup

Implementation: We implement *Pryde*'s Model Inference in Java atop Learnlib [52]. For the Symbolic Attack Generator, we use Python and Z3 [18] (an SMT solver). For the Concrete Attack Generator, we use Python for dependency graph generation and generate concrete packets using *scapy* [53]. We use the *boto3* [54] Python API and CloudFormation templates to spin up testbeds for Model Inference and attack validation in AWS EC2.

Evaluation Setup: We evaluate *Pryde* on 4 off-the-shelf firewalls and 4 victim stacks. We chose these firewalls based on their popularity and ease of deployment. 3 of these are available commercially with closed-source implementations while 1 is open-source. We deploy 2 firewalls on Cloudlab [55] and 2 firewalls on AWS using the AWS Marketplace [19] and EC2. We chose the following victim stacks that are popular [31], [32] in IoT devices:

1) Telnet server on Contiki OS[56]
2) HTTP server on TTYLinux[57] (`thttpd` daemon)
3) TCP server on Ubuntu 11.04 (`netcat`)
4) TCP server on Ubuntu 16.04 (`netcat`)

All 4 victim stacks have open-source implementations. Even though we have the source code for the victims and one firewall, we treat them as black-boxes. For each victim, we handcraft an approximate model and identify states in the model where the victim is likely to accept TCP data. We use this information for symbolic attack generation (§5.1).

For each SFW and the 10 input alphabets from §4.2, we infer a symbolic model. We run the Symbolic Attack Generator module for each pair of inferred symbolic models and victim models. We constrain the Symbolic Attack Generator to generate symbolic attacks upto length 10 and specify the end state of the victim model. We then use the 3 *strategies* mentioned in §5.2 to generate multiple concrete attacks from each symbolic attack. We consider 5 instances of the *delta-step* strategy with $\Delta_{seq}, \Delta_{ack} = \{(1,1),(1,2),(2,1),(2,2),(10000,10000)\}$. We use these parameters to exercise different behaviors – off by 1, off by a small number >1, and off by a large number. Using all the 3 strategies, each symbolic attack generates 7 concrete attacks. Given these concrete attacks, we validate them against a live deployment of the firewall and victim, based on the success criteria mentioned in §3.1. Since S2 and S3 are stronger criteria, we focus on these in our evaluation. Finally, we cluster successful attacks based on an equivalence criteria (described below) and analyze patterns in successful attacks.

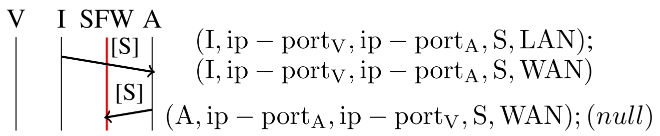

V
I
SFW
A
[S]
(I, ip − port$_V$, ip − port$_A$, S, LAN);
(I, ip − port$_V$, ip − port$_A$, S, WAN)
[S]
(A, ip − port$_A$, ip − port$_V$, S, WAN); (*null*)

Figure 14: Example of symbolic representation used for checking attack equivalence.

Attack equivalence: Many evasion attacks are semantically related and provide no additional value to operators or vendors trying to prioritize countermeasures. To this end, we define *equivalence criteria* to determine if two sequences are related, and explain how this differs from the loop-free invariant (Def. 5.1). We define criteria at the symbolic and concrete level, using attributes visible "on the wire".

- *Symbolic:* As shown in Figure 14, we represent each symbolic packet as a pair of tuples of a packet before and after it is acted on by the firewall: (sender, src-ip-port, dst-ip-port, flags, firewall-iface). firewall-iface is the interface at which the firewall sees the packet. 2 symbolic attacks are semantically related if their packet representations are equal. This representation is based on the prediction of firewall behavior as per models inferred in §4. This equivalence criteria differs from Def. 5.1 as this criteria differentiates attacks based on attributes observable on the wire, while Def. 5.1 also considers model-internal attributes.
- *Concrete:* For each concrete attack, we consider the packet traces at each host (attacker, victim, insider). We represent each sent and received packet as (sender, src-ip-port, dst-ip-port, flags, strategy-class). Here strategy-class is a qualitative parameter to represent seq/ack behavior namely; random vs. off by a large number vs. off by a small number. We define strategy-class $= 0$ for *random*, strategy-class $= 1$ for the *delta-step* instance with $\Delta_{seq}, \Delta_{ack} = (10000, 10000)$ and strategy-class $=$ 2 for others. Unlike symbolic attacks, this representation is based on the actual behavior during attacks, not the behavior predicted by the inferred models.

Given these equivalence criteria, we post-process the results and only report unique uncovered attacks.

7. Results

We summarize our findings and implications along 4 dimensions: (1) firewall, (2) victim, (3) attack success criteria, and (4) attacker capability. Overall, we find that: (1) *Pryde* outperforms alternatives [15] in terms of number of successful attacks generated; (2) *Pryde* uncovers interesting attack patterns unique to different settings; and (3) *Pryde*'s design choices are critical to tackle the diverse problem space.

Finding 1: **Pryde** ***generates up to 2-3 orders of magnitude more distinct successful concrete attacks than baseline black-box fuzzing and prior work on firewall evasion [15] (Table 3).***

We compare *Pryde* with 2 fuzzing approaches and a state-of-the-art genetic algorithm based evasion tool called Geneva [15]. Prior work on protocol fuzzing [11], [12], [25] is inapplicable in our context and cannot be used as baseline vs. *Pryde*. These works assume a simple client-server network deployment and do not model other deployments or stateful firewalls. We consider two natural baseline fuzzing approaches to compare *Pryde* against – stateless fuzzing and model-guided fuzzing (inspired by prior work [58]).

		FW1	FW2	FW3	FW4
S2	***Pryde***	4	169	6956	539
	Stateless Fuzzing	0	0	0	0
	Model-guided Fuzzing	0	0	0	0
	Geneva [15]	0	0	0	0
S3	***Pryde***	4	169	6954	61
	Stateless Fuzzing	0	0	0	0
	Model-Guided Fuzzing	0	0	0	0
	Geneva [15]	0	0	0	0

TABLE 3: Number of distinct successful concrete attacks generated by *Pryde*, stateless fuzzing, model-guided fuzzing and Geneva [15] for different success criteria.

Table 3 shows the number of distinct successful concrete attacks generated by *Pryde* and these baselines for success criteria S2 and S3. These results include all 4 victim stacks. Detailed results of *Pryde* for all 3 success criteria are given in Appendix E. Below, we discuss the baseline approaches and their experimental setup.

Stateless fuzzing: This approach serves as a deployment-aware model-free baseline. We generate random sequences of TCP packets of lengths 1 to 7 with the knowledge of the deployment which includes host IPs, TCP flags, and the insider's capability to spoof packets as the victim. This approach fails to generate any successful attacks (Table 3) across firewalls and victims for success criteria S2 and S3.

Model-guided fuzzing: We consider a deployment-agnostic model-guided baseline, which leverages the SFW models inferred in §4. For each SFW model, we identify "data-forwarding" states; i.e. those where the SFW forwards TCP DATA from the external to the internal network. We identify all paths from the start state to a data-forwarding state and sample a subset of these paths (for path lengths 1 to 7). Each path represents a series of transitions in the SFW model and corresponds to a symbolic packet sequence. We set the seq/ack numbers randomly in each packet. As shown in Table 3, this approach does not generate any successful attacks for success criteria S2 or S3.

Geneva: To run Geneva [15] in our setting, we add a custom plugin for *Pryde* (using the Geneva docs [59]) which defines a fitness function to be used for evaluating evasion strategies. This fitness function sends a SYN from the attacker to the victim, then sends a DATA and finally, checks if it the attacker receives an ACK from the victim. We leave other configurations in Geneva as the default and do not change any internal logic. For each firewall and victim pair, we run Geneva as a client-side evaluator [59] on the attacker, for 50 generations with a population of 200 evasion strategies. We find that Geneva is unable to generate any effective evasion strategies for criteria S2 and S3 (refer to Table 3). This is due to a mismatch between Geneva's assumptions

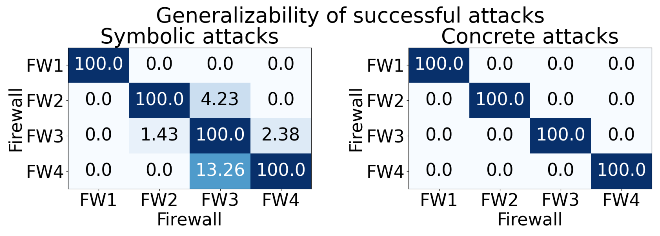


Figure 15: Percentage equivalence while comparing successful symbolic and concrete attacks generated for one firewall (x-axis) against those of another firewall (y-axis).

and the deployment model we consider. Geneva requires visibility into attack success to evolve strategies. However, since neither the attacker nor the insider can observe the firewall forwarding packets to the victim or the victim's behavior on receiving the attacker's packets, this information is not available in our system setting.

Finding 2: ***At least 86.74% and up to 100% of the attacks successful against one firewall are unsuccessful against another. This shows the need for a generalizable workflow that can generate deployment-specific attacks (Figure 15).***

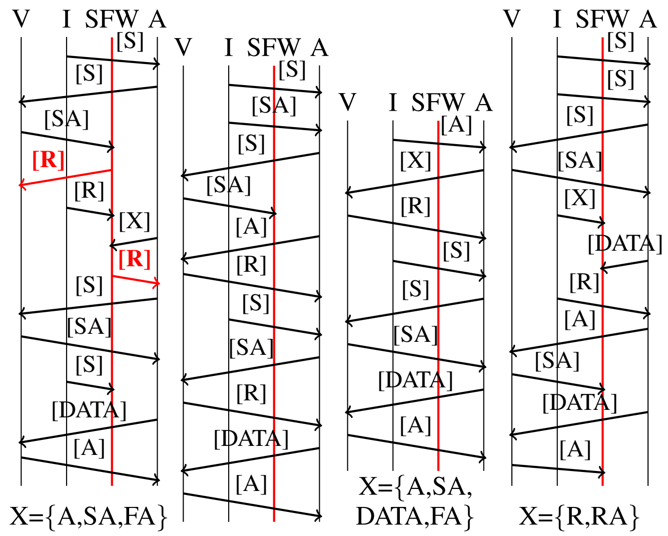
A natural question is: Can we re-purpose attacks that work against one SFW to attack another SFW, instead of re-running *Pryde*. To answer this, we consider pairs of SFWs: $\mathrm{FW_X}$ and $\mathrm{FW_Y}$. For each attack successful against $\mathrm{FW_X}$ (and any victim), we check if it is equivalent to an attack successful against $\mathrm{FW_Y}$, using the equivalence criteria described in §6. We calculate the percentage of attacks successful against $\mathrm{FW_X}$ that pass this check.

Figure 15 shows the results for symbolic and concrete attacks. Each cell in the heatmap shows what percentage of attacks successful against $\mathrm{FW_X}$ (firewall on X-axis) are equivalent to any attack successful against $\mathrm{FW_Y}$ (firewall on Y-axis). For symbolic attacks, at least 86.74% and up to 100% of the attacks are distinct between pairs of firewalls. For concrete attacks, 100% of all attacks successful against a firewall do not translate to success against another firewall. These results validate that evasion attacks need to be tailored to an SFW and deployment, and that *Pryde*'s generalizable workflow can generate such tailored attacks.

Finding 3: **Pryde** ***uncovers subtle and interesting attack patterns, that are unique to each firewall (Figure 16).***

We analyze the set of successful attacks and report subtle interesting patterns that we observe for each firewall. Here, too, we consider attacks successful against criteria S2 where the insider is present.

Figure 16a shows the insider spoofing a SYN and punching a hole in FW-1 which allows the attacker to send a SYN to the victim. The victim receives a RST (in red and bold), which is spoofed and sent by FW-1. FW-1 consistently spoofs RSTs to try to stop communication between the attacker and the victim. However, this RST does not completely teardown the connection state as a later SYN



(a) FW-1 **(b) FW-2** **(c) FW-3** **(d) FW-4**

Figure 16: Unique attack patterns for different firewalls.

from the attacker is forwarded. Ultimately, this allows the attacker to send malicious data to the victim and the attack is successful.

Figure 16b depicts unique attacks against FW-2. The attacker attempts to communicate with the victim but receives RSTs from the victim twice. Inspite of this, malicious data from the attacker is forwarded to the victim by FW-2 and accepted by the victim. We discuss FW-3 in the next finding (Finding **4**).

Figure 16d shows unique attacks with 2 spoofed SYNs. As seen before with FW-1 and FW-2, RSTs don't fully teardown the connection state in FW-4. In this specific attack pattern, the last ACK from the victim is not forwarded. Thus, this attack pattern is successful based on criteria S2 but not based on criteria S3. Note that the ACK being blocked by the firewall only limits the observability of attack success by the attacker, not the impact of the attack on the victim.

Finding 4: **Pryde** ***generates attacks against FW-3 that (a) require no insider, and (b) start with a TCP ACK from the insider (Figure 16c).***

Pryde discovers that FW-3 trivially allows a TCP DATA packet from the *attacker* to the *victim* without any preceding packets. This attack satisfies success criteria S1 and succeeds without an insider. FW-3 is the only firewall against which *Pryde* discovers successful attacks (for criteria S2 and S3) that start with an ACK from the insider. FW-3 forwards these to the attacker without a previous SYN (Figure 16c). We observe that other firewalls are not vulnerable to deployments without the insider.

Finding 5: **Pryde*****'s design choices are crucial for generating successful concrete attacks across diverse firewalls and victims.***

We present a systematic ablation study of *Pryde*'s modules and the design choices shown in Figure 4. We discuss

		FW1	FW2	FW3	FW4
S2	*Pryde*	4	169	6956	539
	Pryde + subset of DAG generation heuristics	2	186	4317	1059
	Pryde + random edge selection for pruning	4	264	6121	661
S3	*Pryde*	4	169	6954	61
	Pryde + subset of DAG generation heuristics	2	186	4317	618
	Pryde + random edge selection for pruning	4	264	6121	179

TABLE 4: Number of successful concrete attacks generated by *Pryde* and its variations with modified heuristics.

strawman solutions and show *Pryde*'s benefit over these.

(1) Deployment-agnostic vs deployment-aware: We compare *Pryde* with Geneva [15] and a deployment-agnostic model-guided baseline in Finding **1**. Table 3 shows that these approaches are ineffective.

(2) Monolithic vs modular: Finding **1** compares *Pryde* against stateless fuzzing, a deployment-aware monolithic approach, and shows that this approach is ineffective.

(3) Model-free vs model-guided: The stateless fuzzing and Geneva [15] baselines discussed above are model-free approaches. This shows the need for adopting a model-guided approach.

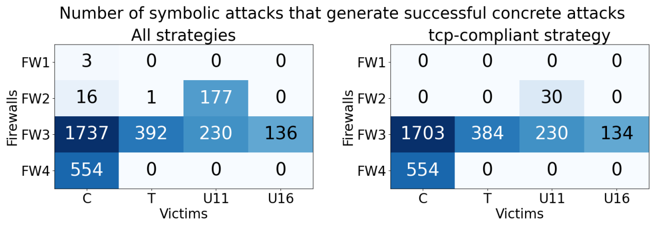
(3a) Value of evasion-aware models: We run *Pryde* using the alphabet sets proposed in Alembic [45]: (1) $\Sigma_{\text{Evasion}} = \Sigma_{\text{Alembic}}$ ($\Sigma_{\mathbb{NC}}$ is a null set); and (2) $\Sigma_{\mathbb{NC}} = \Sigma_{\mathbb{NC}}^{\text{rand}11}$. We discover no successful attacks against any firewall and victim pair, for any success criteria. This result validates our choice to extend Alembic [45] to explicitly consider non-TCP-compliant scenarios for model inference.

(4a) Symbolic attack generation: Running the model checker without the loop-free invariant (Def. 5.1) generates semantically-identical symbolic attacks, that provide no extra value to an operator.

(4b) Concrete attack generation: We compare against 2 strawman approaches to set seq/ack numbers (mentioned in §5.2). The first approach ignores any structure of the symbolic attack and assigns seq/ack numbers randomly to each packet. The second approach leverages the fact that each symbolic packet had a concrete equivalent packet which was played against the firewall in the Model Inference module. This approach naively re-uses the seq/ack values from those concrete packets and sets them in the symbolic attack. We observe that both these approaches are ineffective.

Heuristics for concrete attack generation: We evaluate *Pryde*'s robustness to changes in heuristics for (a) dependency graph generation, and (b) pruning to a spanning tree, in the Concrete Attack Generator (§5.2). For (a), we modify dependency graph generation to use a subset of the

11. Alembic refers to these as `correct-seq` and `combined-seq`



C – Contiki; T – TTYLinux; U11 – Ubuntu 11; U16 – Ubuntu 16

Figure 17: Number of symbolic attacks that generate successful concrete attacks, using all strategies and using only the *tcp-compliant* strategy, respectively.

heuristics[12] from Appendix B. For (b), instead of using heuristics from Appendix C to prune edges and generate a spanning tree, we select an edge at random from the set of incoming edges at each node. For both experiments, we leave the other steps as it is and test the generated attacks against criteria S2 and S3.

Table 4 shows that while changes in these heuristics do affect the exact number of successful concrete attacks, *Pryde* is robust at generating successful attacks across diverse firewalls and victims, robust to heuristic choices.

Ensemble of strategies: To evaluate the benefit of using an ensemble of strategies, we consider symbolic attacks that generate at least 1 successful concrete attack. Figure 17 shows the results of using the entire ensemble of strategies, and only the *tcp-compliant* strategy (we consider this strategy as it produces the largest number of attacks). The *tcp-compliant* strategy misses 100% and 75% of the attacks for FW-1 and FW-2 respectively (even though it performs well for FW-3 and FW-4). Thus, even using the "best" strategy results in missing a majority/all of the attacks for 2 out of the 4 firewalls, supporting our choice to use an ensemble of strategies.

8. Countermeasures

Robust network configuration: Stateful firewalls and network deployments should utilize best practices. Specifically, firewalls should use strict seq/ack checking logic, validate packet length/malformed packets, and protect against spoofing. Vendors and network operators should enable these by default, to mitigate the risk of firewall circumvention.

Model-guided defense: Vendors can use *Pryde*'s inferred models and evasion attacks to identify bug fixes. The models can help narrow down the exact code region/path for a given attack. We envision doing this iteratively. After patching specific bugs, we can re-run *Pryde* to validate the effectiveness of the patches against the attacks and also identify if the patches introduced any new evasion opportunities.

Using traffic normalizers: Operators can use our attacks to synthesize policies for traffic normalizers [60]. Our post-

12. We only used heuristics `CONSECUTIVE`, `VICTIM_SA` and `ATTACKER_S`. The first one adds dependence between consecutive packets while the other two add dependence between SYN, SYN-ACK and ACK based on the TCP 3-way handshake

processing analysis for summarizing the patterns of attacks can be used to generate attack signatures, which can be used by traffic normalizers to drop malicious packet sequences.

Automating firewall patching: Given our findings, we can leverage techniques from program synthesis and program patching to generate patches for the firewall connection-tracking logic that are correct by construction [61].

9. Conclusions

Stateful firewalls are ubiquitously used to protect networks from adversaries. We propose *Pryde*, a modular model-guided framework to automatically analyze these firewalls and uncover evasion attacks. Our analysis shows the importance of *Pryde*'s customizability and unearths unique behaviors for different firewalls. Thus, *Pryde* is a valuable tool for firewall vendors as well as network administrators. An interesting direction of future work is to combine our work with the power of model-free techniques by using model-guided results as "seeds" for model refinement.

Acknowledgements: We thank our anonymous shepherd and the anonymous reviewers for their suggestions. This work was funded in part by NSF awards CNS-2132639 and CNS-1552481, the Combat Capabilities Development Command Army Research Laboratory under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA), the Ann and Martin McGuinn Graduate Fellowship, and the Kavcic-Moura research award. The views and conclusions in this paper are those of the authors and should not be interpreted as presenting the official policies or position, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

[1] "Katherine Pryde," https://marvel-movies.fandom.com/wiki/Katherine_Pryde, last accessed Dec 1, 2023.

[2] "What is a firewall? Firewalls explained and why you need one," https://us.norton.com/blog/privacy/firewall, last accessed Dec 1, 2023.

[3] "The Importance of Industrial Control Systems (ICS) Firewalls," https://www.therma.com/the-importance-of-industrial-control-systems-ics-firewalls/, last accessed Dec 1, 2023.

[4] "Cloud-based firewalls are key to protecting employees while working remotely," https://securityboulevard.com/2020/05/cloud-based-firewalls-are-key-to-protecting-employees-while-working-remotely/, last accessed Dec 1, 2023.

[5] "Red Hat Sprucing OpenShift for Network Functions on Kubernetes," https://www.lightreading.com/virtualization/red-hat-sprucing-openshift-for-network-functions-on-kubernetes, last accessed Dec 1, 2023.

[6] L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah, and P. Mohapatra, "FIREMAN: A toolkit for firewall modeling and analysis," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2006.

[7] E. Al-Shaer, A. El-Atawy, and T. Samak, "Automated pseudo-live testing of firewall configuration enforcement," *IEEE J. Sel. Areas Commun.*, 2009.

[8] Adel El-Atawy, K. Ibrahim, H. Hamed, and Ehab Al-Shaer, "Policy segmentation for intelligent firewall testing," in *NPSec*, 2005.

[9] Y. Cao, Z. Wang, Z. Qian, C. Song, S. V. Krishnamurthy, and P. Yu, "Principled unearthing of tcp side channel vulnerabilities," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2019.

[10] T. Ferreira, H. Brewton, L. D'Antoni, and A. Silva, "Prognosis: Closed-box analysis of network protocol implementations," in *Proceedings of the 2021 ACM SIGCOMM Conference*. Association for Computing Machinery, 2021.

[11] S. Gorbunov and A. Rosenbloom, "Autofuzz: Automated network protocol fuzzing framework," *IJCSNS*, 2010.

[12] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *SecureComm*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer, 2015.

[13] I. Karim, A. Al Ishtiaq, S. R. Hussain, and E. Bertino, "Blediff: Scalable and property-agnostic noncompliance checking for ble implementations," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2022.

[14] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager, "Combining model learning and model checking to analyze tcp implementations," in *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*, 2016.

[15] K. Bock, G. Hughey, X. Qiang, and D. Levin, "Geneva: Evolving censorship evasion strategies," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2019.

[16] K. Bock, G. Hughey, L.-H. Merino, T. Arya, D. Liscinsky, R. Pogosian, and D. Levin, "Come as you are: Helping unmodified clients bypass censorship with server-side evasion," in *Proceedings of the 2020 ACM SIGCOMM Conference*. Association for Computing Machinery, 2020.

[17] R. Jhala and R. Majumdar, "Software model checking," *ACM Comput. Surv.*, 2009.

[18] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, ser. Lecture Notes in Computer Science. Springer, 2008.

[19] "AWS Marketplace," https://aws.amazon.com/marketplace, last accessed Dec 1, 2023.

[20] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "Sok: Security evaluation of home-based iot deployments," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[21] D. Breitenbacher, I. Homoliak, Y. L. Aung, N. O. Tippenhauer, and Y. Elovici, "Hades-iot: A practical host-based anomaly detection system for iot devices," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. Association for Computing Machinery, 2019.

[22] "Guidelines on Firewalls and Firewall Policy," https://www.govinfo.gov/content/pkg/GOVPUB-C13-f52fdee3827e2f5d903fa8b4b66d4855/pdf/GOVPUB-C13-f52fdee3827e2f5d903fa8b4b66d4855.pdf, last accessed Dec 1, 2023.

[23] "2020 Sees a 100 Percent Rise of Compromised IoT Devices," https://atlas-cybersecurity.com/cyber-threats/2020-sees-a-100-percent-rise-of-compromised-iot-devices/, last accessed Dec 1, 2023.

[24] Z. Wang, S. Zhu, Y. Cao, Z. Qian, C. Song, S. V. Krishnamurthy, K. S. Chan, and T. D. Braun, "Symtcp: Eluding stateful deep packet inspection with automated discrepancy discovery," in *NDSS*, 2020.

[25] V. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: A greybox fuzzer for network protocols," in *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track*, 2020.

[26] P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg, "Model learning and model checking of ssh implementations," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. Association for Computing Machinery, 2017.

[27] S. R. Hussain, I. Karim, A. A. Ishtiaq, O. Chowdhury, and E. Bertino, "Noncompliance as deviant behavior: An automated black-box noncompliance checker for 4g lte cellular devices," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2021.

[28] "Heartbleed," https://heartbleed.com/, last accessed Dec 1, 2023.

[29] "Firewall Penetration Testing: Steps, Methods And Tools That Work," https://purplesec.us/firewall-penetration-testing/, last accessed Dec 1, 2023.

[30] "CVEs for Linux 2.6 kernel," https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/version_id-410986/Linux-Linux-Kernel-2.6.html, last accessed Dec 1, 2023.

[31] "Home Router Security Report 2020," https://www.fkie.fraunhofer.de/content/dam/fkie/de/documents/HomeRouter/HomeRouterSecurity_2020_Bericht.pdf, last accessed Dec 1, 2023.

[32] G. Oikonomou, S. Duquennoy, A. Elsts, J. Eriksson, Y. Tanaka, and N. Tsiftes, "The contiki-ng open source operating system for next generation iot devices," *SoftwareX*, vol. 18, p. 101089, 2022.

[33] "Cisco Nexus 9000 Series NX-OS Security Configuration Guide, Release 7.x," https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus9000/sw/7-x/security/configuration/guide/b_Cisco_Nexus_9000_Series_NX-OS_Security_Configuration_Guide_7x/b_Cisco_Nexus_9000_Series_NX-OS_Security_Configuration_Guide_7x_chapter_010111.html, last accessed Dec 1, 2023.

[34] https://www.tenable.com/audits/items/CIS_Cisco_NX-OS-v1.0.0_Level_2.audit:06ddd866be61e9feb8b36e058ade8d1e, last accessed Dec 1, 2023.

[35] "FBI recommends that you keep your IoT devices on a separate network," https://www.zdnet.com/article/fbi-recommends-that-you-keep-your-iot-devices-on-a-separate-network/, last accessed Dec 1, 2023.

[36] "Three security practices that IoT will disrupt," https://www.csoonline.com/article/2599509/three-security-practices-that-iot-will-disrupt.html, last accessed Dec 1, 2023.

[37] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the mirai botnet," in *Proceedings of the 26th USENIX Security Symposium*. USENIX Association, 2017.

[38] "Californian government agency breached in phishing attack," https://www.grcworldforums.com/security/californian-government-agency-breached-in-phishing-attack/1095.article, last accessed Dec 1, 2023.

[39] "Microsoft Employees Exposed Own Company's Internal Logins," https://www.vice.com/en/article/m7gb43/microsoft-employees-exposed-login-credentials-azure-github, last accessed Dec 1, 2023.

[40] "Twitter hack: Staff tricked by phone spear-phishing scam," https://www.bbc.com/news/technology-53607374, last accessed Dec 1, 2023.

[41] "Attackers hijack UK NHS email accounts to steal Microsoft logins," https://www.bleepingcomputer.com/news/security/attackers-hijack-uk-nhs-email-accounts-to-steal-microsoft-logins/, last accessed Dec 1, 2023.

[42] "IoT Hackers Target Millions of Devices in Pandemic," https://www.msspalert.com/cybersecurity-research/iot-report-zscaler-findings/, last accessed Dec 1, 2023.

[43] "Insider threats, supply chains, and IoT: Breaking down a modern-day cyber-attack," https://darktrace.com/blog/insider-threats-supply-chains-and-iot-breaking-down-a-modern-day-cyber-attack, last accessed Dec 1, 2023.

[44] "Somebody's Watching: Hackers Breach Ring Home Security Cameras," https://www.nytimes.com/2019/12/15/us/Hacked-ring-home-security-cameras.html, last accessed Dec 1, 2023.

[45] S. Moon, J. Helt, Y. Yuan, Y. Bieri, S. Banerjee, V. Sekar, W. Wu, M. Yannakakis, and Y. Zhang, "Alembic: Automated model inference for stateful network functions," in *NSDI*. USENIX Association, 2019.

[46] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar, "BUZZ: testing context-dependent policies in stateful networks," in *NSDI*. USENIX Association, 2016.

[47] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, 1987.

[48] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *NSDI*. USENIX Association, 2012.

[49] Y. Yuan, S.-J. Moon, S. Uppal, L. Jia, and V. Sekar, "Netsmc: A custom symbolic model checker for stateful network verification." in *NSDI*. USENIX Association, 2020.

[50] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal Methods in System Design*, 2001.

[51] "Rfc 9293: Transmission control protocol (tcp)," https://datatracker.ietf.org/doc/html/rfc9293, last accessed Dec 1, 2023.

[52] H. Raffelt and B. Steffen, "Learnlib: A library for automata learning and experimentation," in *FASE*, ser. Lecture Notes in Computer Science, vol. 3922. Springer, 2006.

[53] "Scapy," http://www.secdev.org/projects/scapy/, last accessed Dec 1, 2023.

[54] "Boto3," https://github.com/boto/boto3, last accessed Dec 1, 2023.

[55] "Cloudlab," https://www.cloudlab.us/, last accessed Dec 1, 2023.

[56] "Contiki OS," https://www.contiki-ng.org/, last accessed Dec 1, 2023.

[57] "TTYLinux," https://www.minimalinux.org/ttylinux/, last accessed Dec 1, 2023.

[58] J. De Ruiter and E. Poll, "Protocol state fuzzing of tls implementations," in *Proceedings of the 24th USENIX Conference on Security Symposium*. USENIX Association, 2015.

[59] "Running the Evaluator – geneva documentation," https://geneva.readthedocs.io/en/latest/howitworks/evaluator.html, last accessed Dec 1, 2023.

[60] M. Handley, V. Paxson, and C. Kreibich, "Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics," in *USENIX Security Symposium*. USENIX Association, 2001.

[61] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. J. Argyraki, and G. Candea, "A formally verified NAT," in *Proceedings of the 2017 ACM SIGCOMM Conference*. Association for Computing Machinery, 2017.

Appendix A. Vendor disclosure

We share anecdotes from the vendor interactions below:

- FW-1: On the vendor's request, we evaluated a newer version of FW-1 and found successful attacks. We observed that the newer version was more aggressive in spoofing TCP RSTs. Thus, some attacks successful against criteria

S3 against the older version only satisfied criteria S1 against the newer version. This re-emphasizes the need for a model-guided workflow that can detect subtle differences in firewall implementations and generate attacks that are customized to a firewall's behavior.
- FW-2: The vendor acknowledged our initial report but did not follow up.
- FW-3: The vendor mentioned that they had enabled a stricter TCP check by default in their configuration, which could potentially thwart our attacks. In fact, this check was enabled only after our initial disclosure to the vendor back in Sep 2020. While enabling this check blocked some attacks, we still found many attacks successful against criteria S1. Some attacks that are successful against criteria S3 with the strict check disabled only satisfied criteria S1 with the check enabled. Similar to FW-1 above, this re-emphasizes the need for a model-guided workflow.
- FW-4: On the vendor's request, we evaluated a newer version of FW-4 and all attacks were successful against the newer version, including attacks that passed our strictest success criteria (criteria S3 in Figure 3).

Appendix B. Dependency graph generation

In the dependency graph, a directed edge from node X to Y signifies a potential dependency of the seq/ack of packet Y on the seq/ack of packet X. We use 6 heuristics to add these edges and construct the graph.

1) PRED: X is a predecessor of Y, and X and Y have the same prefix.
2) CONSECUTIVE: X is the immediate predecessor of Y.
3) SAME_DIR: X is the latest predecessor of Y such that X and Y have the same prefix and the same direction.
4) DIFF_DIR: X is the latest predecessor of Y such that X and Y have the same prefix but different directions.
5) VICTIM_SA: X is a SYN-ACK packet sent by the victim; Y is a DATA packet sent by the attacker.
6) ATTACKER_S: X is a SYN packet sent by the attacker; Y is a SYN-ACK packet sent by the victim.

Appendix C. Edge selection

Given a dependency graph, Algorithm 1 selects edges to create a spanning tree.

Appendix D. Setting concrete seq/ack using a strategy

Given a spanning tree, Algorithm 2 describes how to assign seq/ack numbers to each packet (each node in the spanning tree represents a packet).

Algorithm 1: Edge selection

```
Function EdgeSelection(dependencyGraph):
    tree_edges = []
    for node in dependencyGraph.nodes do
        edges = PRUNE_VISIBLE(node.incoming_edges)
        if len(edges) == 1 then
            tree_edges.append(edges[0])
            continue
        e = PRUNE_TYPE(edges, [VICTIM_SA, ATTACKER_S])
        if e is not None then
            tree_edges.append(e)
            continue
        e = PRUNE_TYPE(edges, [SAME_DIR, DIFF_DIR])
        if e is not None then
            tree_edges.append(e)
            continue
        e = PRUNE_TYPE(edges, [PRED])
        if e is not None then
            tree_edges.append(e)
            continue
        e = PRUNE_TYPE(edges, [CONSECUTIVE])
        if e is not None then
            tree_edges.append(e)
            continue
    return (dependencyGraph.nodes, tree_edges)
Function PRUNE_TYPE(edges, types):
    edges.SORT() /* sort edges in decreasing order of source node index */
    for edge in edges do
        for type in types do
            if edge.type == type then
                return edge
    return None
Function PRUNE_VISIBLE(edges):
    visibleEdges = []
    for edge in edges do
        if edge.src.direction == edge.dst.direction then
            /* src and dst packets are in the same direction */
            visibleEdges.append(edge)
        else
            if edge.src.is_forwarded then
                /* src pkt is forwarded to its destination */
                visibleEdges.append(edge)
    return visibleEdges
```

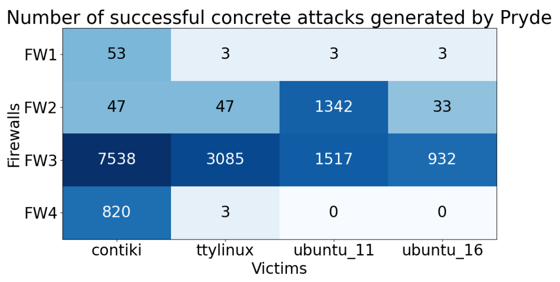
Appendix E. Detailed results

For all 3 success criteria given in §3.1, Figure 18 shows the number of unique successful concrete attacks, for all

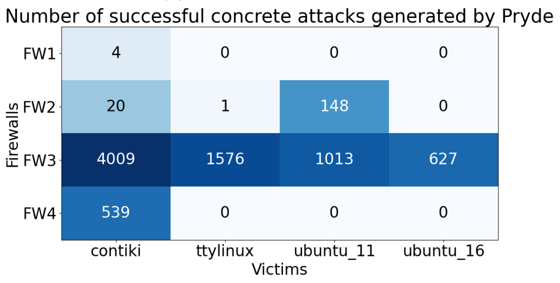
Algorithm 2: Strategy application

```
Function StrategyApplication(spanningTree, initialization, update):
    nodes = TOPOLOGICAL_SORT(spanningTree)
    for node in nodes do
        if node has no incoming edge then
            $initialization(node.seq)$
            $initialization(node.ack)$
        else
            $update(node.seq)$
            $update(node.ack)$
```

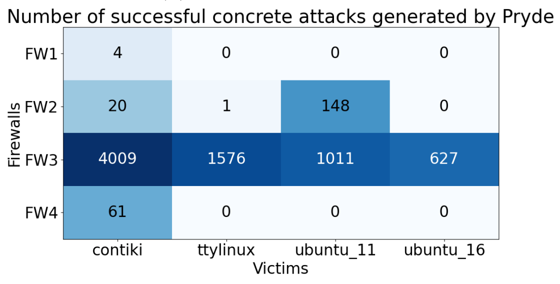
firewalls and victims. Figure 19 contains results for the effectiveness of all strategies against different firewalls and victims.



(a) Success criteria: S1

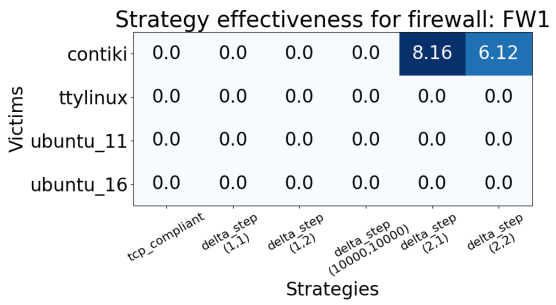


(b) Success criteria: S2

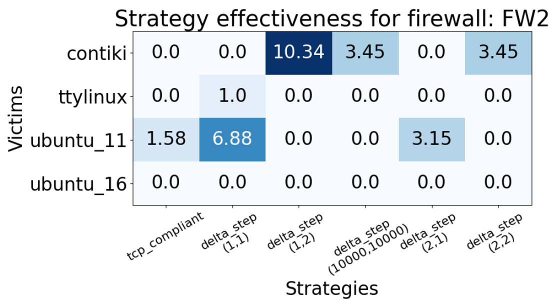


(c) Success criteria: S3

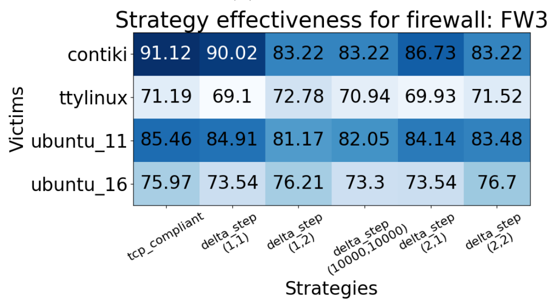
Figure 18: Number of distinct successful concrete attacks generated by *Pryde*



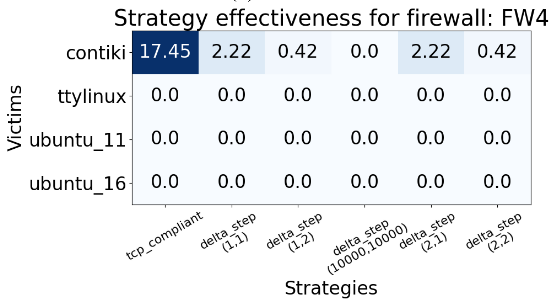
(a) FW-1



(b) FW-2



(c) FW-3



(d) FW-4

Figure 19: Strategy effectiveness for different firewalls and victims

Appendix F. Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

F.1. Summary

The paper presents Pryde, which combines black-box automata learning with model checking to identify evasion attacks against stateful firewall deployments. Pryde's approach has been shown to be effective in identifying more than 6,000 evasion attacks against 4 popular stateful firewalls and 4 networking stacks.

F.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field

F.3. Reasons for Acceptance

1) The paper presents a principled model inference-based approach embodied in a tool called Pryde, which can identify evasion attacks against a stateful firewall implementation in a highly automated fashion.
2) Pryde's effectiveness has been demonstrated by discovering many unique evasion attacks against stateful firewall products, some of which are from commercial-grade vendors.
3) Experimentally Pryde has been shown to be more effective in generating 2-3 orders of magnitude more evasion attacks compared to two baselines, namely, a custom model-guided fuzzer and Alembic++ (Alembic-inferred stateful firewall model enhanced with Pryde's symbolic attack generation).

F.4. Noteworthy Concerns

1) Although the paper demonstrates Pryde's effectiveness by experimentally comparing it with two baselines, namely, model-guided fuzzing and Alembic++, the paper could be further improved by directly comparing Pryde with SymTCP.
2) One of the reviewers had concerns that the experiments presented in the paper were performed using outdated network stacks and it is unclear whether Pryde is equally effective in newer network stacks.

Appendix G. Response to the Meta-Review

We emphasize that our contribution in this paper is *Pryde*, a generalizable framework for uncovering evasion attacks against stateful firewalls, not the attack sequences themselves. While we have tested *Pryde* on a few popular IoT/router network stacks, *Pryde* itself generalizes to other stacks or firewalls. Practitioners can run *Pryde* in these new scenarios and uncover custom evasion attacks. Appendix A mentions some related anecdotes for FW-1 and FW-3.