Practical Black-Box Analysis for Network Functions and Services

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Soo-Jin Moon

BASc., Electrical Engineering, University of Waterloo

Carnegie Mellon University
Pittsburgh, PA

September 2020

Acknowledgments

I am deeply grateful to many outstanding people who have helped to make this dissertation possible.

First, I am immensely grateful to my advisor, Vyas Sekar, for guiding me through the Ph.D. process, giving me the tremendous freedom to chart my own course and believing in my work. Vyas has been instrumental in teaching me how to formulate the problem in the right context and articulate my thoughts and work clearly. Among many things, he taught me how to efficiently and rigorously approach research problems. Working with him made me the researcher I am today.

I want to take this opportunity to thank and acknowledge the support of my thesis committee members: Sujata Banerjee, Lujo Bauer, Bryan Parno, and Michael K. Reiter. I am grateful to Sujata for her mentorship, collaboration, and support in the past few years. Sujata had helped my overall development as a researcher. She had always shown confidence in my work even when things were not looking too great (multiple paper rejections), which served as a great morale booster. While I haven't had the luxury to collaborate closely with Bryan and Lujo, I am grateful for their time and feedback. Bryan has always been ready to offer his time to read my research drafts and let me be a teaching assistant for the introduction to computer security class. Lujo helped me improve this thesis with his insightful questions and constructive feedback. Mike, whom I first met as a first-year Ph.D. student to discuss my research project (Nomad), only had words of encouragement whenever I interacted with him. I continue to be amazed by his attention to technical details and insightful questions that trigger deep thoughts for days.

I am also fortunate to have worked with many additional collaborators: Yves Bieri, Jeffrey Helt, Limin Jia, Ruben Martins, Rahul Muthoo, Rahul Anand Sharma, Jono Spring, Sahil Uppal, Wenfei Wu, Mihalis Yannakakis, Yucheng Yin, Yifei Yuan, and Ying Zhang. I benefited from discussing ideas and designs of Alembic (Chapter 3) with Ying, Wenfei, and Mihalis. Jeff gave me tremendous help in building and debugging Alembic. Yifei contributed to the theoretical proofs for Alembic and taught me how to formalize my insights. Ruben was always ready to offer his time and help whenever I was stuck with Z3 when building Pryde (Chapter 4). Yucheng helped me implement and run large-scale measurements, and Rahul helped me with the analysis framework for AmpMap (Chapter 5). I am grateful for all my collaborators and for the opportunity to learn from each one of them.

As a Ph.D. student, I spent a significant amount of my time at CyLab. I would like to thank the members of CyLab for providing a welcoming and collaborative environment. I am grateful to the remaining CyLab faculty for their advice, our research group members for many discussions, and the faculty and students of the Tuesday systems seminar for their valuable feedback on my talks. The administrative aspects of the Ph.D. programs were easy thanks to the staff members at CyLab and the Department of Electrical and Computer Engineering: Brigette Bernagozzi, Toni Fox, Karen Lindenfelser, Chelsea Mendenhall, Jamie Scanlon, and Nathan Snizaski. Also, I thank Chad Dougherty for managing and helping with our Beluga lab

clusters.

I want to thank my friends I met during my Ph.D. journey and friends outside of graduate school. I thank them for providing me with the much-needed distraction when I needed it the most and tolerating me during my many deadlines.

I saved the most important for the last. I am indebted to my family for their unconditional support, prayers, and encouragements. My parents and grandparents gave me the courage to pursue my dream and have been always supportive of my decision. My sister has always been my best friend as long as I can remember. Lastly, I want to thank my Lord and Savior, Jesus Christ, for your grace and blessings throughout my Ph.D. journey. *"In their hearts humans plan their course, but the Lord establishes their steps." (Proverbs 16:9)*

The work presented in this thesis was supported in part by NSF awards CNS-1440065, CNS-1552481, and generously supported with funds from the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. The work presented in this thesis was sponsored in part by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, 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 here on.

Thesis Committee Members:

Vyas Sekar (Chair)
Sujata Banerjee
Lujo Bauer
Bryan Parno
Michael K. Reiter

Abstract

Modern networks are exploding with an increasing array of diverse network functions (e.g., network firewalls) and services (e.g., public servers). Despite their critical role in our modern infrastructure, they remain largely *black-box* in nature, given that they are proprietary or configured and deployed by third parties. This black-box nature makes it fundamentally difficult for operators and Internet security experts to reason about security implications and correctness of these functions and services. Unfortunately, this lack of understanding and analysis leaves gaps for high-impact network attacks exploiting their insecurities and network outages.

This dissertation aims to bridge this operational gap by building techniques to automatically analyze the behavior and vulnerabilities of these network devices and services. Specifically, we design techniques to (1) automatically infer high-fidelity models to enable accurate testing and verification, and (2) identify new avenues for potential abuse against network functions and services. Given that we only have black-box access, our techniques do not require access to the code or binary for instrumentation. However, designing these techniques is challenging. First, we need to reason about their behavior under a large traffic space and possible configurations. Second, they may exhibit complicated (hidden) behaviors. Our high-level approach in building these tools is to leverage *structural properties* inherent to black-boxes and their input and configuration space. This insight allows us to reduce the relevant search space and efficiently search over the relevant part of the search space.

The key contributions of this thesis are three concrete tools. First, is Alembic, a tool that can automatically synthesize high-fidelity models of stateful network functions, for accurate testing and verification workflow. Second, is Pryde, a tool which provides operators with capabilities for identifying subtle evasion vulnerabilities in stateful firewalls. Lastly, is AmpMap, a low-footprint measurement framework that can systematically quantify the amplification risk against black-box protocol servers at scale. In presenting each of these tools, we highlight how each tool (1) uncovered unexpected behavior and new security vulnerabilities, and (2) highlighted significant variability in the behavior and security implications of these black-boxes across vendors and implementations. Our findings and results affirm the need for automatic tools to analyze the behaviors for black-box functions and services to properly understand their security implications.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern networks (e.g., enterprise, cloud, wide-area) are abounding with an increasing array of diverse network functions [170] (e.g., firewalls, load balancers) and services [95, 96] (e.g., public servers, applications). They are deployed for security, performance, and efficiency to meet the sophisticated objectives of our workloads and needs. For instance, these functions and services may monitor and control network traffic for security purposes (i.e., firewalls [51], intrusion detection systems [36]), distribute traffic across servers to increase capacity and reliability (i.e., load balancers [38]), and translate human-readable domain names such as `cmu.edu` to numeric IP addresses to help browsers load Internet resources (i.e., public DNS servers [37]).

However, these network functions and services are *black-box* in nature, given that these network functions can be proprietary and public services are configured and deployed by third parties. In these black-box settings, network operators deploying these functions or Internet security experts assessing vulnerabilities of these services may not have full knowledge about their internal workings and lack access to the source code. Despite their critical roles in various network infrastructures, it is fundamentally difficult for operators and Internet security experts to reason about their security implications and correctness of these network functions and services. Unfortunately, this lack of understanding and hence, a lack of systematic analysis leave gaps for high-impact network attacks that exploit insecurities in these functions and services; e.g., many

recent high-impact distributed denial of service attacks (DDoS) attacks have exploited vulnerabilities in public servers [24, 63]. Further, a failure to correctly test and verify networks can result in high-cost network outages [20, 63] and significant performance degradation [9]. This thesis aims at bridging this operational gap by building techniques to analyze the behavior and security vulnerabilities of these black-box functions and services.

1.1 Motivating Scenarios

We now discuss three motivating scenarios that highlight how lacking appropriate tools for analyzing these black-box functions and services leads to significant operational gaps for network operators and Internet security experts.

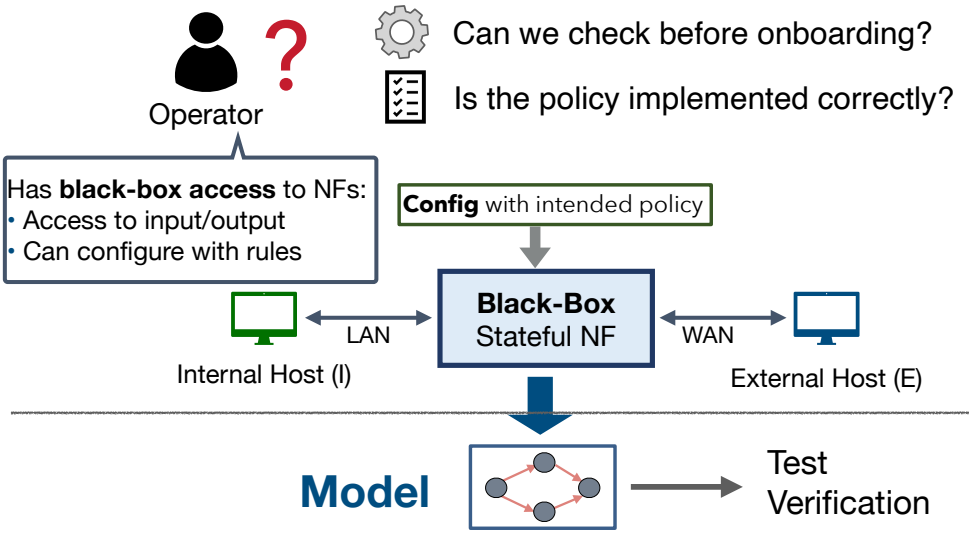
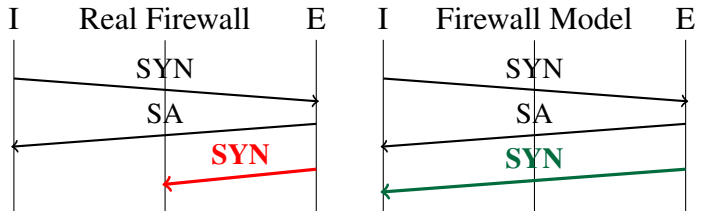


Figure 1.1: A black-box network function in an enterprise network, and an operator who wants to run a testing and verification workflow

S1) Network testing and verification: Modern production networks are composed of black-box network functions (NFs) such as firewalls and load balancers [170]. Given their proprietary natures, operators only have black-box access to configuration interfaces but lack access to the code (Figure 1.1). To help network operators manage and configure these networks and NFs,

there are many efforts in network testing and verification [97, 152, 172] as well as "onboarding" new virtual NFs [141]. However, these tools implicitly assume access to high-fidelity models of these NFs for generating verification proofs and test traffic. However, given a lack of tools that can automatically synthesize these models, operators are forced to rely on their domain expertise to *hand-craft* these NF models. Unfortunately, these hand-crafted models' inaccuracy leads to fundamental correctness issues in these verification and testing tools.



(a) Firewall implementation (b) Handwritten model [97]

Figure 1.2: A concrete example showing how an inaccurate model can affect the correctness of a network testing tool

Consider an operator (in Figure 1.1) who needs to test whether the following intended policy is implemented correctly: "an external host's TCP traffic should only be allowed if a packet belongs to an already established connection initiated from an internal host." To test this policy, the operator runs a testing tool, BUZZ [97], with a hand-written model of an NF configured with this policy. BUZZ flags that the policy has been violated. However, the firewall was implementing the policy correctly. Upon investigating the root cause, an operator found that the hand-crafted model did not correctly reflect the real implementation. As seen in Figure 1.2, a SYN from an internal host on both a handwritten model and a firewall were forwarded, followed by a SYN-ACK (SA) from an external host (E). At this point, the three-way handshake has not been completed from an internal host (I). When E sends a SYN packet, the real firewall drops the SYN packet, but the BUZZ model lets it through. Due to the discrepancy in these outputs, BUZZ flags it as a policy violation when there is no violation (i.e., a *false positive*). Even this simple example highlights how the model's inaccuracies detrimentally affect the utility of this testing tool. We need

to equip operators with the capability of automatically synthesizing NF models to help them run these tasks effectively.

S2) Semantic evasion testing for stateful firewalls: Stateful firewalls (FW) play a critical role in securing network infrastructures in various deployments [42, 48]. These firewalls impose restrictions on undesirable traffic from outside networks. As opposed to simple access control lists, stateful firewalls track the state of individual TCP connections (along with rules) to determine which packets are allowed. A typical policy shown in Figure 1.3 is to drop all external packets that do not belong to an established connection initiated from an internal host. An error in implementing these stateful semantics will have severely detrimental security implications.

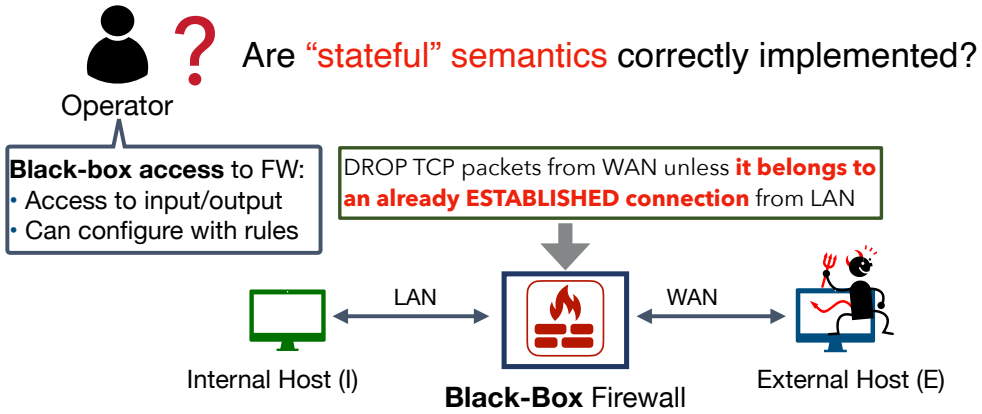
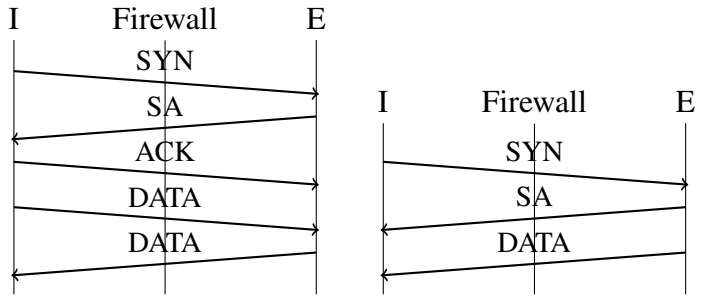


Figure 1.3: A stateful firewall in an enterprise network, and an operator who wants to uncover semantic evasion attacks against this firewall

However, operators deploying these firewalls implicitly assume vendors implement these stateful semantics correctly (specifically, given a lack of necessary tools and having only black-box access). Unfortunately, this is not the case. Specifically, we find a simple *vulnerable* sequence that leads to the firewall forwarding a DATA packet from an untrusted external host (E) to an otherwise unreachable internal host (I). To explain this, we contrast it with a normal (standard) packet sequence (Figure 1.4a). Here, (I) wants to access an external service and initiates a connection setup by sending a TCP SYN to (E). An external service, (E), acknowledges with a SA, followed by an ACK from an internal host. As the TCP three-way handshake is completed,

they can now freely exchange DATA packets. Now, consider this strange sequence (Figure 1.4b), where FW-1 allows a DATA packet from an external host *even before* a three-way handshake has been completed. Here, the firewall is not checking for the last ACK packet of the handshake.



(a) Normal Scenario **(b) Exploitable Scenario**

Figure 1.4: Packet sequences played against a firewall

While this is a simple illustration, the reality is much worse. These semantic errors manifest in so many different ways (e.g., polymorphic variants of this attack, exploiting other TCP aspects). Given the critical roles played by these firewalls, we need to equip operators with the capability to automatically identify evasion vulnerabilities against *black-box* firewalls.

S3) Risk quantification of DDoS amplification attacks: Despite extensive industrial and research efforts (e.g., [6, 8, 148]), distributed denial-of-service (DDoS) attacks still continue to plague the Internet. Recent report [2] suggests that the scale and severity of their impact have risen by nearly 200% only in the past two years itself. Specifically, many recent high-impact DDoS attack attacks rely on *amplification* [24, 31, 32]. In an amplification attack, as shown in Figure 1.5, an attacker spoofs the victim's source IP address and sends carefully-constructed queries (e.g., 60 bytes query) to a public server on the Internet (e.g., DNS, NTP, Memcached servers). This server acts as an *amplifier* and, in turn, sends large responses to the victim; in Figure 1.5, the response has been amplified by 50×. While there are best practices to mitigate these attacks [3, 4, 5], they are unevenly applied. If a source IP address can be spoofed, any stateless protocols (UDP) in which the response is larger than the query can be abused. Further, there still

continue to be many public-facing *black-box* servers that are exploited for amplification.

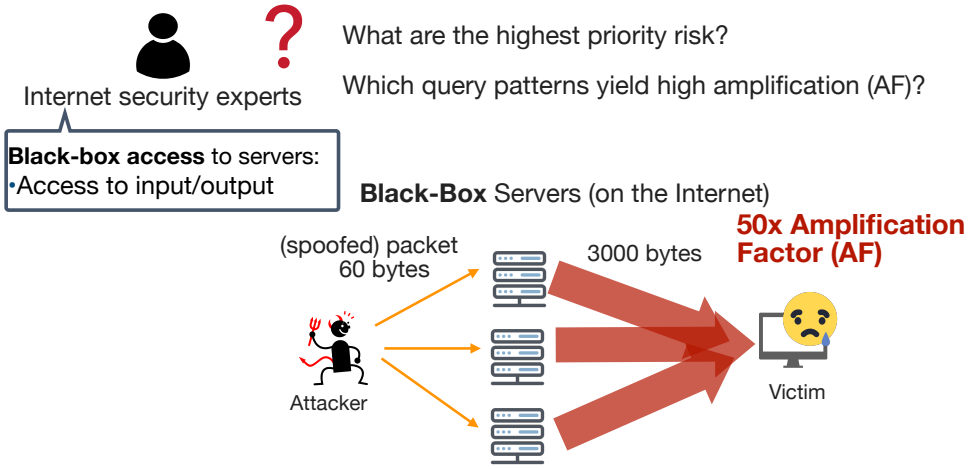


Figure 1.5: A primer on amplification attack, and Internet security experts who want to measure the amplification risk on the Internet

Given these serious threats, Internet security experts need to assess the precise risk to focus their remediation efforts and identify which query patterns yield amplification. However, given that these servers are deployed by third parties, they also only have black-box access to these servers (i.e., ability to send and observe outputs). Existing efforts for measuring the risk simply count the number of servers (e.g., [10]) or focus on only a handful of amplification-inducing patterns (e.g., [85, 175]). Unfortunately, these approaches are fundamentally imprecise; they do not account for the variability of risk across servers and miss many amplification-inducing patterns. In light of the continued threat, we need to equip these security experts with the ability to precisely assess amplification risk. This can inform remediation efforts such as throttling servers, generating signatures, informing protocol changes, and provisioning defenses [10, 14].

Summarizing these scenarios: Having discussed these scenarios, we summarize in Figure 1.6 why we only have black-box access and what having black-box access means for each scenario. Across all scenarios, operators and security experts have access to input and output interfaces of these black boxes. Further, they know the input format of these black boxes (e.g., DNS packet format for S3). However, they lack access to code and binary for instrumentation and do not

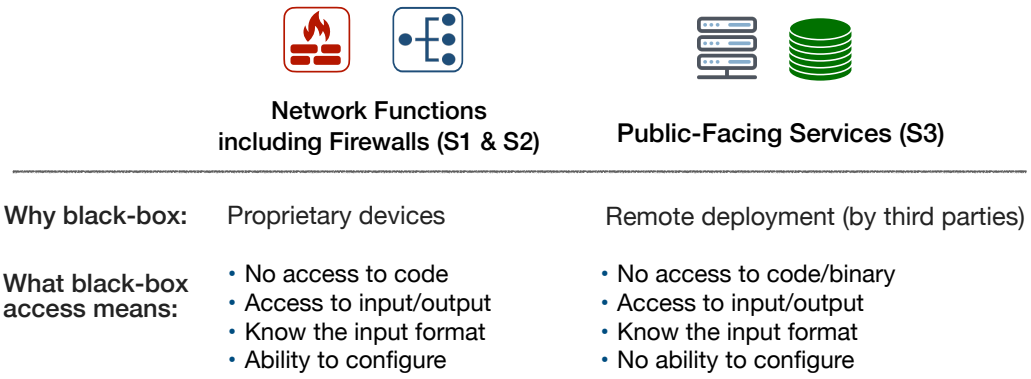


Figure 1.6: Definitions of black-box access for each motivating scenario (Section 1.1)

know the internal workings of these black boxes. Lastly, only operators deploying these NFs (including firewalls) in a local deployment have the ability to configure them.

These motivating scenarios showcase that modern networks invariably incorporate a diverse array of *black-box* functions and services. Hence, we need to equip operators and security experts with the capabilities to reason about the security implications and correctness of them. Depending on the scenario, this involves developing techniques to (a) infer models of these *black boxes* to enable accurate testing and verification (S1), and (b) proactively identify potential attack vectors (S2, S3). However, given only black-box access, we can only infer models using the input and output pairs seen so far. Similarly, we can only identify potential attack vectors if an output packet from a black-box function or service satisfies specific properties of interest (e.g., large response size for an amplification attack). Naturally, behavior and vulnerabilities that are not observable as an output packet (e.g., software bugs) would be outside the scope of these settings.

1.2 A Taxonomy of Alternatives

Having summarized these scenarios highlighting the pain points faced by operators and security experts today, we present a taxonomy of alternatives (Figure 1.7) for analyzing these black boxes. This taxonomy also helps us put our contributions in perspective; the high-level approach taken in this dissertation is specified in bold text in Figure 1.7. We defer a more in-depth description

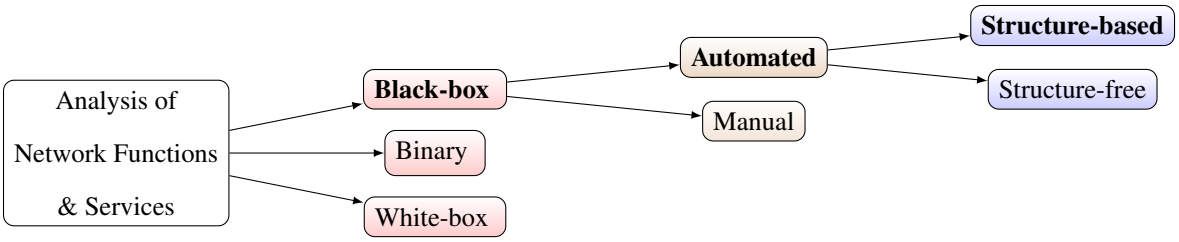


Figure 1.7: A taxonomy of alternatives; the branch marked in bold denotes the approach taken in this dissertation

of prior work and examples for each category to Chapter 2. At the first level, we can classify a specific approach or a technique based on the type of analysis as shown in Figure 1.7:

- *White-box analysis*: White-box analysis (e.g., [155, 182]) uses program analysis techniques to investigate the internal logic using the source code of the system.
- *Binary analysis* : Binary analysis (e.g., [71]) requires having access to the binary code so that an operator can add instrumentation. The process involves reverse-engineering the binary to model data types, and control paths using various techniques.
- *Black-box analysis*: Black-box analysis (e.g., [64, 106, 107]) does not require access to the code or binary for instrumentation. However, it requires access to the system where the user can send inputs and observe outputs. Depending on the problem setting, it may have the ability to generate more packet traces (e.g., [106, 107]) and have some knowledge about the types of inputs (e.g., [64, 171]) such as "this server takes a DNS packet."

Given that it can be impractical for operators to obtain the code or binary for instrumentation, we take a **black-box approach** that requires only black-box access with limited information about the inputs. Among black-box approaches, we can further classify an approach based on: (1) a manual analysis such as manually generating hypothesis and testing them, and (2) an automated analysis. Our dissertation takes an **automated** approach given the diversity and the number of vendors and implementations. Lastly, we can further classify automated approaches based on a specific algorithm or a technique used:

- *Structure-free approach*: This class of approaches (e.g., [10, 69, 175]) does not take into

account of the domain-specific properties. They either send a few probes to black boxes to determine the presence of vulnerabilities (e.g., [10]) or use un-modified fuzzing-based strategies (e.g., random search) to discover exploitable inputs.

- *Structure-based approach*: This class of **structure-based** approaches makes use of certain structural properties and domain-specific insights to optimize and customize the baseline algorithms (e.g., machine learning [110], learning theory [62]). Given the complexity of the black boxes we consider, structure-free approaches were highly ineffective (e.g., unable to discover vulnerabilities). Hence, this thesis takes the **structure-based** approach.

We will expand this taxonomy and discuss a few examples from prior work in Chapter 2.

1.3 Thesis Overview

We start by presenting our thesis statement and then discuss the challenges of designing techniques to support this statement. We then discuss our high-level approach before concluding with in-depth summary of our key technical contributions.

1.3.1 Thesis Statement

Given black-box access to network functions and services, the structural properties of these black boxes and their input and configuration space can be leveraged to automatically analyze the behavior and vulnerabilities of network functions and services. This thesis presents three techniques to: (1) infer high-fidelity models of stateful network function for accurate testing and verification; (2) identify semantic evasion attacks against network firewalls; and (3) identify DDoS amplification vulnerabilities against public services at scale. Each technique is more accurate than prior methods or achieves higher coverage than naive fuzzing techniques.

The thesis is composed of three technical components. Each addresses the pain points identified in our motivation scenarios (from Section 1.1) stemming from lacking appropriate tools to

analyze these black-box functions and services. Note that while the goal of (1) is to design a general technique for inferring models of network functions, (2) and (3) focus on specific problem settings. For instance, (2) focuses on network firewalls, and (3) focuses on DDoS amplification attacks. However, (2) and (3) address high-impact areas in operational security. The insights that we leverage for (2) and (3) can also be applicable if we were to tackle similar problems for other functions and services (more in Section 6.3.2).

1.3.2 Key Challenges

There are three high-level challenges in building techniques to support our thesis statement.

C1. Large input space: The input space of these black boxes can be prohibitively large (e.g., all possible TCP packets). In the case of stateful network functions, the input space also includes all possible *sequences* of packets.

C2. Large configuration space: These black-box functions and services take concrete configurations or settings. Network functions (e.g., firewalls) take a configuration composed of multiple rules, where fields within a rule (e.g., source IP) can take large sets of values (e.g., IP prefixes). Further, different server instances vary in the set of amplification-inducing query patterns. Further, the exact amplification for a given query pattern also differs across server instances. We need to search this configuration or server space efficiently.

C3. Complex internal behavior: The internal logic of a black box can be quite complex. Network functions can modify packet headers or drop packets, making reasoning about their behavior more complex. The structure of amplification a black-box server yields is a complex function of multiple factors (e.g., an input packet, a configuration, data contained in the server).

1.3.3 High-Level Approach

We address the above challenges by using domain-specific insights that leverage **structural properties** of the black-box function or service. These insights help us reduce the large search

space of input and configuration space and systematically explore the search space.

We briefly discuss only a few examples of how we leverage structural properties but defer a full list of structural properties to each applicable chapter. First, in building Alembic for inferring NF models (Chapter 3), we find that considering all possible input packets under an NF configured with all possible configurations is infeasible. Fortunately, we observe that given a rule type (e.g., firewall drop rule), the NF's logical behavior is identical across different values of rule parameters (i.e., drop X vs. drop Y). This structural property allows us to infer symbolic representations once for a given rule type. Second, in building Pryde for uncovering evasion attacks (Chapter 4), we observe that these attacks exploit *subtle* implementation errors specific to a firewall implementation. This property inspired us to use a model-guided approach rather than naively searching over packet sequences. Lastly, in building AmpMap for identifying amplification-inducing patterns (Chapter 5), we observe that amplification-inducing patterns exhibit locality and overlap in the field values. Hence, once we bootstrap a query in a specific operating regime (i.e., gives some amplification), we can search one field at a time instead of searching all N fields. This insight, combined with others (more in Chapter 5), allows us build a low-footprint measurement system while discovering multiple amplification-inducing patterns.

1.3.4 Contributions

We now provide an overview of three key contributions of this thesis.

Model inference for stateful black-box network functions (Chapter 3): We present Alembic [146], a tool that can automatically infer the models of black-box network functions (i.e., firewalls, network address translators). Alembic fills the critical missing piece (i.e., the ability to synthesize NF models) of modern testing and verification tools. In building Alembic, we tackle the challenges of (1) a large NF configuration space containing diverse rule types, and (2) dynamic and stateful NF behaviors. Alembic leverages multiple structural properties to enable a scalable solution. Apart from inferring symbolic models (discussed in Section 1.3.3), we also

observe that NF's behavior is the logical composition of its behavior for individual rules. This property allows us to compose models across rules instead of considering all rules at once. Finally, to infer a stateful behavior model, we identify a natural parallel to a classical algorithm for FSM inference (i.e., L* [62]) and extend it to handle NF-specific behavior.

Results summary: Using Alembic, we inferred high-fidelity models of firewalls, load balancers, and network address translators. We highlight significant variability across NF implementations of a given type (e.g., firewalls, load balancers). We showcase that our high-fidelity models improve the effectiveness of testing and verification tools (i.e., [97, 152]).

Automatic synthesis of evasion vulnerabilities against stateful firewalls (Chapter 4): We present Pryde,[1] a tool which automatically synthesizes semantic evasion vulnerabilities against black-box stateful firewalls. In developing Pryde, we tackle the challenges of a large input space including adversarial inputs and discovering *multiple* subtle attack opportunities. Pryde uses a model-guided approach where it first infers a model (using an extended Alembic) to reason about adversarial scenarios. Then, Pryde efficiently encodes various deployment settings in a model checker and defines custom refinement constraints to discover multiple attack opportunities.

Results summary: We evaluated Pryde on four popular (virtual) firewalls (three commercial-grade and one open-source). We found 294 to 8,200 attack strategies depending on the vendor. Further, post-processing these attacks reveals that the discovered attacks exploit different TCP aspects (e.g., incomplete handshake, simultaneous open). Further, we find that these attacks are highly vendor-specific. We also demonstrate that structure-less approaches are ineffective; i.e., random fuzzing finds 0 to 3 attacks after 15K tries for two complex firewalls.

Risk quantification of DDoS amplification attacks (Chapter 5): We present AmpMap [147], a low-footprint Internet health monitoring service that systematically quantifies amplification risk to inform mitigation efforts. AmpMap equips security experts with the capability to precisely identify (server-specific) patterns exploitable for amplification attacks. In designing AmpMap,

[1] The name is inspired by the Marvel X-men superhero Kitty Pryde who has the ability to walk through walls [46]

we tackle the challenges of a large input/server space and handle a complicated relationship between query (header) values, amplification it induces, and server instances. Specifically, we leverage multiple key structural insights across protocol header and server space. In addition to leveraging the locality structure to search one dimension at a time, we also use appropriate sampling strategies to explore large fields (e.g., 16-bit field) based on our observations. Further, we also observe that while servers vary in their risk, they share some similarities, allowing us to reduce overhead by sharing insights.

Results summary: Using AmpMap, we scanned 10K servers across 6 UDP-based protocols (e.g., DNS, SNMP, NTP). Our measurements revealed new patterns and polymorphic variants of known patterns; e.g., for NTP, while prior work only stresses a known pattern (i.e., MONLIST), we discover additional NTP commands (e.g., get restrict) that can also incur more than 500× amplification. Relying on prior recommendations to block specific queries still leaves open substantial residual risk as they miss many query patterns. Lastly, we demonstrate that our strategy achieves higher coverage across patterns than structure-free approaches (e.g., random search).

1.4 Outline

The rest of this dissertation is organized as follows. Chapter 2 expands on the taxonomy of alternatives and discusses the prior work in this space. Then, Chapters 3, 4, and 5 present the three key components to this dissertations: (1) Alembic automatically infers high-fidelity models of stateful network functions from black-box observations; (2) Pryde automatically synthesizes evasion vulnerabilities against black-box stateful firewalls using a model-guided approach; and (3) AmpMap quantifies the amplification risk at scale and identifies query patterns that lead to large amplification for each black-box server. In presenting each technique, we demonstrate that each technique is more accurate than prior methods or achieves higher coverage than naive fuzzing techniques. Finally, Chapter 6 reflects our findings and learned lessons, discusses the limitations of the proposed solutions, and concludes with future work.

Chapter 2

Related Work

Having laid the vision of this thesis in Chapter 1, we expand on the taxonomy of alternatives (Figure 1.7) for analyzing network functions and services. We first discuss the prior work on analyzing network functions and services in light of this taxonomy (Section 2.1). We then discuss similar efforts in other domains to put this body of work on analyzing network devices and services in perspective (Section 2.2). Note that we will present other domain-specific related work relevant for each contribution (in Chapters 3, 4, and 5) in each respective chapter.

2.1 Analyzing Network Functions and Services

As a high level, we can classify an approach based on the level of access an approach needs: (1) white-box analysis requires the most access by needing the source code; (2) binary analysis[1] does not require source code but requires access to binary for instrumentation; and (3) black-box approach does not need source code or binary for instrumentation but only requires access to input and output interfaces with potential knowledge of input format. Table 2.1 lists the prior

[1] Another possible classification is to use grey-box analysis [142], which include a large body of work on binary analysis. The grey-box analysis uses static information or obtains execution paths, whereas white-box analysis analyzes the source code or an intermediate code

			Examples of Prior Work
White-box analysis			Rossow [165], Cosby et al. [84], Bro [153], Ptacek et al. [160] PIC [155], NFactor [182],CASTAN [156], MAX [130] Dobrescu et al. [92], VigNAT [188], Vigor [187], Gravel [189]
Binary analysis			Rossow [165] , Polyglot [71], MACE [80]
Black-box analysis	**Manual**		INTANG [179], Khattak et al. [127],Lib•erate [136] Qian et al. [161], Chen et al. [77], Joncheray [124]
	Automated	**Structure-free**	CyberGreen [10], Geneva [69]
		Structure-based	Pulsar [106], AutoFuzz [111], HVLearn [171], SFADiff [64] SNAKE [121], Kif [53], SNOOZE [67] Bishop [68], Lin et al [140]

Table 2.1: Examples of prior work on analyzing network functions, services, and protocols mapped to each category in the taxonomy (from Section 1.2)

work in this space and where each work fits into the taxonomy; these examples of prior work were chosen based on the relevance to our motivating scenarios. We now describe each approach in turn.

2.1.1 Type I: White-Box Analysis

White-box analysis analyzes the source of the underlying functions and services to investigate the internal logic. We can further classify white-box analysis based on manual vs. automated.

We start by discussing manual approaches (e.g., [153, 160]). We generally observe that these manual approaches are more common at the earlier stage of the exploration for a given attack or relatively-less-understood functions and services at that given time. For instance, one of the earlier works that exposed amplification vulnerabilities in multiple UDP-based protocols [165] manually analyzed the code (and the binary). Seminal papers on network intrusion detection systems (NIDS) evasion [153, 160], and algorithmic complexity attacks [84] against network

services also used manual code analysis.[2] While these manual approaches have inspired a large body of follow-up work, such approaches will not scale given the diversity of vendors and implementations.

Given the inherent limitations of manual approaches, researchers have developed automated tools (e.g., [130, 155, 182]), which use program analysis techniques (e.g., program slicing, symbolic execution) to examine the system's source code. These approaches have been proven effective for testing (e.g., [155]), generating adversarial inputs (e.g., [156]), modeling (e.g., [182]), and verifying certain properties (e.g., [92, 187, 188, 189]).

For testing purposes, PIC [155] uses symbolic execution for testing interoperability in protocol implementations. For generating adversarial inputs, CASTAN [156] uses symbolic execution to generate adversarial workloads for network functions to cause performance degradation. For modeling purposes, NFactor [182] used program slicing techniques to infer the behavior for network functions, in the form of a match-action table (NFactor [182] shares a similar goal as one of our key contributions, Alembic in Chapter 3. However, Alembic works in a black-box setting and, as a result, can generate models for proprietary NFs).

For verification purposes, prior work (e.g., [92, 187, 188, 189]) has used symbolic execution to verify the correctness of NFs. Earlier work in this space by Dobrescu et al. [92] used symbolic execution to discover low-level programming errors such as memory safety and crash-freedom for stateless NFs written in Click [129]. VigNAT [188] focuses on building a framework for writing a Network Address Translator (NAT) that is guaranteed to be semantically correct and memory-safe. Vigor [187] generalizes VigNAT to handle more NFs and verifies the underlying OS network stack and the packet-processing framework. Both VigNAT [188] and Vigor [187] require developers to write NFs using their frameworks. Both tools require developers to write pseudocode-like low-level specifications. Unlike VigNAT [188] and Vigor [187], Gravel [189] verifies higher-level NF-specific properties (e.g., load balancer's connection persis-

[2] As the authors do not discuss automated techniques and are aware of implementation issues, we can hypothesize that they used manually analyzed the code. In fact, the author of [153] implemented the Bro NIDS.

tency) for Click-based NFs. Gravel, too, requires developers to provide high-level specifications on a symbolic trace of packets. However, these efforts (e.g., [187, 188, 189]) are emerging ideas that may not apply to legacy NFs that are not written in Click [189] or written in other frameworks [187, 188]. Furthermore, writing high-level specifications or synthesizing them may be non-trivial using these tools. Nevertheless, these ideas could be beneficial for building network functions or services that are correct by construction (more in Chapter 6). (We also note that orthogonal efforts test and verify network-wide properties in large networks [97, 152, 172]. As we specifically focus on prior work on analyzing a single network function or service in this chapter, we discuss these orthogonal efforts when we present Alembic in Chapter 3.)

Overall, these white-box approaches can complement our efforts to enable black-box analysis. Specifically, unlike the black-box approach, the white-box approach has access to and can reason about the system's internal logic. However, as we saw from the motivating scenarios in Section 1.1, it can be impractical for network operators and security experts to obtain the system's source code. Further, the assumption that functions and services are written in specific languages or using specific frameworks may not make particular work applicable for legacy network functions and services (which is the focus of this thesis).

2.1.2 Type II: Binary Analysis

Another alternative is to use binary analysis, which involves analyzing and reverse-engineering the binary code. It can be a practical alternative when the source code is not available (i.e., the white-box approach is not feasible), but the binary is available. This analysis works by performing lightweight static analysis or gathering dynamic information about its execution [142].

As many works in the binary analysis focused on building automated tools, we focus our discussion on automated tools (e.g., [71, 80, 183]). For instance, MACE [80] analyzes the code at a binary level to infer the protocol state machine and use the inferred machine to uncover software bugs (e.g., buffer overflow, out-of-bounds reads). Specifically, it uses a combination of symbolic

and concrete executions. Other efforts in this space use binary analysis techniques to infer protocol message formats and specifications (e.g., [71, 83]) or generate fingerprints for malicious servers (e.g., [183]). These works are similar in spirit to the binary analysis techniques in the software domain (e.g., [169, 186]) but have been adapted for network functions and services.

Binary analysis is complementary to our efforts in enabling black-box analysis as it has access to information such as code coverage that the black-box approach does not. However, obtaining the binary code can be impractical in certain scenarios. For instance, to assess amplification risks (i.e., AmpMap in Chapter 5), we need to run vulnerability assessment for services deployed and configured by third parties where we only have access to input-output interfaces.

2.1.3 Type III: Black-Box Analysis

Having described the prior work for both white-box and binary analysis, we now discuss related work in black-box analysis, which is an approach that this thesis takes. Black-box approaches, too, can be classified based on manual vs. automated.

We start by discussing prior work that used manual analysis. For instance, these manual approaches have been applied to identify censorship evasion against nation-wide censorship devices (e.g., [127, 136, 179]), attacks against TCP congestion control (e.g., [124, 167]), or side-channel attacks that hijack TCP connections (e.g., [73, 77, 161]). For instance, INTANG [179] developed a suite of effective hand-crafted evasion strategies against the Great Firewall of China (Note that Pryde in Chapter 4 generates evasion attacks against stateful firewalls). However, INTANG [179] manually generated hypotheses based on domain knowledge. While such manual identification is valuable, these approaches will not scale given that these strategies may need to evolve due to changes in implementations of these network functions and services.

We observe that manual black-box analysis has been useful, especially in the earlier exploration of the behavior or attacks against functions and services. Unfortunately, this approach does not scale given a diversity of vendors and implementations. As such, researchers have de-

veloped automated black-box analysis tools to address this scalability and accuracy challenges inherent to manual approaches. In fact, this dissertation also takes an automated approach to building black-box analysis tools. As mentioned in the taxonomy before (Figure 1.7), we can classify automated black-box analysis based on *structure-free* vs. *structure-based* approaches.

2.1.3.1 Structure-Free Approach

This class of approaches usually either probes network functions and services using already-known probes (e.g., network packets) or uses off-the-shelf fuzzing-based strategies. For instance, prior work on identifying amplification vulnerabilities enumerates the number of servers by checking for open ports (e.g., CyberGreen [10], `openresolver` [13]) or sends previously known attack vectors to servers (i.e., send a handful of known queries for DNS [175]). Unfortunately, these simplistic approaches will not capture the diversity of attack vectors across servers and the inputs. For example, these will misses many other vectors that lead to high amplification. Others use off-the-shelf fuzzing-based strategies to search over the input space. For instance, Geneva [69] recognizes that a body of work (e.g., [136, 179]) that manually identified evasion strategies against nation-wide censors does not scale. Therefore, Geneva leverages a genetics algorithm to build an automated approach. Similar genetics algorithm-based approaches have been applied to other popular protocols (e.g., [138]). While these structure-free approaches (e.g., a genetics algorithm, random search) can be sufficient for certain scenarios, they were highly ineffective for our application context. Furthermore, these structure-free approaches are unaware of the (hidden) internal states which could be useful to discover multiple attacks or highlight relevant behavior for complex black boxes. This motivated us to build structure-based approaches we discuss next.

2.1.3.2 Structure-Based Approach

Structure-based approaches recognize that while pure random-based strategies may work in some scenarios, the probability of finding interesting behavior or uncovering relevant security vulnerabilities is low. Hence, these structure-based approaches use domain-specific insight or certain guidelines (e.g., behavior models) to explore relevant portions of the input space. We further classify these structure-based approaches based on whether an approach tackles a black box that is stateful vs. stateless.

Stateful black boxes: Structure-based approaches that analyze stateful black boxes (e.g., a firewall keeping connection states) explicitly infer the (stateful) behavioral model or use it as a guide to efficiently search over the space. Approaches that infer a model either leverages seminal inference algorithms (e.g., L* algorithm) or statistical methods. For instance, a large body of work (e.g., [64, 79, 80]) leveraged L* algorithm to discover protocol vulnerabilities (e.g., [79, 171]), TCP/IP implementation errors in operating systems (e.g., [101, 102]), and specific attacks (e.g., cross-site scripting) against web-application firewalls [64]. (While Alembic in Chapter 3 also leveraged the L* algorithm to infer NF models, these prior works (e.g., [101, 171]) cannot be used to model NFs. They cannot handle NF-specific challenges, such as handling large configuration space and NF-specific behavior.) Other approaches (e.g., [106]) use statistical methods to infer a model. For instance, PULSAR [106] takes an input of network traffic captures and uses a probabilistic technique [132] to infer a message content and a state machine. Then, PULSAR uses the inferred model to uncover vulnerabilities against protocol implementation (e.g., FTP).

Apart from inferring models, prior work (e.g., [102, 118, 122]) also used model checking [123] or model-based testing [174] efficiently to search over the large search space. Specifically, these methods were successful in identifying security bugs in TCP congestion control (e.g., [122]), TCP/IP implementations (e.g., [102]), and 4G LTE implementation (e.g., [118]). (While the specifics of the challenges and techniques differ, we also leverage a similar model-based approach for Pryde in Chapter 4 to uncover evasion attacks against enterprise stateful

firewalls.) Similarly, a body of prior work (e.g., [53, 67, 121]) uses the specifications or state machine to efficiently fuzz the input traffic, and these approaches are often referred to as stateful or guided fuzzing. However, these fuzzing tools primarily focus on crashes or fatal errors in the program. Further, these assume that they are given the specifications (in contrast to Pryde in Chapter 4, which directly infers these models).

Stateless black boxes: Prior work also has leveraged statistical approaches and machine learning techniques (e.g., GAN) to uncover security implications or behaviors/specifications of stateless network black boxes. For example, they have been used in the context of protocol message inference (e.g., [107, 140, 178]), and identifying exploitable inputs (e.g., [140]).

Given network packet captures, ProDecoder [178] uses statistical methods to infer the protocol-complaint format. Specifically, their key structural insight is to leverage highly skewed frequency distribution in messages to enable accurate protocol message format inference. NEMESYS [107] learns the message format by examining the distribution of changes in the bits throughout the protocol message.

Furthermore, given the rise of machine learning, some have started looking into leveraging machine learning techniques (e.g., GAN or generative adversarial network [110]) to enable these tasks. For instance, Lin et al. [140] demonstrate the early promise of GAN in inferring protocol message format and identifying inputs that can be exploited for attacks. However, these GAN-based approaches are not yet viable to infer a complex state machine (for stateful black boxes). Supporting this would be an exciting avenue for future work.

2.2 Analyzing Other Application Domains

While our thesis focuses on analyzing network functions and services, we discuss similar efforts that analyze other application areas. First, given that network functions or services that we consider implement protocols, we start with discussing prior work on testing and verifying

protocol implementation (Section 2.2.1). Then, we discuss related efforts on analyzing software (Section 2.2.2). Specifically, fuzz testings in the software domain have some of the same goals that we have. Further, our high-level taxonomy (i.e., white-box vs. black-box) is also inspired by classification used in the software domain. Then, while not an extensive list, we conclude by summarizing similar efforts on analyzing other application domains (Section 2.2.3).

2.2.1 Protocol Testing and Verification

We start with protocol testing and verification (e.g., [1, 53, 67, 68, 130, 155]). These works have focused on conformance testing (e.g., [155]), identifying potential attack vectors against protocol implementations (e.g., [130]), and running verification proofs (e.g., [86, 87, 150]).

We first discuss prior work that used white-box approaches (e.g., [130, 150, 155]) that analyze the source code. For instance, PIC [155] applies symbolic execution for checking interoperability in protocol implementations. Kothari et al. [130] applied symbolic execution to uncover manipulation attacks. Musuvathi et al. [150] designed a model checker to verify the Linux TCP implementation against a formal specification. This body of work shares similar ideas and techniques to the white-box approaches used for network functions and services.

Apart from white-box approaches, other prior work in this space (e.g., [1, 53, 67, 68]) used black-box analysis. However, this body of work usually requires either user-specified specifications (e.g., [1]) or stateful models of protocols as inputs (e.g., [53, 67]). For instance, SNOOZE [67] used a state machine to efficiently fuzz the input traffic to identify fatal errors or crashes. (Again, this approach contrasts with Pryde in Chapter 4, which directly infers firewall models to uncover evasion attacks.)

Others have looked into proving security properties; e.g., protocol composition logic (PCL) (e.g., [86, 87, 88, 89, 115]) is a formal logic for stating and proving security properties of network protocols such as SSL/TLS. This formal logic supports reasoning about each step in a protocol, which can be composed to prove complex protocols' properties. While our thesis focuses on

testing or generating inputs to verification tools (Alembic in Chapter 3), proving that certain security properties are met given the black-box implementations of these functions or services would be an exciting future direction (more in Chapter 6).

Overall, while protocol testing and verification share similar goals as this thesis, our thesis differs from this body of work. Specifically, this thesis focuses on modeling network devices or findings attacks against network functions or services instead of finding protocol bugs or checking protocols' correctness. Designing techniques for network functions and services brings additional challenges. For instance, the behavior and security vulnerabilities of these functions and services are not just a function of the underlying protocol implementation. These also depend on configuration settings and other factors (e.g., data contained in the server). Naturally, supporting this thesis statement requires building tools that can efficiently reason about the behavior under a large configuration space and handle NF or server-specific challenges.

2.2.2 Software Analysis and Fuzzing

We now discuss similar efforts in the software domain. One of the first uses of fuzzing is described by Miller et al. [144], and it was applied against standard UNIX utilities. Since then, fuzzing has been remained highly popular due to its simplicity and empirical evidence of its success in discovering software vulnerabilities [142]. At a high level, these fuzz testers repeatedly generate inputs to programs. The goal is to generate syntactically or semantically malformed inputs that trigger particular software behavior [142]. All three types of analysis have been used for analyzing software: (1) white-box (e.g., [72, 108]) that requires the source code, (2) binary (e.g., [109, 186]) that require access to the binary and ability to instrument and collect information; and (3) black-box (e.g., [76, 181]) that require no access to the code or binary for instrumentation. These fuzzers focus on finding inputs that lead to unintended actions (e.g., [158, 169, 181, 186]) such as crashes.

An example of a white-box approach is KLEE [72], one of the popular symbolic execution

implementations. The idea of KLEE is to execute the program with symbolic-valued input and identify inputs that drive the program execution to go down a specific path. KLEE has also been used in network-wide testing (i.e., BUZZ [97]). However, BUZZ [97] had to leverage their domain ideas to make it work in the context of networking. A popular binary fuzzer, American Fuzzy Lop (AFL) [186], mutates inputs and focuses on testing new code paths using evolutionary techniques (also, utilized by Geneva [69] to find evasion attacks against nation-wide censors). An example of a black-box approach is by Woo et al. [181]. This fuzzer designed to maximize the number of unique bugs found for software. Further, with recent attention given to machine learning, a large body of work (e.g., [168, 169]) have focused on using machine learning techniques to improve fuzzing.

While fuzz testers in the software domain are related to this thesis, the goal and the use cases we consider are orthogonal. As a result, the key insights and the techniques differ from prior work in the software domain. For instance, Alembic (Chapter 3) leverages structural properties to enable a general method of inferring high-fidelity NF models. Our approach contrasts to these fuzz testers that directly identify exploitable inputs. The work in this thesis that is closest to fuzz testing is AmpMap (Chapter 5). While AmpMap can be seen as a form of *guided fuzzing* that efficiently uncovers amplification-inducing queries, AmpMap, too, leverages structural properties specific to amplification-inducing patterns to enable our low-footprint solution.

2.2.3 Other Prior Work

Analyzing behavior and uncovering security vulnerabilities have been widely applied in multiple other domains. While not an extensive list, they have been applied in the context of IoT mobile applications (e.g., [75, 91, 139, 149, 151]), IoT devices and protocols (e.g., [98, 99, 117]), and manufacturing domains (e.g., [143, 190]).

For instance, prior work on IoT application analysis (e.g., [75, 91, 139, 149, 151]) seeks to find violations in the application actions. IoTMon [91] extracts interactions between applications

with the device and environment and use this information to identify risky or unsafe behavior. As the application code is readily available, these tools tend to use white-box approaches requiring application code [75, 91, 139, 151] or metadata [149]. Other works in this space focus on findings flaws in smart home devices and protocols (e.g., [98, 117, 164]) or uncovering privacy leaks (e.g., [74, 99]).

Other efforts (e.g., [143, 190]) also have looked into the manufacturing domain in the context of programmable logic controllers (e.g., [143, 190]) and industrial control systems (e.g., smart meters [57], water purification system [78]). TSA [143] runs static symbolic execution on PLC's temporal execution graph. In contrast, VetPLC [190] verifies the PLC implementation at runtime by constructing timed event sequences. ARTINALI [57] mined temporal properties from smart meters to build intrusion detection. Similarly, Chen et al. [78] inferred invariants from data traces to build anomaly detection for a water purification testbed. While these works are related to this dissertation, the goal and the use cases that we consider are orthogonal. As a result, the presented tools in this thesis address additional challenges specific to network functions and services (e.g., a large configuration space).

Chapter 3

Alembic: Automated Black-Box Model Inference for Stateful Network Functions

The Problem: As discussed from one of our motivating scenarios (Section 1.1), modern network testing and verification tools rely on *network functions (NF) models* to create test cases, generate verification proofs, and run compatibility tests. Today, NF models are handcrafted based on manual investigation [97, 172], which is tedious, time-consuming, and inaccurate. As we saw from a simple example (in Section 1.1), using low-fidelity models can affect the correctness and effectiveness of these testing and verification tools (we expand more on this in Section 3.1). Further, these handcrafted models do not capture subtle implementation differences across vendors [97, 125]. Surprisingly, while many efforts from the networking community have focused on building more efficient and general network testing and verification tools (e.g., [97, 152, 172, 185]), not much attention has given to automatically synthesizing models of these NFs to guide this testing and verification workflow.

The Solution: To fill this critical gap, we present Alembic [146] in this chapter. Alembic is a tool that can automatically infer a high-fidelity model of network functions. However, synthesizing these NF models is challenging because: (1) NFs have large state spaces; (2) their state may be

mutated by any incoming packet (i.e., large input space); and (3) in response, the NF may react with any number of diverse and possibly even non-deterministic actions (i.e., complex internal logic). While we also make simplifying assumptions to make the problem tractable, Alembic addresses a scoped portion of this open challenge. Specifically, we focus on modeling NFs where their internal states are mutated by incoming TCP packets, and their actions are restricted to dropping and forwarding packets, possibly with header modification. Our goal is to synthesize high-fidelity NF models in a black-box setting, given only the binary executable, vendor manuals, and a specific configuration with which the NF is to be deployed. We adopt this pragmatic black-box approach as vendors may not be willing to share their source code, even with customers. Even this scoped problem presents significant challenges (Recall the high-level challenges we discussed in Section 1.3.2):

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

To tackle these challenges, we leverage the following key insights (Section 3.2) based on the structural properties inherent to these black-box network functions (NFs) and their input and configuration space:

- **A) Compositional model**: Rather than exhaustively modeling an NF under all possible configurations, we consider the NF's behavior as the logical composition of its behavior for individual rules in a configuration.
- **B) Learning symbolic model**: Configurations consist of different rule types, such as a firewall drop rule, where each type is associated with a different runtime behavior of the NF. For a given type, the logical behavior of the NF is the same across different values of the rule's parameters. Hence, we can learn a *symbolic* model for each rule type rather than exhaustively infer a new model for each possible value.
- **C) Ensemble representation** : Even with the above insights, each rule has a large search space as each rule parameter can take a range of values (e.g., a range of ports). Fortunately, we observe that NF behavior is logically independent for subsets of these ranges. For instance, assume a firewall contains one rule and we know it keeps per-connection state. We can then model this rule using an *ensemble of independent models* by cloning the model learned using a single connection. However, we must then consider how to infer the specific granularity of state tracked by the NF (e.g., per-connection or per-source). We show in Section 3.4 how we can automatically infer this granularity and prove the correctness in Section 3.4.2.
- **D) Finite-state machine (FSM) learning** : FSMs are a natural abstraction to represent stateful NFs [97, 152], and using them allows us to potentially leverage classical algorithms for FSM inference (e.g., L* [62]). But there are practical challenges in directly applying L* here: First, we need to create suitable mappings between logical inputs (i.e., an input alphabet) that L* uses and the real network packets/configurations that NFs take as inputs (Section 3.3). Second, header modifications by NFs make it incompatible with L*, so we need domain-specific ideas to handle such cases (Section 3.5).

Having described the high-level insights, we discuss how they specifically address the challenges: Compositional modeling (Insight A) addresses the large configuration space (C3). Both

symbolic and ensemble representations (Insights B and C) address the large input space (C2) by learning a symbolic model for each rule type and then appropriately cloning it to create an ensemble representation (say for large IP/port ranges). Lastly, extending L* (Insight D) enables us to represent stateful NF behavior (C1 and C4).

Building on these insights, we design and implement Alembic.[1] In the *offline stage*, we infer symbolic FSMs for different rule types as defined by an NF's manual. To concisely represent the internal states of an NF, we extend the L* algorithm [62]. We also leverage our L*-based workflow to infer the state granularity tracked by the NF (e.g., per-connection). Since model synthesis need only be done once per NF, we can afford several tens of hours for this stage. Given a concrete configuration (i.e., a set of rules), the *online stage* uses these symbolic models to construct a concrete model within a few seconds. Specifically, the online stage maps each rule in a configuration to a corresponding symbolic FSM which, coupled with the inferred granularity, is used to create an ensemble of FSMs. The ensemble is logically composed together for each rule to construct the final concrete model for the given configuration. The resulting concrete model can then be used as an input to network testing and verification tools.

Evaluation and Findings: We evaluate Alembic with a combination of synthetic, open-source, and proprietary NFs: PfSense [26], Untangle [34], ProprietaryNF, Click-based NFs [129], and HAProxy [17]. We show that Alembic generates a concrete model for a new configuration in less than 5 seconds, excluding the offline stage. Alembic finds implementation-specific behaviors of NFs that would not be easily discovered otherwise, including some that depart significantly from typical high-level handwritten models (Section 3.7.4). For instance, we discover: (1) in contrast to a common view of a three-way TCP handshake, for some NFs, the SYN packet from an internal host is sufficient for an external host to send any TCP packets; and (2) the FIN-ACK packet does not cause internal NF state transitions leading to the changes in the NF's behavior. Finally, we show that using Alembic-generated models can improve the accuracy of network

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

testing and verification tools (Section 3.7.5).

3.1 Motivation

In this section, we expand on an example from Section 1.1 and elaborate on more examples to highlight how inaccuracies in handwritten NF models may affect the correctness of network verification and testing tools. Figure 3.1 shows an example network, where the operator uses a stateful firewall (FW) to ensure that external hosts (e.g., B1) cannot initiate TCP traffic to internal hosts (e.g., A1). This intent translates to three concrete policies:

- **Policy 1:** To prevent unwanted traffic from entering the network, A1 must establish a connection with B1 before the firewall forwards B1's TCP packets to A1.
- **Policy 2**: When A1 sends a RST or RA (RST-ACK) packet to terminate the connection, the firewall should drop all subsequent packets from B1.
- **Policy 3:** To protect against an attacker sending out-of-window packets to de-synchronize the connection state [179], the firewall should drop or send a RST when it receives packets with out-of-window sequence (seq) or acknowledgment (ack) numbers.

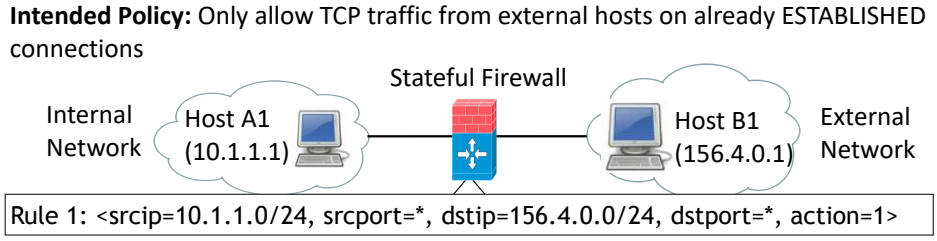


Figure 3.1: Network set-up

To implement these policies, the firewall is configured with the rule shown in Figure 3.1. Since many firewalls implement a default-drop policy, there is no explicit drop rule for packets originating externally. Note we do not need explicit rules for Policy 2 and 3 as they should be performed by the firewall when following the TCP protocol.

To check if the network correctly implements the intended policies, operators use testing and

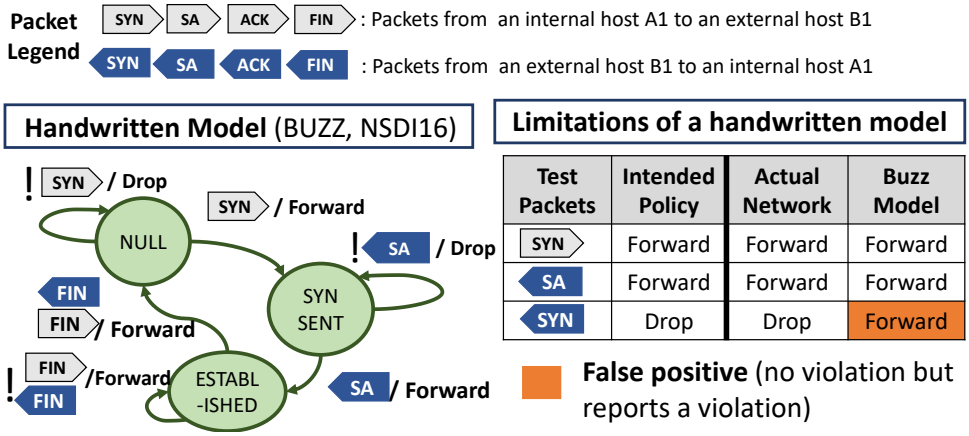


Figure 3.2: A handwritten model of a stateful firewall (FW) which incorrectly reports a policy violation

verification tools [97, 152, 172]. These tools use *NF models* to generate test traffic [97, 173] or to verify intended properties [152]. If these models are inaccurate, the results can have any of the following error types: (1) *false positives*, where the tool reports violations when there is no violation; (2) *false negatives*, where the tool fails to discover violations; or (3) *inability to test or verify* where the tool fails completely because the models are not expressive enough. As an example, consider BUZZ [97], a recently-developed network testing tool. BUZZ uses a model-based testing approach to generate test traffic for checking if the network implements a policy, and the original paper includes several handwritten models. In the remainder of this section, we present three examples of how operators can encounter issues while using the BUZZ tool due to *discrepancies* between handwritten models and NF implementations. Our goal is not to pinpoint limitations of the BUZZ tool but to highlight shortcomings of handwritten models. We find that models from other tools lead to similar problems [152, 172].

To control for NF-specific artifacts (for now), we use two custom, Click-based [129] firewalls that correctly implement the above policies. Figure 3.2 shows the handwritten model of a stateful firewall used in the BUZZ tool [97]. We use the BUZZ firewall model for comparison as it implements a policy similar to our example (i.e., the firewall only forwards packets belonging to a TCP connection initiated by an internal host).

Test case (policy 1): The operator uses the BUZZ tool to generate test traffic and check if TCP packets from B1 can reach A1. Figure 3.2 shows a sample test traffic sequence generated by BUZZ: $\mathrm{SYN}^{\mathrm{Internal}}_{\mathrm{A1}\to\mathrm{B1}}$ (i.e., TCP SYN packet from A1 to B1), $\mathrm{SYN\text{-}ACK}^{\mathrm{External}}_{\mathrm{B1}\to\mathrm{A1}}$, and finally $\mathrm{SYN}^{\mathrm{External}}_{\mathrm{B1}\to\mathrm{A1}}$. Our Click-based firewall drops the last SYN from B1, which matches the policy intent as the TCP handshake did not complete. However, according to the handwritten model, $\mathrm{SYN}^{\mathrm{External}}_{\mathrm{B1}\to\mathrm{A1}}$ is marked as forwarded. Specifically, the model updates the state to ESTABLISHED on receiving a $\mathrm{SYN\text{-}ACK}$ (SA in Figure 3.2) from B1, allowing $\mathrm{SYN}^{\mathrm{External}}_{\mathrm{B1}\to\mathrm{A1}}$ to be forwarded to A1. This discrepancy between the model and the Click-based firewall will be flagged as a policy violation, resulting in a *false positive*.

Test case (policy 2): The operator wants to test if a RST from A1 actually resets the connection state of the firewall. However, as we see in Figure 3.2, the handwritten model only checks for FIN packets but not RST packets to reset the connection state. Hence, the test cases generated by the handwritten model will have discrepancies with the Click-based firewall, resulting in a *false positive* (similar to policy 1).

Test case (policy 3): The operator wants to test whether the firewall correctly handles packets with out-of-window seq and ack numbers. We observe that many firewall vendors enable this feature by default (examples in Section 3.7.4). Unfortunately, the handwritten model is not expressive enough to encode the notion of packets with correct and incorrect seq and ack numbers.

To make matters worse, existing tools (e.g., [97, 152, 172]) assume homogeneous models across vendor implementations for a given NF type. However, we found non-trivial differences in implementations (Section 3.7.4). Further, NF models fed to testing and verification tools need to be aware of the *impact of specific configurations*, which can easily be missed by handwritten models. For instance, the BUZZ firewall model assumes a *default drop* policy from the external interface, which is consistent with many vendors. However, while running model inference using Alembic, we found that one specific NF (Untangle firewall) *allows* packets by default [34]. To implement a default-drop policy in Untangle, we need an explicit drop-all rule, and a model for

Untangle needs to be customized for this configuration.

3.2 Alembic System Overview

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

Preliminaries: We introduce the terminology related to NF configurations, which describe an NF's runtime behavior. A `configuration schema` contains NF rule types. Each `rule type` has various configuration fields, and the data types these fields accept (e.g., "srcip" takes an IPv4 range). Once we specify the concrete values for the fields (concrete values can be wild-card), we obtain a `concrete rule` of the rule type. A `concrete configuration` consists of multiple concrete rules. Figure 3.3 shows an example of a firewall (FW) and a network address translation (NAT) configuration schema and their corresponding concrete configurations. In the NAT Rule type, the outsrcip field denotes the possible output IP values used in address translation.

3.2.1 Problem Formulation

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

Alembic takes five inputs: (1) the NF executable binary, (2) the *configuration schema* (ConfigSchema), (3) the high-level rule *processing semantics* of parsing the configuration (e.g., first match), (4) a list of network interfaces, and (5) the set of input packet types (e.g., TCP SYN or ACK) the model needs to cover. For (1), we assume no visibility into the internal implementation or source code and only have access to its manual describing configuration. For (2), the

ProprietaryNF firewall

ConfigSchema:

Rule type 1 (Accept): ⟨srcip:IPv4 range, srcport:Port range, dstip:IPv4 range, dstport:Port range, action:1 ⟩

Rule type 2 (Deny): ⟨ srcip:IPv4 range, srcport:Port range, dstip:IPv4 range, dstport:Port range, action:0 ⟩

ConcreteConfig:alembic:

Rule 1: ⟨ srcip:10.1.1.1,srcport:*,dstip:156.4.0.1,dstport:*, action:1 ⟩

Rule 2: ⟨ srcip:10.8.0.0/16,srcport:*,dstip:151.0.0.0/8,dstport:*,action:0⟩

PfSense outbound NAT

ConfigSchema:

Rule type 1: ⟨srcip: IPv4 range, srcport: Port range, dstip: IPv4 range, dstport: Port range, outsrcip: IPv4 range, outsrcport: Port range⟩

ConcreteConfig:alembic:

Rule 1: ⟨srcip:10.1.0.0/16,srcport:*,dstip:156.4.0.0/16,dstport:*,outsrcip:126.2.0.0/16,outsrcport=* ⟩

Rule 2: ⟨srcip:10.0.0.0/8,srcport:*,dstip:162.4.0.0/16,dstport:*,outsrcip:192.1.0.0/16,outsrcport=* ⟩

Figure 3.3: Example of a simplified ConfigSchema and ConcreteConfig for a firewall (FW) and a NAT

ConfigSchema is typically already available from vendor documentation. (Alembic requires a one-time, manual effort to translate this documentation into a format compatible with our current workflow). The ConfigSchema in Figure 3.3 assumes we are explicitly given a set of rule types (e.g., accept or deny), where each rule type is associated with a different runtime behavior. In practice, the vendor documentation may only specify a set of fields and their types. For instance, a firewall ConfigSchema provides one rule type with an action field that takes a binary value, in which each value leads to a rule type with different runtime behaviors. We show how we generate a set of all rule types in such a case (Section 3.6). For (3), we assume the rule processing semantics are available from the vendor documentation. Our design can handle any NF that applies a single rule per packet. Our implementation currently supports first-match semantics but can be easily extended to handle others (e.g., last-match). For (4), we need to know a list of interfaces that the NF is configured with. In this work, we assume that we are given two

interfaces (e.g., internal and external-facing interfaces). Lastly, given packet types (5), Alembic will automatically configure each packet type with appropriate field values.

Here, we focus on modeling TCP-relevant behavior for NFs that forward, drop, or modify headers (e.g., firewalls, NATs, and load balancers). We provide default packet types for TCP, but Alembic can be extended with additional packet types. We scope the types of NFs and their actions that Alembic can handle in Section 3.2.3 and discuss how to extend Alembic to handle more complex NFs in Section 6.3.1.

3.2.2 Key Ideas

To highlight our main insights to address challenges *C1* through *C4* from the preamble of this chapter, suppose we want to model an NF with a concrete configuration C_1 composed of N concrete rules $\{R_1 \cdots R_N\}$. Figure 3.4 illustrates our ideas to make this modeling problem tractable.

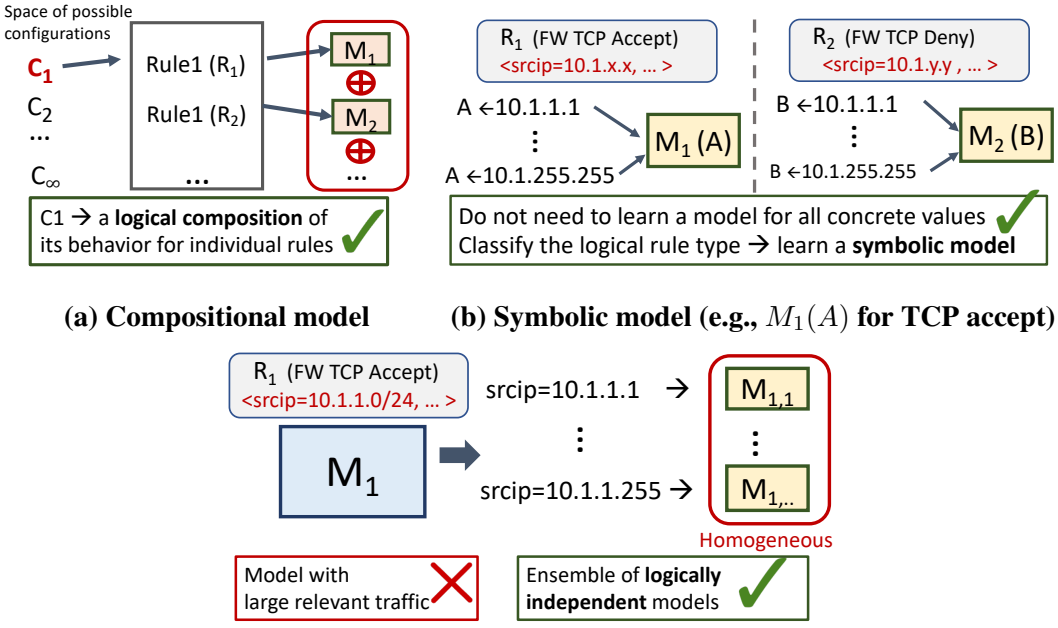


Figure 3.4: Alembic key insights leveraging structural properties

A) Compositional model (Figure 3.4a): The concrete configuration C_1 can be logically *de-*

composed into individual rules. As seen in Figure 3.4a, suppose we have models M_1 for R_1 and M_2 for R_2. Then, we can create a *compositional model* for the NF given the processing semantics defined by the ConfigSchema (e.g., first-match). If the packet matches $Rule_1$, then apply $Model_1$, else if it matches $Rule_2$, then apply $Model_2$. Otherwise, apply $Model_{default}$.

B) Symbolic model (Figure 3.4b): To start, we make two simplifying assumptions, which we relax below: (1) the IP and port fields in a concrete rule take a single value from a range (e.g., 10.1.1.1 for srcip); and (2) the NF keeps per-connection state. Suppose the srcip field in R_1 (Figure 3.4b) takes a single IP from 10.1.0.0/16. It is infeasible to exhaustively infer the model for all possible values. Fortunately, we observe that the logical behavior of the NF for a particular rule type (e.g., firewall accept rule) is *homogeneous* across different values for the IPs and ports in this range. Thus, we can efficiently generate a model by representing each IP and port field in a rule with a symbolic value. Hence, for each logical rule type (e.g., firewall accept rule), we can learn a symbolic model (e.g., $M_1(A)$).

C) Ensemble representation (Figure 3.4c): We relax the assumption that IPs and ports take single values and discuss how we handle ranges within a rule (i.e., R_1 in Figure 3.4b takes a /16 prefix for a srcip). We observe that NF behavior is *logically independent* for subsets of this large traffic space. Consider a stateful firewall that keeps per-connection state. Rather than viewing M_1 as a monolithic model that captures the behavior of all relevant connections, we can view the model as a *collection of independent models*, one per connection (i.e., $M_{1,1}$ for connection 1, $M_{1,2}$ for connection 2, etc.). Combining this idea with B above, we learn a *symbolic model* for each rule type and *logically clone the model* to represent IP and port ranges (henceforth, an ensemble of models). However, to leverage this idea, we need to infer the granularity at which an NF keeps independent states (e.g., per-connection or per-source). We show in Section 3.4 how to automatically infer this.

D) FSM inference: The remaining question is how to represent and infer a symbolic model. Following prior work in stateful network analysis, we adopt the FSM as a natural abstraction [97,

Algorithm 1: NF operational model for processing incoming packets

```
1  Function NF (p, Config c, ProcessingSemantic ps, Map[rule, Map[key, state]] stateMap):
2      poutList = []
3      rule = FINDRULETOAPPLY(p, c, ps); if rule is None then
4          rule = GETDEFAULTRULE( )
5      keyType = GETKEYTYPE(rule)
6      key = EXTRACTHEADER(keyType, p) v
7      FSM = GETMODEL(key, rule) curState = GETSTATE(stateMap, rule, key)
8      poutList, nextState = TRANSITION(FSM, p, curState)
9      UPDATESTATE(stateMap, rule, key, nextState)
10     return poutList
```

152]. To this end, we develop a workflow that leverages L* for FSM inference [62]. At a high-level, given a set of relevant inputs, L* adaptively constructs sequences, probes the blackbox, and infers the FSM. However, directly applying L* for an NF entails significant challenges: First, L* requires the set of inputs a priori. Hence, we need to generate inputs from a large input space, and create suitable mappings between inputs that L* takes and real packets for the NF. Second, L* is not suitable for learning a FSM for a header-modifying NF because it assumes: (1) we know the input alphabet a priori, and (2) the underlying system is deterministic. As an example violation of (2), a NAT may nondeterministically choose the outgoing ports. We leverage a domain-specific idea to extend L* for such cases (Section 3.5).

3.2.3 Operational Model and Limitations

Having described our key insights, we scope the types of NFs for which Alembic is applicable. We use an abstract NF (Algo. 1) to describe how incoming packets are processed (a more detailed description can be found in Section 3.6.2.2). Our goal is to handle NFs with logic stated in Algo. 1.

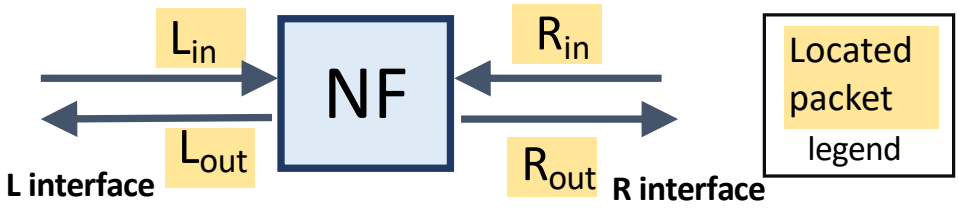


Figure 3.5: An NF with located packets

NF operational model: We start by describing the inputs and outputs of the abstract NF. The NF receives or transmits *a located packet* [126] (i.e., a packet associated with an interface). Figure 3.5 shows a setup for an NF with 4 located packets. The NF is configured with two interfaces, L (e.g., internal) and R (e.g., external). As an example, $\mathrm{L_{in}}$ is a packet entering the NF via L, and $\mathrm{L_{out}}$ is an outgoing packet from the NF via L.

The abstract NF is configured with a *concrete configuration*, composed of a set of *rules*. Each rule maintains a mapping between keys and concrete FSMs. For instance, if the NF uses a per-connection key, then it will keep a concrete FSM for each unique 5-tuple. The concrete FSMs describe the appropriate action (i.e., $\mathrm{L_{out}}$ or $\mathrm{R_{out}}$) for an incoming located packet (i.e., $\mathrm{L_{in}}$ or $\mathrm{R_{in}}$). As shown in Algo. 1, when a located packet arrives, the NF searches the configuration for the correct rule to apply based on the processing semantics. If no rule is found, the NF uses the default (i.e., empty) rule. Then, it uses the relevant packet headers determined by the rule's key to find the concrete FSM and current state associated with that key. Finally, the NF processes the packet according to the FSM and updates the current state (Lines 8 and 9). Alembic aims to synthesize models for NFs following Algo. 1.

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

- Rules in a concrete configuration are independent. For instance, we do not consider NFs that share the same state across different rules. At most one rule in a configuration can be applied to an incoming packet.
- Within a concrete rule, the states across different *keys* (i.e., state granularity tracked by an NF) are *independent*. For a per-connection firewall with a rule that takes IP and port ranges, states

across connections are independent.

- When IPs and ports in a concrete rule take ranges (e.g., ports=*), NFs treat each value in the range *homogeneously* such that we can pick a representative sample and learn a symbolic model (i.e., the symbolic model obtained using port 80 or port 5000 for an outsrcip is identical).

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

- For simplicity, we only consider single-function NFs, excluding cases such as combined NFs processing firewall rules and then NAT rules.
- To make learning tractable, we only look at IP and port modifications. Our implementation does not consider seq/ack numbers, ToS, or other fields (Section 3.3.3). We only handle header modifications for connection-oriented NFs (Section 3.5). (Most header modifying NFs we are aware of are connection-oriented.) We tackle header modification for an NF that initially modifies IP/port of a packet, p1, entering from a particular interface *before* modifying a packet, p2 (that belongs to the same connection as p1) entering from the other interface. Lastly, we cannot infer context-sensitive relations such as how the modified IP or port (e.g., NAT ports) is chosen.
- We do not explicitly model temporal effects, such as connection timeouts. When we inject input packets into the NF, we collect outputs for Δ_{wait} (e.g., 100 ms) before injecting the next input packet. Alembic cannot handle cases where output packets are results of prior input packets (e.g., retries after 1 second).
- We support five types of state granularity: per-connection, per-source (e.g., a scan detector which counts a number of SYN packets), per-destination (e.g., DDoS detector), cross-connection, and stateless.

3.2.4 Alembic Workflow

Having described our key insights and scope, we now present our workflow (Figure 3.6) consisting of two stages:

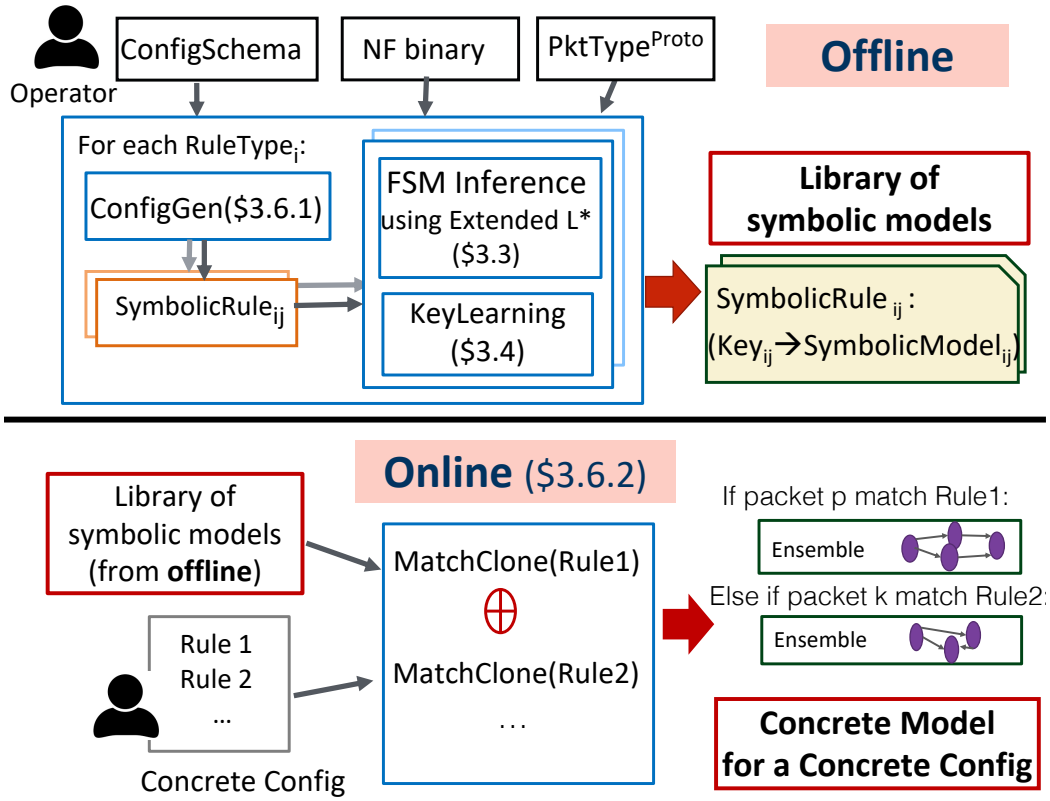


Figure 3.6: Alembic Workflow

Offline stage: From the ConfigSchema, we generate a set of rule types (Section 3.6). Given each rule type, the **ConfigGen** module generates a SymbolicRule, R^{symb}, and a corresponding ConcreteRule. For instance, given a firewall ConfigSchema, it generates two SymbolicRules and ConcreteRules (e.g., firewall accept and deny rule as shown in Figure 3.7).

For each SymbolicRule, we use the **FSMInference** module, which leverages L*-based workflow to infer a symbolic model where IPs and ports are symbolic (Section 3.3) and handles header modifications (Section 3.5). This module uses our version of L* (i.e., **Extended L***). We also design the **KeyLearning** module, which leverages the FSMInference module and infers the state granularity (i.e., key type) tracked by the NF (e.g., per-connection). Using the key type, we can identify the **key**, a set of header field values that identifies logically independent states (e.g., a 5-tuple for per-connection NF). The offline stage produces a set of symbolic models, mapping each SymbolicRule to a symbolic model and its key type.

Firewall TCP Accept Rule
R_1^{symb} : ⟨src:A,srcport:Ap1,dst:B,dstport:Bp1, action:1⟩
R_1^{conc}: ⟨src:10.1.1.1,srcport:2000,dst:156.4.0.1,dstport:5000,action:1⟩
Firewall TCP Deny Rule
R_2^{symb}: ⟨src:A,srcport:Ap1,dst:B,dstport:Bp1, action:0⟩
R_2^{conc}: ⟨src:10.1.1.1,srcport:2000,dst:156.4.0.1,dstport:5000,action:0⟩

Figure 3.7: SymbolicRules and ConcreteRules for a Firewall

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

Roadmap: In the interest of clarity, Section 3.3 describes the FSMInference module of Alembic for a given SymbolicRule with the following simplifying assumptions: NFs keep per-connection state and do not modify headers. In subsequent sections, we relax these assumptions and show how we infer the state granularity (Section 3.4) and handle header-modifying NFs (Section 3.5). Section 3.6 discusses how we generate a set of rule types and the corresponding SymbolicRule and the Alembic online stage.

3.3 Extended L* for FSM Inference

We now present the FSMInference module, which leverages the Extended L* for inferring a symbolic model given a SymbolicRule, R^{symb} (e.g., in Figure 3.7). Recall that we are also given a corresponding ConcreteRule, R^{conc}, to configure the NF. For clarity, we start with two simplifying assumptions: (1) NFs keep per-connection state, and (2) NFs do not modify packet headers. We relax these assumptions in Section 3.4 and Section 3.5.

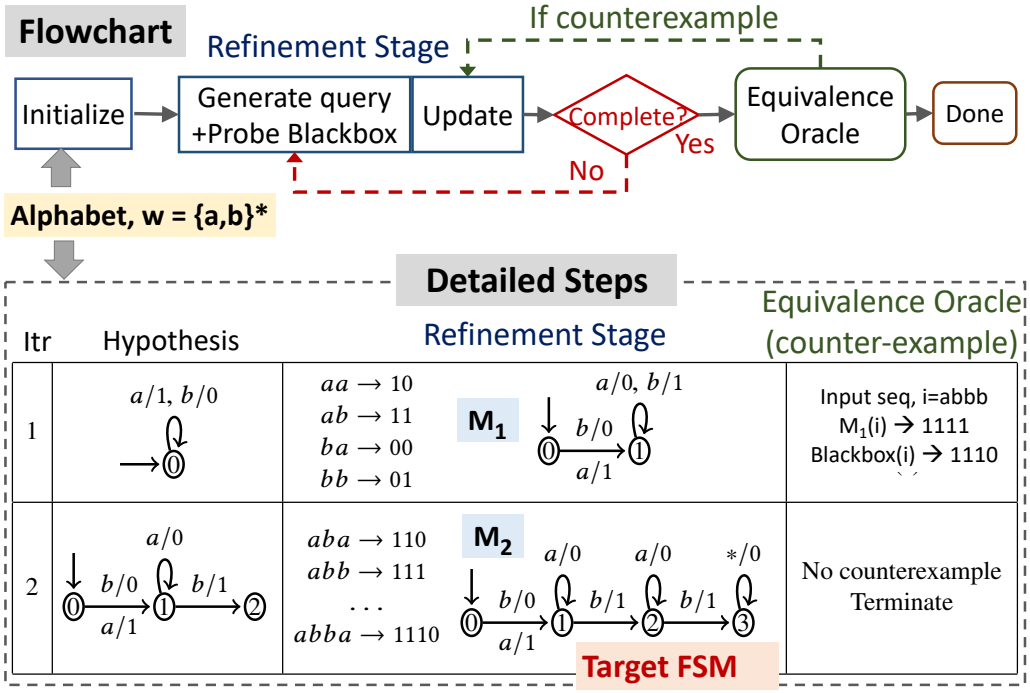


Figure 3.8: L* overview and example

3.3.1 Background on L* Algorithm

Before discussing the challenges of directly applying L*, we provide a high-level description of the L* algorithm [62], which infers a FSM for a given black-box. Given the input alphabet, Σ (e.g., $\{a, b\}$ where a, b are input symbols), L* generates sequences (e.g., a, aa, aba), and probes the black-box, resetting the box between sequences. For each input sequence, L* builds a *hypothesis* FSM consistent with the input-output pairs seen so far. Specifically, it builds a Mealy machine whose outputs are a function of its current state and inputs. As shown in Figure 3.8, L* iteratively refines the hypothesis FSM until it is complete (i.e., the set of probing sequences cover the state space of this hypothesis). After the hypothesis converges, L* queries an Equivalence Oracle (EO), which checks if the inferred FSM is identical to the black-box and provides a counterexample if they are not. If the EO reports that the hypothesis is identical to the black-box, the algorithm terminates. Otherwise, L* uses the counterexample to further refine the hypothesis. The process repeats until the EO reports no counterexamples. L*'s runtime complexity is poly-

nomial in the number of states and transitions of a minimal FSM representing the target FSM as well as the length of the longest counterexample used to refine the hypothesis [62].

Example: Figure 3.8 illustrates an example of the steps in L* for the target FSM shown with $\Sigma = \{a, b\}$. Initially, L* starts with the inputs, a and b, and a single-state FSM. It generates four sequences to refine the model and converges to M_1 as shown. It then queries the EO and finds a counterexample where Blackbox($abbb$)=1110 but $M_1(abbb)$=1111, which is used to update the model. To explore the state space of the new hypothesis, L* generates longer sequences. After this second iteration, the EO finds no counterexamples (as M_2 is identical to the blackbox), and the algorithm terminates.

3.3.2 Challenges in using L* for Black-box NFs

While L* is a natural starting point, there are practical challenges in applying it directly to NFs. We will describe these challenges using Figure 3.9 and discuss our solutions.

1) Generating input alphabet (Section 3.3.3): L* assumes the input alphabet (Σ) is known. As discussed in Section 3.2, we can set Σ for Alembic to be a set of located symbolic packets, which are packets with symbolic IPs and ports associated to interfaces. From now on, when we say packets, we refer to located packets. The main disconnect here is that the NF (i.e., the black-box in the L* workflow) takes in concrete packets and not symbolic packets. Thus, we need to map a symbolic packet to a concrete packet. Two challenges exist here: First, the possible header space for concrete packets is large (i.e., all IPs and ports), and second, the concrete packets need to exercise the internal states of the NF (e.g., trigger the NF behavior).

2) Classifying output packets (Section 3.3.4): Next, for each symbolic packet suggested by L*, we need to map it to an NF action. The practical challenge is that NFs may require an unpredictable delay. If we assume a processing delay that is too short and classify the action as a drop, we might learn a spurious model. While a delay that is too long will lead to our inferences taking a long time. Thus, we need a robust way to map an input to the observed output.

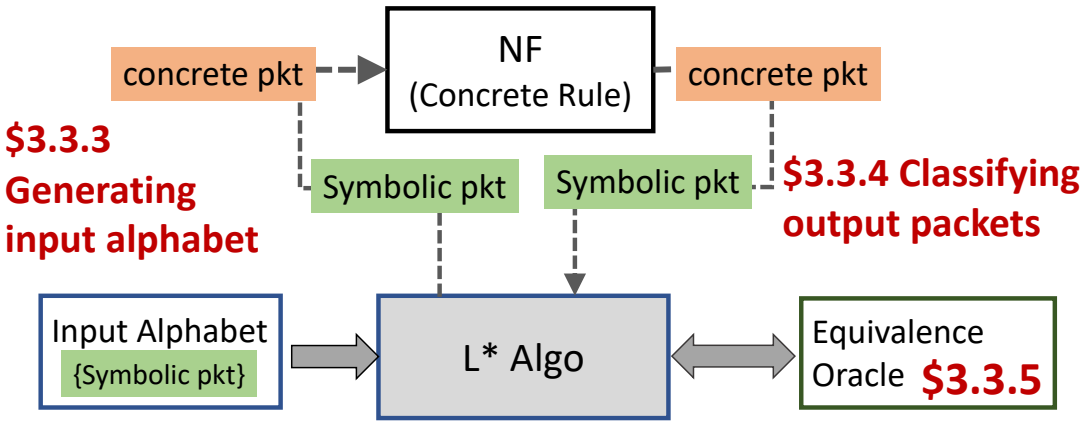


Figure 3.9: Key challenges in adopting the L* workflow for NF model inference

3) Building an equivalence oracle (Section 3.3.5): L* assumes access to an EO (Figure 3.9). In cases where we do not have access to the ground truth, we can only approximate the oracle via input-output observations. There are two practical issues. First, existing approaches (e.g., [81, 105]) to building an EO generate a large number of equivalence queries, creating a scalability bottleneck. Second, different approaches for building an EO may affect the soundness of Alembic (Section 3.3.5).

3.3.3 Generating Input Alphabet

We now describe how we generate a set of *located symbolic packets* for the input alphabet and how we map each located symbolic packet to a concrete packet. As discussed in Section 3.2, we are given the representative packet types of interest $\mathrm{PktType}^{\mathrm{Proto}}$ (e.g., TCP handshake) as an input.

To illustrate these challenges, consider two straw-man solutions that generate packets for: (1) every possible combinations of header fields, and (2) randomly generated header fields. (1) is prohibitively expensive, and (2) may not exercise the relevant stateful behaviors. Our idea is to use the symbolic and concrete rules to identify relevant header fields and their values. Specifically, we observe that the header fields and their values (e.g., IP-port) in R^{conc} will trigger relevant NF behaviors. Thus, we generate all combinations of these relevant IP-port pairs using

their concrete values from R^{conc}. Using a pair of R_1^{symb} and R_1^{conc} as an example (Figure 3.7), we identify A=10.1.1.1 as a possible candidate for both source and destination IPs across *all interfaces* (i.e., A can be a source or destination IP on packets entering from internal or external interfaces). We consider all interfaces, as a packet entering different interfaces can be treated differently.

We also consider the scenario where the packet does not match any rules. One approach is to pick concrete header values that do not appear in the concrete rule and generate a corresponding symbolic packet (e.g., not A=12.1.0.1). However, this would double the size of Σ. Instead, we leverage our insight regarding the compositional behavior of NFs and view this as composing the action with the *default* behavior of the NF when no concrete rule is installed. We separately infer a model, $M_{default}$, with an empty configuration (e.g., a firewall without any rules). (We acknowledge an assumption that rule matching is correctly implemented by the NF. If the NF has a rule for src=A and dst=B but a buggy implementation that matches A' and B', we will not uncover this behavior.)

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

the external interface.

3.3.4 Classifying Output Packets

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

3.3.5 Building an Equivalence Oracle

Building an efficient oracle is difficult with just black-box access [81, 105]. Any EO will be incomplete as it cannot generate all sequences. Our goal is to achieve soundness with respect to the generated Σ without sacrificing scalability.

We tested three standard approaches for generating EOs that LearnLib [162], an open-source tool for FSM learning, supports: (1) *Complete Oracle* (CO), which exhaustively searches sequences to a specified length; (2) *Random Oracle* (RO), which randomly generates sequences; and (3) *Partial W-method* (*Wp-method*) [105], which takes d as an input parameter which is an upper bound on the number of additional states from its current estimate at each iteration. (In

practice, the number of states can grow by $> d$ at each iteration.) We discarded the CO as it simply performs an exhaustive search and the RO as it is not systematic in exploring the state space. Instead, we use the Wp-method, a variant of the W-method [81] that uses fewer test sequences without sacrificing W-method's coverage guarantees. Briefly, the W-method uses a characterization set, the W-set, which is a set of sequences that distinguish every pair of states in the hypothesis FSM. The W-method searches for new states that are within d additional inputs of the current hypothesis and uses the W-set to confirm the new states. In theory, one can set d to be large but increases the runtime by a factor of $|\Sigma|^d$. For this reason, we set $d = 1$ in Alembic. Alembic can only discover additional NF states that are discoverable by the Wp-method with $d = 1$; i.e., Alembic with Wp-method ($d = 1$) is sound. Even with $d = 1$, Alembic synthesizes models that are more expressive than many handwritten models and discovers implementation-specific differences (Section 3.7).

Distributed learning: Both L* and Wp-method for $d = 1$ are polynomial in runtime. However, the Wp-method is the bottleneck as the number of sequences generated by Wp-method is approximately $|\Sigma|$ factor higher than that of the L*. Fortunately, the equivalence queries can be parallelized. In our system implementation (Section 3.7), we run equivalence queries in parallel across multiple workers until we find a counterexample. Using this technique, we can significantly reduce the time for learning a complex behavioral models (Section 3.7.3).

3.4 KeyLearning: Learning State Granularity

Thus far, we assumed that the NF maintains per-connection state. We now relax this assumption and show how we tackle NFs that maintains other key types (e.g., per-source). Specifically, we implement a KeyLearning module. Given a SymbolicRule, the module outputs the **key type**, a set of header fields that identify a relevant model in an ensemble representation. Note that here we still assume that the NF does not modify packet headers, which we will relax next in Section

3.5. We start with discussing the high-level intuition and workflow of a KeyLearning (Section 3.4.1) module and formally prove the correctness of this approach (Section 3.4.2).

3.4.1 Intuition and Workflow

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

The KeyLearning algorithm is composed of test cases to distinguish between different key types. As a concrete example of a test case, suppose we have a SymbolicRule, which takes $\langle$srcip=A, dstip=B$\rangle$ where A and B are ranges of IPs (e.g., A=10.1.0.0/16 and B=156.4.0.0/16). First, we infer two models with two separate ConcreteRules, where we configure each IP using a concrete singleton (e.g., R_1^{conc}, with $\langle \text{srcip=}10.1.1.1, \text{dstip=}156.4.0.1\rangle$ to learn $Model_1$, and R_2^{conc} with $\langle \text{srcip=}10.1.1.1, \text{dstip=}156.4.0.2\rangle$ to learn $Model_2$). Note that these two have the *same srcip*. We leverage the FSMInference module in Section 3.3. We first generate Σ_1 for R_1^{conc} and use the FSMInference in Section 3.3 to obtain $Model_1$, and then repeat for $Model_2$. Assuming these models are independent, we run a logical *FSM composition* operation to construct $Model_{composite}$ (Def.7 in Section 3.4.2). This is what the hypothetical model will be if these two connections are *independent*. As a second step, we now learn a joint model $Model_{joint}$, where we combine input alphabets from both connections. Specifically, we configure a $ConcreteRule$,

where the dstip takes a range of IPs (e.g., 156.4.0.1-156.4.0.2).

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

Test Cases (diff means different)

	Stateless	Per-conn	Per-src	Per-dst	Cross-conn
Test 1 (diff src, diff dst)	N	N	N	N	**Y**
Test 2 (same src, diff dst)	N	N	**Y**	N	**Y**
Test 3 (diff src, same dst)	N	N	N	**Y**	**Y**

Decision Tree

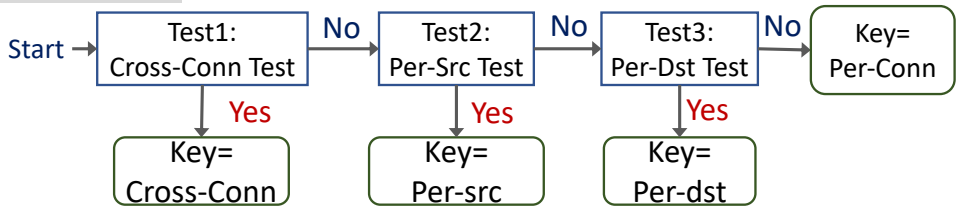


Figure 3.10: KeyLearning Decision Tree

Inference Algorithm: Our inference algorithm generalizes the basic test described above. By crafting different ConcreteRules (i.e., changing the overlap on srcip or dstip) and running the equivalence tests between $Model_{composite}$ and $Model_{joint}$ for each case, we create a decision tree to identify the *key type* maintained by the NF, which are: (1) per-connection, (2) per-source (e.g., a scan detector), (3) per-destination, (4) cross-connection, or (5) stateless. Note that the key for a stateless NF is a 5-tuple. We can view a stateless NF as an FSM with a single state, which is identical to each 5-tuple keeping one state.

Figure 3.10 shows the result of test cases for these key types. For instance, Test 1 configures two connections to have different sources and destinations, to check whether the NF keeps cross-connection state. Test 2 configures two connections to have the same sources, but with different destinations. If Test 2 outputs that two connections affect the states relevant for each other, then the NF is maintaining either a cross-connection or per-source state. The decision tree (Figure 3.10) uniquely distinguishes the key and the correctness naturally follows from our carefully constructed test cases.

3.4.2 Correctness Proof

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

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

Definition of keys: To prove the correctness of our KeyLearning algorithm, we first formalize the definition of NF keys. Table 3.1 summarizes the notations we use. Using this notation, the definition of keys is given as follows.

Definition 1 (Cross-connection NF)**.** *A rule r in an NF keeps* cross-connection *state iff for all*

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

Table 3.1: Notations for the KeyLearning correctness proof

NF instances $NF_r^{\langle X,Y\rangle}$, all pairs of connections (a,b) and (c,d) such that $a,c \in X$, $b,d \in Y$, and $(a,b) \neq (c,d)$, there exists a sequence $\sigma \in \Sigma^{(a,b)}$, such that $NF_r^{\langle X,Y\rangle}(\sigma + +(c,d)) \neq NF_r^{\langle X,Y\rangle}((c,d))$.

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

1. *$\forall\ a \in X$ and $b \in Y$, there exits a σ over $\Sigma^{(\{a\},Y)}$, such that $NF_r^{\langle X,Y\rangle}(\sigma + +(a,b)) \neq NF_r^{\langle X,Y\rangle}((a,b))$.*
2. *$\forall\ a \in X$, $b \in Y$, and σ_1, σ_2 over $\Sigma^{(\{a\},Y)}$ such that σ_1 and σ_2 have the same length, $NF_r^{\langle X,Y\rangle}(\sigma_1 + +(a,b)) = NF_r^{\langle X,Y\rangle}(\sigma_2 + +(a,b))$.*
3. *$\forall\ a \in X$, $b \in Y$, and σ over $\Sigma^{(X,Y)}$, $NF_r^{\langle X,Y\rangle}(\sigma + +(a,b)) = NF_r^{\langle X,Y\rangle}(\sigma|_{(a,\text{-})} + +(a,b))$.*

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

1. *$\forall\ a \in X$ and $b \in Y$, there exits a σ over $\Sigma^{(X,\{b\})}$, such that $NF_r^{\langle X,Y\rangle}(\sigma + +(a,b)) \neq NF_r^{\langle X,Y\rangle}((a,b))$.*
2. *$\forall\ a \in X$, $b \in Y$, and σ_1, σ_2 over $\Sigma^{(X,\{b\})}$ such that σ_1 and σ_2 have the same length,*

$NF_r^{\langle X,Y\rangle}(\sigma_1 + +(a,b)) = NF_r^{\langle X,Y\rangle}(\sigma_2 + +(a,b))$.

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

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

1. $\forall\ (a,b) \in X \times Y \cup Y \times X$, *there exits a* σ *over* $\Sigma^{(\{a\},\{b\})} \cup \Sigma^{(\{b\},\{a\})}$, *such that* $NF_r^{\langle X,Y\rangle}(\sigma + +(a,b)) \neq NF_r^{\langle X,Y\rangle}((a,b))$.
2. $\forall\ (a,b) \in X \times Y \cup Y \times X$, *and* σ *over* $\Sigma^{(X,Y)} \cup \Sigma^{(Y,X)}$, $NF_r^{\langle X,Y\rangle}(\sigma + +(a,b)) = NF_r^{\langle X,Y\rangle}(\sigma|_{(a,b),(b,a)} + +(a,b))$.

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

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

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

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

Definition 7 (FSM composition for key learning)**.** *Suppose we are given two FSMs,* $FSM_i = (S_i, \Sigma_i, \Delta_i, \delta_i, s_i^0)$, *where (1)* S_i *is the state space, (2)* Σ_i *is the space of possible input symbols such that* $\Sigma_1 \cap \Sigma_2 = \varnothing$, *(3)* Δ_i *is the set of output symbols, (4)* $\delta_i : S_i \times \Sigma_i \to S_i \times \Delta_i$ *is the transition function, and (5)* $s_i^0 \in S_i$ *is the initial state of* FSM_i. *The composite FSM of* FSM_1 *and* FSM_2 *is* $FSM_{composite} = (S_1 \times S_2, \Sigma_1 \cup \Sigma_2, \Delta_1 \cup \Delta_2, \delta, s_1^0 \times s_2^0)$, *where* $\delta((s_1,s_2),p) = ((s_1',s_2'),p')$ *if and only if 1)* $\delta_1(s_1,p) = (s_1',p')$ *and* $s_2 = s_2'$; *or 2)* $\delta_1(s_2,p) = (s_2',p')$ *and* $s_1 = s_1'$.

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

Theorem 1 (Correctness of KeyLearning)**.** *Figure 3.10 is correct.*

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

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

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

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

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

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

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

space, $A_1(\sigma|_{(a,b)}) = B(\sigma|_{(a,b)})$. Thus, $C(\sigma) = B(\sigma)$. In order words, B and C are equivalent. □

3.5 Handling NF Header Modifications

Now, we extend our FSMInference in Section 3.3 to handle header modifications, such as a NAT rewriting a private IP-port pair to a public IP-port pair. We currently only handle NFs that maintain per-connection state while modifying IPs and ports. We consider two cases of possible header modifications: (1) *static* (e.g., a source NAT modifies a private port to a static public port), and (2) *dynamic* (e.g., a source NAT or LB randomly generates port mappings across resets). We first describe how we handle each case individually, then present our combined workflow to handle both cases. Our workflow does not require knowing a priori that an NF modifies header fields, which field it modifies, or how it modifies packet headers (i.e., static or dynamic).

Static header modifications: Consider a source NAT that deterministically maps a source IP-port pair (e.g., A:Ap1) to a public source IP-port pair (e.g., X:Xp1). To discover the NAT's behavior that rewrites the public IP-port back to the private IP-port, we need to generate a symbolic packet using the public (modified) IP-port (i.e., X:Xp1). However, we may not know the concrete value of X:Xp1 a priori. Hence, we cannot generate a complete set of $|\Sigma|$. Our idea is to first run the inference module (Section 3.3) and check whether a symbolic model has additional symbolic IPs and ports. If so, we append the new IP-port pairs to the Σ and re-run the inference. We repeat this step until the output FSM contains no new IP-port pairs. Given that the static modification maps an IP-port to the same IP-port pair, this approach converges.

Dynamic header modification: The above approach of updating the input alphabet will not converge for NFs that dynamically modify packet headers, however. Consider a NAT that *randomly* picks one of the available ports for the same 5-tuple (e.g., a private IP-port (e.g., A:Ap1 first maps to X:Xp1 but then to X:Xp2 after L* resets the NF). Since L* assumes a deterministic FSM, it will crash as a result of this nondeterminism. Our idea is simple. If L* crashes, then

we identify the IP-port pair that caused the nondeterministic behavior. Next, we *mask* this non-deterministic behavior of the NF from L* by explicitly mapping such IP-port pairs to consistent symbolic values (e.g., Alembic maps $\mathrm{SYN}^{\mathrm{Internal}}_{\mathrm{A\to B}}$ to $\mathrm{SYN}^{\mathrm{Internal}}_{\mathrm{X\to B}}$ regardless of the concrete value of the rewritten source IP). Since the concrete value of X will change across resets, the extended L* uses the most-recently observed concrete value of X when playing sequences.

Combining both cases, we first run the FSMInference module (Section 3.3). If L* completes but discovers new symbols (i.e., static modification), then we re-run the workflow with new symbols. However, if L* crashes due to a nondetermistic FSM (i.e., dynamic modification), we mask the non-deterministic behavior as discussed. After the required modifications are applied, the L* is repeated until it converges. As we only handle modification for per-connection NF, we assume the key is per-connection for an NF that modifies packet headers.

3.6 Handling an Arbitrary Config

We now discuss how we generate a set of SymbolicRules (Section 3.6.1) and then how the *online* stage constructs a concrete model given a concrete configuration (Section 3.6.2).

3.6.1 Generating SymbolicRules

The ConfigGen module generates a set of SymbolicRules. As discussed in Section 3.2.1, the vendor documentation may not clearly give a set of rule types where each type is associated with a different runtime behavior (e.g., firewall accept vs. deny). Suppose the firewall ConfigSchema specifies a rule types as $\langle$srcip, srcport, dstip, dstport, action$\rangle$ where "action" takes a binary value. To obtain a set of logical rule types, we use a set of conservative heuristics. Typically, we observe that fields which take a large set of values (e.g., IPs and ports) demonstrate similar behaviors across values within the set. For fields that only take a small set of values (e.g., action), each value typically carries a distinct run-time behavior. Based on this observation, the ConfigGen module first assigns a new symbol (i.e., A for srcip) to each field that takes a large set of values. Then

for each combination of other small fields (e.g., action), this module generates a SymbolicRule (for each rule type). We also generate a corresponding $ConcreteRules$ by sampling a value for each field. For the example above, ConfigGen generates two rule types, accept and deny.

3.6.2 Alembic Online: Instantiating a Concrete Model

We now describe Alembic's *online* stage, which constructs a concrete model for a given a configuration. The concrete model then uses our operational model (Algo. 1) to model how an NF processes incoming packets. We start with a high-level workflow (Section 3.6.2.1) and then present a more detailed description of how we instantiate an ensemble of FSMs (Section 3.6.2.2).

3.6.2.1 High-level workflow

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

Processing incoming packets: Upon receiving a packet, the NF fetches the corresponding rule in a configuration using the processing semantics (e.g., first-match). The NF then uses the *key* to access the relevant concrete FSM in an ensemble of FSMs and the current state associated with the packet (Line 7 in Algo. 1). Finally, the NF applies the appropriate action and updates the current state associated with that packet.

3.6.2.2 Instantiating an Ensemble of Concrete Models

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

Case 1) NFs that keep per-connection state but do not modify headers: For NFs that do not modify packet headers, we define a key with (A:Ap1, B:Bp1) where A:Ap1 is a srcip-port and B:Bp1 is a dstip-port. Note that the logic for matching a per-connection key is bi-directional. For example, a TCP packet with srcip-port, B:Bp1, and dstip-port, A:Ap1, would also match the key, (A:Ap1, B:Bp1). Then, for each concrete value of the key in a rule, we instantiate a concrete FSM. We posit that our instantiation logic is correct for an input packet type with *all TCP packet* types (e.g., SYN, SYN-ACK, ACK, RST-ACK) for the following reasons:

1. A model learned using one connection from the offline stage represents the ground truth (assuming a perfect Equivalence Oracle).
2. Because we assume each connection is independent and has the same logical behavior (from Section 3.2.3 and Def. 4 in Section 3.4.2), cloning a model learned from one connection to represent other connections does not introduce additional errors.

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

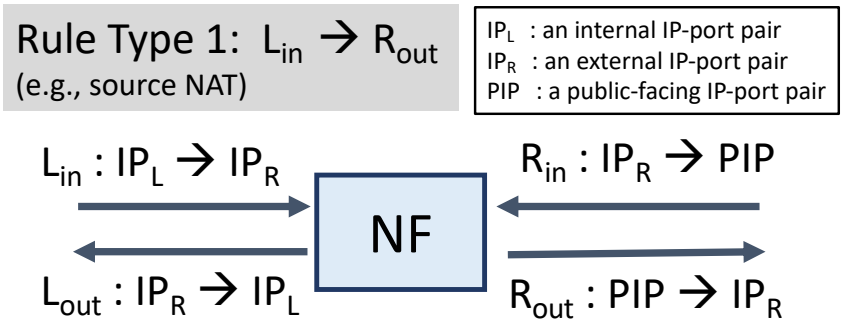


Figure 3.11: NAT example

with two rule types defined in its ConfigSchema.

1. *Rule Type 1* : $\mathrm{L_{in}} \rightarrow \mathrm{R_{out}}$ where the initial modification for a new connections happens for $\mathrm{L_{in}}$ (e.g., modifying the source IP of an internal IP to a public-facing IP).
2. *Rule Type 2* : $\mathrm{R_{in}} \rightarrow \mathrm{L_{out}}$ where the initial modification for a new connections happens for $\mathrm{R_{in}}$ (e.g., port forwarding where the TCP packets with port 8080 entering from the R interface is forwarded to port 80 on the internal server).

For ease of explanation, we first show how we instantiate a concrete model for a model inferred for a *rule type 1* and later describe how we extend our design to handle a *rule type 2*. Figure 3.11 shows the ranges of valid source and destination IPs and ports for located packets for a NAT configured with a concrete rule for a rule type 1 (e.g., a valid ranges for $\mathrm{L_{in}}$ is $\mathrm{IP_L}$ for a srcip pair and $\mathrm{IP_R}$ for a dstip-port pair).

To tackle the challenge above, we introduce two maps to associate an output (or modified) packet's 5-tuple to the corresponding input packet's 5-tuple for both interfaces. Specifically, we use $\mathrm{T_{L\rightarrow R}}$ to map $\mathrm{L_{in}}$ to $\mathrm{R_{out}}$, and $\mathrm{T_{R\rightarrow L}}$ to map $\mathrm{R_{in}}$ to $\mathrm{L_{out}}$ (Algo. 2). Algo. 2 is a detailed description after Line 3 in the operational model (Algo. 1 in Section 3.2). For ease of presentation, we assume we found a rule that matches an incoming packet (Line 3 in Algo. 1).

If an NF receives a packet from the L interface, the algorithm checks whether the packet is a new connection by performing a lookup in the map (in `ForLin`). If the connection does not already exist in the map, we update the $\mathrm{T_{L\rightarrow R}}$ with $(\mathrm{IP_L}, \mathrm{IP_R}) \rightarrow (\mathrm{PIP}, \mathrm{IP_R})$ and $\mathrm{T_{R\rightarrow L}}$ with

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

```
1  Function OnlineForModification(locatedPkt p, Rule r, Map[rule, Map[key, state]] stateMap,
       T_{L→R}, T_{R→L}):
2      if p.interface == L then
3          p_out = FwdDirection(p, r, stateMap, T_{L→R}, T_{R→L})
4      else
5          p_out = ReverseDirection(p, r, stateMap, T_{L→R}, T_{R→L})
6      return p_out
7  Function ForLin(locatedPkt p, Rule r, Map[rule, Map[key, state]] stateMap, T_{L→R}, T_{R→L}):
8      if NewConnection then
9          Update T_{L→R}, T_{R→L}
10     Extract FSM, currentState
11     p_out, nextState ← Get action from the FSM
12     Update currentState with nextState
13     return p_out
14 Function ForRin(locatedPkt p, Rule r, Map[rule, Map[key, state]] stateMap, T_{L→R}, T_{R→L}):
15     if p ∈ T_{R→L} then
16         Extract FSM, currentState
17         p_out, nextState ← Get action from the FSM
18         Update currentState with nextState
19     else
20         Extract default FSM, currentState
21         p_out, nextState ← Get action from default FSM
22         Update currentState with nextState
23     return p_out
```

$(\mathrm{IP_R}, \mathrm{PIP}) \rightarrow (\mathrm{IP_R}, \mathrm{IP_L})$. Then, we extract the corresponding FSM and the current state (or an initial state for a new connection) to apply an appropriate action (i.e., determine p_{out}). If an incoming packet enters from the R interface, we look up the corresponding map, $\mathrm{T_{R\rightarrow L}}$, to

fetch the original IP-port (e.g., $\mathrm{IP_L}$). Then, it uses the key to fetch the corresponding FSM and determine an appropriate action for the incoming packet. If the entry does not exist in the map, our concrete model instead uses the FSM associated with the NF's default behavior. Note that in the case of static header modification, such as a NAT configured with a list of static mappings between internal and external IP-port pairs, we pre-populate $\mathrm{T_{L\rightarrow R}}$ and $\mathrm{T_{R\rightarrow L}}$ with these static mappings. Hence, for an NF that statically modify packet headers, we will not reach Line 20 as these mapping already exist.

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

1. Line 3 must change to call `ForLin` (Line 8) if the packet comes via the R interface.
2. For the corresponding packet coming from the reverse direction (i.e., $\mathrm{L_{in}}$ for rule type 2), we need to perform a look up in $\mathrm{T_{L\rightarrow R}}$ to check if the reverse mapping exists instead of $\mathrm{T_{R\rightarrow L}}$ (i.e., change Line 16).

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

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

after injecting packets and observing outputs).

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

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

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

Case 3: NFs that do not keep per-connection state: We now consider NFs that do not modify packet headers but have keys other than per-connection. Recall the following key types and their corresponding header fields: (1) *Per-source key*, defined by a source IP; (2) *Per-destination key*, defined by a destination IP; (3) *Cross-connection key*, defined by *any* packet (i.e., all IP and ports with the range); and (4) *Stateless key*, defined by srcip-port and dstip-port. Note that we view the stateless NF as keeping a per-connection state but the FSM is always just a single state.

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

specified range.

We *posit* that our instantiation logic outputs a correct model for an input packet type, with all TCP-relevant symbols (i.e., all TCP-relevant symbols as there are no modifications) if the per-source NF adheres to the Def. 2 in Section 3.4.2:

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

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

3.7 Implementation & Evaluation

System Implementation: We implemented Alembic using Java for the extended L* built atop LearnLib [162]. We used C for monitoring NF actions, and Python for the rest. We create packet templates using Scapy [27]. Then, Alembic feeds the output of prior modules into the Extended L*. We re-architected the Learnlib framework to enable distributed learning where queries are distributed to workers via JSON-RPC [23].[2] Our L* implementation tracks the symbol-concrete mapping of IPs and ports to translate between symbolic and concrete packets. The symbolic FSM output is stored in DOT format, which is then consumed by the online stage.

L* assumes that we have the ability to reliably reset the NF between every sequences. For Alembic, we need to reset the connection states. For some NFs, this can be performed using a single command (e.g., `pfctl -k` in PfSense). However, other NFs required that the VM

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

be rebooted (e.g., Untangle). In such cases, we take a snapshot of the initial state of the VM and restore the state to emulate a reset. This does cost up to tens of seconds but is a practical alternative to rebooting.

Experimental Setup: We used Alembic to model a variety of synthetic, open-source, and proprietary NFs. First, we created synthetic NFs using Click [129] to validate the correctness of Alembic. Each Click NF takes an FSM as input and processes packets accordingly, so we know NF's ground-truth FSM. To validate against real NFs, we generated models of PfSense [26] (firewall, static NAT, NAT that randomizes the port mappings, and LB), ProprietaryNF (firewall, static NAT), Untangle [34] (firewall), HAProxy [17] (LB). We now use NAT to refer to a static NAT and a randNAT to refer to a NAT that randomizes the IP-port mappings. Our experiments were performed using CloudLab [94]. We ran PfSense, Untangle, ProprietaryNF, HAProxy, and Click in VMs running on VirtualBox [35]. Recall that Δ_{wait} needs to be customized for each NF. We used Δ_{wait} of 100 for PfSense and Click-based NFs, 250 ms for ProprietaryNF, 200 ms for Untangle, and 300 ms for HAProxy. For NFs that incur unexpected delays (e.g., HAProxy, ProprietaryNF, Untangle), we took a majority vote of 3.

Packet types: We use two TCP packet types. First, the `correct-seq` set consists of standard TCP packets, $\{\mathrm{SYN_C}, \mathrm{SYN\text{-}ACK_C}, \mathrm{ACK_C}, \mathrm{RST\text{-}ACK_C}, \mathrm{FIN\text{-}ACK_C}\}$, where the handling of seq and ack are under-the-hood. Instead of introducing seq and ack numbers in Σ, we introduce additional logic in the Extended L* to track seq and ack of the transmitted packets and rewrite them during the inference to adhere to the correct semantics (i.e., update the ack of $\mathrm{SYN\text{-}ACK_C}$ after we observed an output of $\mathrm{SYN_C}$). (The seq number is incremented by 1 for packets with a SYN or FIN flag set and otherwise, by the data size. T. The ack number for a side of a connection is 1 greater than any received packet's sequence number.) Second, we introduce `combined-seq` set to model the interaction of TCP packets in the presence of out-of-window packets. We extend the correct-seq set with packets with randomly-chosen, incorrect seq and ack values, $\{\mathrm{SYNACK_I}, \mathrm{ACK_I}, \mathrm{RSTACK_I}, \mathrm{FINACK_I}\}$.

3.7.1 Validation using Synthetic NFs

A) Inferring the ground-truth model: We provide Click [129] with a 4-state FSM that describes a stateful firewall that only accepts packets from external hosts after a valid three-way handshake. We also constructed another 18-state FSM describing a similar firewall and a 3-state FSM describing a source NAT. In all three cases, Alembic inferred ground-truth FSMs.

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

Ground Truth	**Test 1**	**Test 2**	**Test 3**	**Result**
Cross-connection	Y			Cross-connection
Per-source	N	Y		Per-source
Per-destination	N	N	Y	Per-destination
Per-connection	N	N	N	Per-connection

Table 3.2: Validating the correctness of KeyLearning using Click-based NFs

C) Validating key learning: We wrote additional Click [129] NFs that track the number of TCP connections based on different keys. We applied the key learning algorithm to each and confirmed it identifies the correct key (Table 3.2).

3.7.2 Correctness with Real NFs

As summarized in Table 3.3, we generated models for PfSense and ProprietaryNF firewalls using both correct-seq and combined-seq sets. For the other NF types, we used only the correct-seq set

	Firewall			staticNAT		randNAT	load balancer (LB)	
PktType	pf	ut	pNF	pf	pNF	pf	pf	hp
correct-seq	●	○	●	●	●	◐	◐	○
combined-seq	●		●					

pf: PfSense, ut: Untangle, pNF: ProprietaryNF, hp: HAProxy

◐: TCP-handshake pkts, $\{\mathrm{SYN_C}$,SYN-$\mathrm{ACK_C}$,$\mathrm{ACK_C}\}$, for both interfaces

○: ◐ set excluding $\mathrm{SYN_C}$ from the *external* interface

Table 3.3: Coverage of models over input packet types

because the firewall models for these NFs already modeled the interaction of TCP packets in the presence of out-of-window packets. For an NF that uses dynamic modification (e.g., randNAT), we cannot correctly instantiate the model in the presence of $\mathrm{RST\text{-}ACK}$ and $\mathrm{FIN\text{-}ACK}$ packets (Section 3.6.2.2). Hence, we only showcased how this NF handles connection establishment. Untangle and HAProxy have SYN retries and spurious resets (i.e., temporal effects) that are beyond our current scope (Section 3.2.3) and could not be disabled. Thus, we again only model how these NFs handle connection establishment. Further, during our attempts to infer models, we discovered these two NFs are connection-terminating, where an external $\mathrm{SYN_C}$ packet interfered with the connection initiation attempt from the internal host, which violates our independence assumption. To make the learning tractable, we removed the $\mathrm{SYN_C}$ from the external interface for these connection-terminating NFs.

Complementary testing methodologies: Since we do not know the ground truth models and thus cannot report the coverage of *code paths* inside the NF, we used three approaches to validate the correctness of our modelsec:alembic: (1) *iperf [21] testing*, generating valid sequences of TCP packets; (2) *fuzz testing*, randomly picking a packet type and a concrete IP and port; and (3) *stress testing*, generating packets by first picking a packet type and selecting concrete IP and port values to activate at least one rule.

For each test run, we generated an arbitrary configuration. For NFs that take multiple rules (e.g., firewall and NAT), we varied the number of rules between 1, 5, 20, and 100. For each con-

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

Table 3.4: Results of stress testing (C for correct-seq, and CI for combined-seq)

crete rule, we randomly sampled a field from the field type defined by the ConfigSchema. We ensured that we picked concrete configurations different from the ones used during the inference (Section 3.3). For firewalls and NATs, the generated configurations were installed on one interface (i.e., internal). Further, as Alembic cannot compose models for multi-function NFs (i.e., a firewall with NAT), we set allow rules on the firewalls when we inferred models for NATs and LBs. For iperf [21] testing, we set up a client and a server and collect the traces on each interface. Because iperf [21] generates a deterministic sequence of packets, we only tested with 1 accept rule. For stress and fuzz testing, we generated sequences of 20, 50, 100, and 300 packets. In each setting, we measured model accuracy by calculating the fraction of packets for which the model produced the exact same output as the NF. Each setting is a combination of the NF vendor and type (e.g., PfSense firewall with the correct-seq set), input packet type (e.g., 300 packets), and the number of rules (e.g., 20 rules).

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

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

Stress testing: We summarize the results in Table 3.4. For many NFs (e.g., ProprietaryNF and PfSense firewalls), we see the lowest accuracy (e.g., 98%) for 1 rule with 300 packets. This is expected because our testing generates a long sequence of packets that the Wp-method with $d = 1$ did not probe. Also, given the same firewall (e.g., PfSense firewall), we observe

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

Table 3.5: Time to infer a symbolic model (h: hours, m: min)

higher accuracy for an NF modeled with the correct-seq set compared to that modeled using the combined-seq set. We confirm that the model learned using the combined-seq set is rather large (> 100 states) resulting from the many ways in which the correct and incorrect packets can interact. Note that ProprietaryNF NAT correct-seq took 49 hours to model and ProprietaryNF firewall combined-seq took 5 days to infer the model. Going back to our earlier requirements that we can afford several tens of hours (i.e., a couple days) for the offline stage, we ran the accuracy testing on an intermediate model inferred after 48 hours, which still achieved high accuracy. We did not perform fuzz or stress testing for Untangle firewall and HAProxy LB. These NFs have temporal effects that result in mis-attribution, which is outside our scope (Section 3.2.3). We see that Alembic achieves high accuracy even with large configurations.

3.7.3 Scalability

We now evaluate the runtime of Alembic's components.

Time to learn symbolic models: For each NF, we report the longest time to model one SymbolicRule as learning can be parallelized across symbolic rules. In all cases, we use 20 servers setup, except for with PfSense random NAT which used one. The results are summarized in Table 3.5. In summary, we achieved our goal of inferring high-fidelity models in less than 48 hours. We find that the runtime depends on: (1) the size of the FSM and $|\Sigma|$, and (2) Alembic or NF-specific details (e.g., rebooting). For (1), as the size of $|\Sigma|$ was double for the combined-seq

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

Table 3.6: Scalability benefits of our design choices

set, it took more than 48 hours to discover 72-state FSM (ProprietaryNF firewall, combined-seq) but less than 26 hours for 79-state with the correct-seq set. For (2), discovering the NAT model ProprietaryNF NAT (correct-seq) took longer than the firewall as the NAT inference ran in two phases (Section 3.5). Lastly, PfSense models take less time to infer as PfSense does not require rebooting, and has shorter Δ_{wait}.

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

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

Scalability benefits of our design choices: The insights to leverage compositional modeling and KeyLearning allow Alembic are critical in achieving reasonable runtime by reducing the size of Σ. Suppose one firewall rule takes a source IP field takes a /16 prefix. Without KeyLearning, we need to infer a model with all 2^{16} connections. Similarly, for a configuration with 20 rules, we need to infer a model with all relevant connections. The top half of the Table 3.6 shows how the runtime drastically increases as we increase the number of connections using a Click-based [129] firewalls from Section 3.7.1 (using just one worker). Further, we measured the runtime as a function of d in Wp-method (bottom of Table 3.6). Using $d = 1$, we were still able to infer the ground truth while reducing the run time. These results demonstrate how reducing the size of $|\Sigma|$ is critical to obtain a reasonable runtime. Lastly, distributed learning helps scalability.

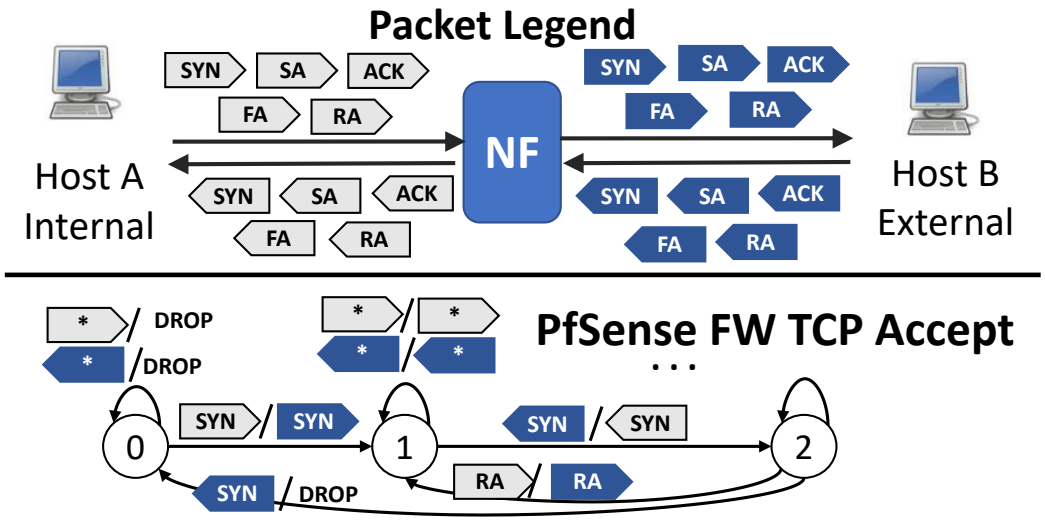


Figure 3.12: The light/dark coloring indicates packets on host A/B's interface, respectively. The figure below shows the 3 states for PfSense firewall (FW) accept rule

The Click-based [129] firewall with 18 states takes 1.6 hours with 1 worker but only 16 minutes with 19 workers (and 1 controller).

3.7.4 Case Studies

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

Firewall (correct-seq): A common view of stateful firewalls in many tools is a three-state abstraction (SYN, SYN-ACK, ACK) of the TCP handshake. Using Alembic, however, we discovered that the reality is significantly more complex. With a single firewall accept rule, the inferred PfSense model (correct-seq) shows that a TCP SYN from an internal host, A, is sufficient for an external host, B, to send any TCP packets (Figure 3.12). Furthermore, FIN-ACK, which signals termination of the connection, does not cause a state transition. We find that ProprietaryNF firewall has 79 states for a firewall accept rule in contrast to 3 states for PfSense firewall. ProprietaryNF, too, does not check for entire three-way handshake (e.g., only SYN, SYN-ACK). We find that the complexity of the FSM (i.e., 79 states) results from the number

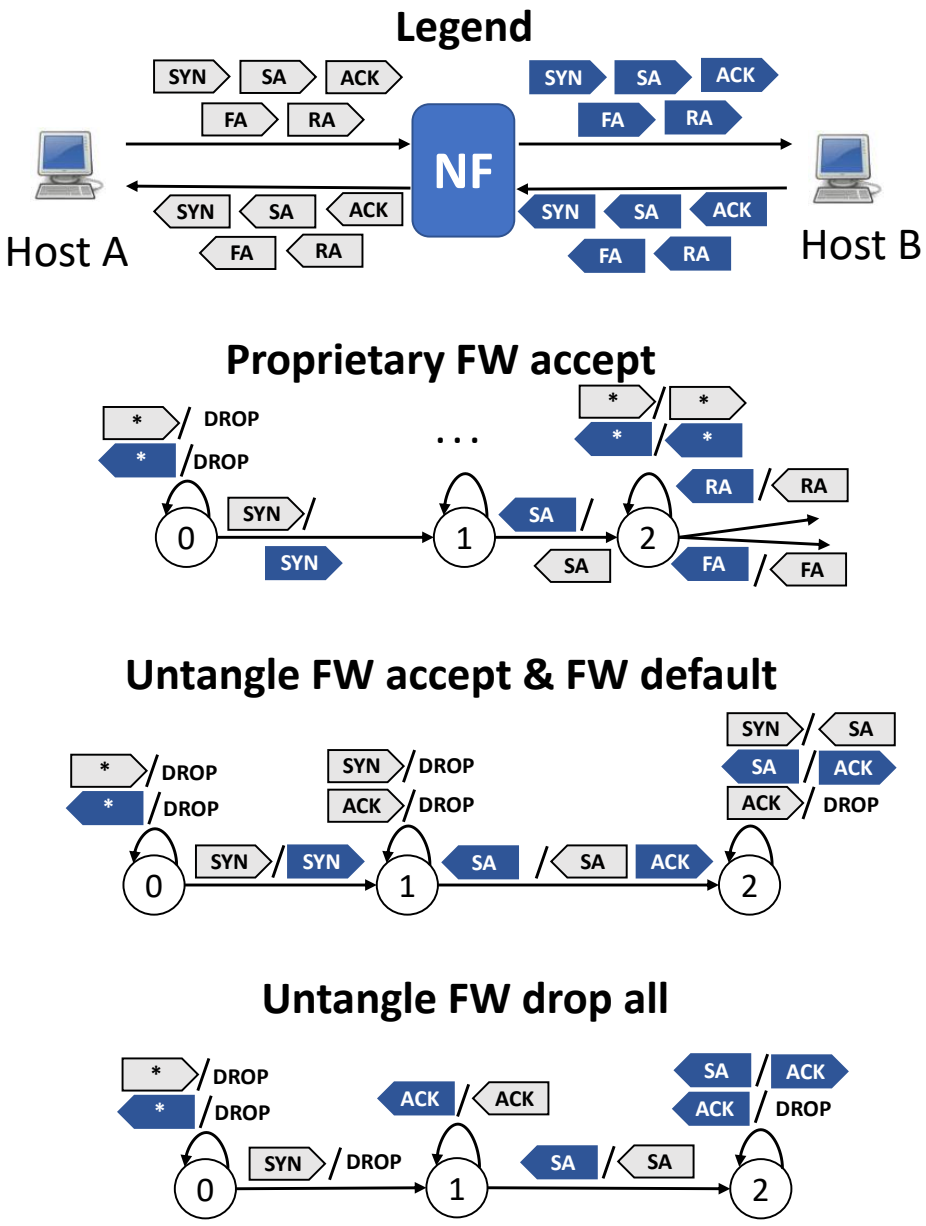


Figure 3.13: Partial FSM for Untangle firewall (FW) accept, drop, default rule, and ProprietaryNF accept rule

of ways in which the two TCP handshakes (from A and B) can interfere with each other. Such behavior could not have been exposed through handwritten models. Untangle firewall actually behaves like a connection-terminating NF (Figure 3.13 shows a partial model). The firewall lets the first SYN from A through, but when B replies with a SYN-ACK, Untangle forwards it but

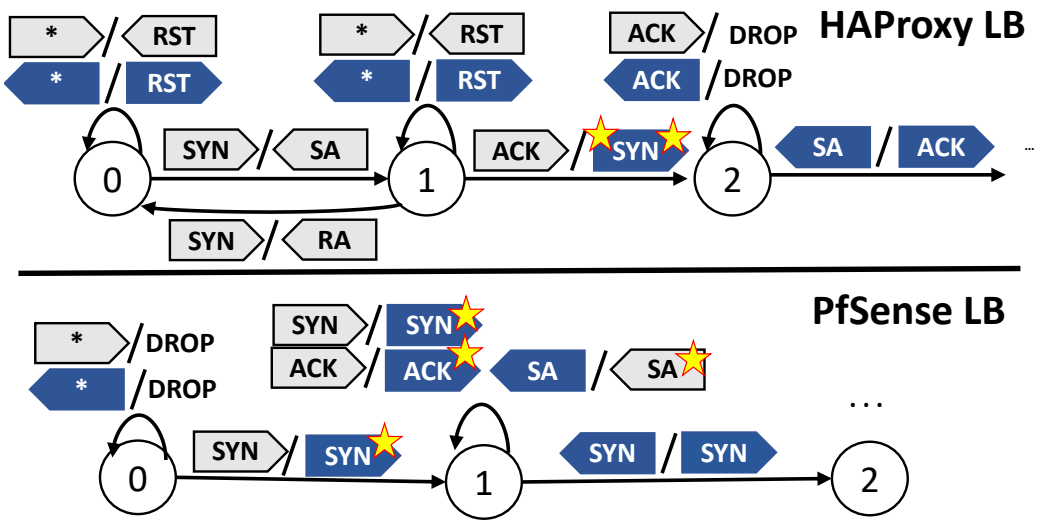


Figure 3.14: First 3 states of the HAProxy and PfSense load balancers (LBs). Stars on head/tail of packets indicate src/dst modification

preemptively replies with an ACK. When the A replies with ACK, Untangle drops it to prevent a duplicate.

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

Load balancer: HAProxy (Figure 3.14) follows the NF's connection-terminating semantics. It completes the TCP handshake with the client before sending packets to the server. After the handshake, the source of outgoing packets is modified to server-facing IP of LB, and destina-

tion is modified to the server (i.e., star on both-ends of TCP SYN in Fig. 3.14). In contrast, PfSense LB behaves like a NAT. When a client sends a SYN to the LB, the destination is modified to the server's IP (i.e., star in state 1 in Figure 3.14). Then, the LB modifies destination of packets from both client and server. We confirmed that PfSense indeed implements load balancing this way [159]. Alembic automatically discovered this without prior assumptions of any connection-termination behavior. Further, the connection-termination semantics of HAProxy differ from those of Untangle firewall. Unlike HAProxy, Untangle lets SYN packets through and preemptively completes the connection with the external host. This is yet another example of non-uniformity across NF implementations.

3.7.5 Implications for Network Testing and Verification

We use two existing tools, BUZZ [97] and VMN [152], to demonstrate how Alembic can aid in network testing and verification. Using a Click-based [129] firewall which adheres policy 1 and 2 (Section 3.1), we compare the test output using: (1) $M_{Alembic}$ inferred using Alembic, and (2) existing M_{hand} for firewall. Using $M_{Alembic}$, BUZZ did not report a violation. Using M_{hand}, BUZZ reported a violation (*false-positive*) as 1 of 6 test traces did not match (trace in Section 3.1). Similarly, for policy 2, BUZZ reported a violation using M_{hand}. The failed test case isec:alembic: $\mathrm{SYN}^{\mathrm{Internal}}_{\mathrm{A\to B}}$, $\mathrm{RST}^{\mathrm{External}}_{\mathrm{B\to A}}$, $\mathrm{RST}^{\mathrm{External}}_{\mathrm{B\to A}}$. M_{hand} predicts that both RST packets are dropped, as the model does not check for RST flags. However, Click NF allows the first RST packet to reset the NF state. We also plugged the model for PfSense into a network verification tool, VMN [152]. The existing model in VMN does not check TCP flags. Using VMN, we verified the property: "TCP packets from an external host, B, can reach A even if no SYN packet is sent from A." Recall that in PfSense, $\mathrm{SYN}^{\mathrm{Internal}}_{\mathrm{A\to B}}$ needs to be sent for B to send TCP packets to A. Hence, the property is NOT SATISFIED. Using $Model_{hand}$, the tool returned that the property is SATISFIED whereas using $Model_{Alembic}$ indicated that it is not (i.e., *false-negative* for $Model_{hand}$).

3.8 Other Related Work on Network-Wide Verification and FSM Inference

Chapter 2 presented prior work on analyzing individual network functions and services (Section 2.1) and other domains such as protocols and software (Section 2.2.3). We now discuss the related work specific to Alembic. First, we discuss orthogonal efforts on testing and verifying network-wide properties. While Alembic focuses on inferring high-fidelity models of a single NF. a large body of work on network testing and verification (e.g., [97, 104, 126, 128, 152, 172, 185]) assumes that individual functions are correct but seek to test conformance and verify network-wide properties (e.g., reachability, absence of black holes). The high-fidelity models produced by Alembic can be consumed by these testing and verification tools. Then, we also discuss related work on automatically learning FSMs from the learning theory community. This line of work (e.g., [62]) laid the foundation for inferring FSMs for Alembic.

Network-wide testing and verification: Earlier work in this space (e.g., [104, 126, 128]) focused on simple switches and router. More recent work have focused on testing (e.g., [97, 172]) or verifying network (e.g., [152, 185]) composed of "stateful" network functions. BUZZ uses symbolic execution to test for policy compliance. Symnet [172] statically analyzes an abstract model of the data plane model consisting of NF models written using their domain language, SEFL. NetSMC [185] develops a network semantic model and a policy language to verify a wide range of policies (e.g., dynamic service chaining, path pinning). However, many of these tools use network function (NF) models to guide testing and verification. Unfortunately, the models of these NFs are hand-generated, and using a low-fidelity NF model can affect the effectiveness of verification tools [146] (While Symnet wrote parsers to automatically generate NF models using their language, SEFL, this parser does not generalize to other NFs not written in Click or to arbitrary configurations). In fact, this is motivation for building Alembic.

FSM inference: For Alembic, L* algorithm by Dana Angluin lays the foundation for learn-

ing the FSM [62]. Similar to Alembic, L* algorithm has also been leveraged by related efforts on discovering cross-site scripting attacks against web-application firewalls [64], TCP/IP implementations [102], and SSL/TLS implementations [171] as discussed in Chapter 2. Further, the techniques of learning FSMs has been used for model checking black-box systems (e.g., [113, 157]). Symbolic finite automata (SFA) [177] are FSMs where the alphabet is given by a Boolean algebra with an infinite domain. While Alembic does not directly formulate the problem to infer SFAs, we use the homogeneity assumption in the IP and port ranges to learn a *symbolic model*. Hence, using abstractions like SFA may help us to naturally embed symbolic encodings. We could potentially leverage a tool (e.g., [93]) that extends L* to infer the SFA. However, using SFA does not address the NF-specific challenges (e.g., inferring the key, handling modifications) but may serve as the basis for interesting future work.

3.9 Summary

In this chapter, we presented Alembic, a system to automatically synthesize NF models. Alembic leverages structural properties of NFs and their input and configuration space. Specifically, to tackle the challenges stemming from large configuration spaces, we synthesize NF models viewed as an ensemble of FSMs. Alembic consists of an offline stage that learns a symbolic model for each rule type (e.g., a firewall drop rule) and an online stage to compose concrete models given a concrete configuration. Our evaluation shows that Alembic is accurate, scalable, and enables more accurate network verification compared to prior methods. While Alembic demonstrates the promise of NF model synthesis, there remain some limitations (Section 3.2.3). We also discuss interesting avenues for future work in Chapter 6.

Chapter 4

Pryde: Automatic Synthesis of Evasion Attacks for Black-Box Stateful Firewalls

> "Well, let's just say I know a little girl
> who can walk through walls."
>
> *Charles Xavier [46]*

The Problem: Network firewalls (FW) play a critical role in securing our current network infrastructures in various deployment settings – including enterprise networks [51], cloud virtualized networks [42], and modern containerized settings [48]. Of particular interest to the security community are stateful firewalls. These stateful firewalls, as opposed to simple access control lists, track the state of individual TCP connections (together with firewall rules) to determine which packets are forwarded. To protect internal hosts from untrusted external hosts, a canonical policy in these settings is to only allow packets on TCP connections that have previously been established by hosts inside the intranet.

Despite the critically of stateful firewalls in securing our intranet, administrators deploying firewalls implicitly assume that the vendor implements the stateful semantics correctly. Unfortunately, if vendor implementations are erroneous, then it can weaken the security posture. That

is, an enterprise stateful firewall with such errors in implementing these stateful semantics could allow external attackers to reach internal hosts that should not be reachable. This is orthogonal to prior work in firewall analysis that examines the safety of the firewall rules and misconfigurations (e.g., [54, 56, 184]). The types of vulnerabilities we consider are a more fundamental semantic vulnerability in tracking internal states and hence are viable even if the rules are configured correctly. Unfortunately, operators have few, if any, tools to check if the vendor firewall implementations have such semantic vulnerabilities. Manual investigation will be ineffective and not scalable across many vendors and versions. Ideally, we need to automatically identify such evasion vulnerabilities. Complicating this further, firewalls are proprietary and acquired from vendors. Hence, operators have limited visibility into the code/internals of these firewalls.

This black-box setting makes the analysis even more challenging. While fuzzing or black-box pen-testing tools (e.g. [41, 50, 106, 111]) or recent work on censorship evasion (e.g., [69, 179]) appear as strawman solutions, in practice they have shortcomings. First, their focus is often orthogonal to our intent. For instance, pen-testing focuses on finding privilege escalation or application-layer problems in the management APIs. The prior work on censorship evasion focuses on the opposite problem of an internal host accessing an external service. Second, they cannot handle the large search space of possible stateful or connection-oriented packet sequences. Third, they are not robust in discovering subtle attacks across firewall implementations with varying degrees of implementation complexity.

The Solution: In this chapter, we present Pryde[1], a black-box analysis framework for automatically uncovering semantic evasion vulnerabilities for firewalls. Pryde works in a black-box setting; Pryde only requires input-output access to the firewall and does need visibility into the binary or the code. We specifically focus on evasion vulnerabilities that allow an external attacker to circumvent the firewall to send an undesirable "data" packet to an internal target which

[1]The name Pryde is inspired by the Marvel X-men superhero Kitty Pryde who has the ability to use her "phasing" powers to walk through walls [46].

should not be reachable by the policy. This capability can subsequently be used as a starting point for more detailed attack campaigns for persistent threats, lateral movement, or exfiltration of sensitive information.

As we also saw from our motivating scenario in Section 1.1, these evasion attacks exploit subtle implementation errors (similar to those in Figure 1.4) in implementing the stateful semantics. Hence, instead of directly searching for these complex attack sequences, Pryde uses a *model-guided approach* [174] that proceeds in two logical phases. The first phase uses black-box model inference to reason about the stateful behavior a given firewall implements for different packet sequences. Then, given this inferred model, we consider different deployment and threat scenarios to identify evasion vulnerabilities. In contrast to structure-free approaches (i.e., random fuzzing or randomly generating test packet sequences), our approach is *efficient* and can uncover many semantically different vulnerabilities (Section 4.5). Our approach is general across firewall implementations and extensible to support future evasion scenarios. In designing and implementing Pryde, we address key technical challenges to (1) make the model inference robust to consider different types of packets, and (2) efficiently encode the scenarios in a model checker and define custom refinement constraints to enable the model checker to explore new attack pathways efficiently.

Findings: We designed and implemented Pryde realizing the aforementioned workflow and key ideas. We evaluate Pryde on four popular (virtual) firewalls where three are commercial-grade and one open-source. Two of them are taken from the Amazon EC2 marketplace [40] and also used in the cloud deployments (As we are disclosing our findings to the affected vendors, we anonymize vendor names in this thesis). We summarize our findings below (Section 4.5):

- *Many semantically distinct attacks*: We uncover thousands of semantically distinct (i.e., with respect to the connection states traversed) attacks—2,591 for FW-1, 2,355 for FW-2, 8,220 for FW-3, and 294 for FW-4. Post-processing these attacks, we find that these attacks exploit different aspects of a TCP protocol. For instance, some of these attacks exploit a firewall for-

warding an external DATA packet "even before" the three-way handshake has been completed or after an incomplete handshake has been disrupted. Some exploit the simultaneous open feature of the TCP and/or SYN retries. At a high level, these attacks exploit (1) non-traditional sequences of TCP packets; (2) interference from other TCP connections (e.g., same headers in reverse direction or non-compliant sequence/ acknowledgement numbers), and combination of (1) and (2).

- *Many evasion attacks are subtle*: While some attack sequences are simple (i.e., requiring only one or two TCP packets), many others are quite subtle and require a nuanced TCP packet sequence. Specifically, we uncover more attacks involving longer sequences that are not captured in smaller sequences; e.g., 47 attacks for an attack length of 1 to 3 vs. 9,231 for an attack sequence length of just 7. For instance, FW-1 entails a carefully constructed TCP packets involving RST packets after the SYN retries to allow a DATA. Several attacks are also unique to specific vendors and do not extrapolate across vendors.
- *Strawman solutions are ineffective*: In contrast to Pryde, random fuzzing only finds a handful of attacks (0 to 3) for ProprietaryNF and FW-1 after more than 15K tries. This approach is not robust across firewall implementations. Recent work on automatic censorship evasion using genetic algorithms (e.g., [69]) is also ineffective at uncovering these attacks. (To be fair, censorship circumvention focuses on an orthogonal problem with a different system goal and threat model that makes their strategies ineffective in our context.)

Having summarized our findings, we now put them in a historical context. We note that evasion attacks have been demonstrated in other settings. For instance, earlier works have uncovered evasion attacks against network intrusion detection systems (NIDS) (e.g., [153, 160]) or censorship firewalls (e.g., [69, 136, 179]). In these attacks, an attacker similarly sends a sequence of malformed (incorrect) packets to evade the intended policy (i.e., a censorship firewall blocks certain content). However, Pryde is the first system that demonstrates that such evasion attacks that exploit the semantic vulnerabilities are feasible against *enterprise stateful firewalls*

(with different configurations and settings than those of NIDS and censorship firewalls).

Apart from uncovering evasion attacks against enterprise firewalls, Pryde differs from these prior works in other settings in that it takes a more *principled* method using a model-guided workflow. As a result, Pryde can discover subtle and complex attacks that exploit the fundamental errors in implementing stateful semantics of tracking per-connection states. Our approach contrasts with manual analysis (e.g., [153, 160, 179]) or random-based testings (e.g., [69]). Our findings suggest that these attacks are subtle and complex such that random-based testings are highly ineffective for complex firewall implementations. Lastly, our automated, general framework enables Pryde to reason about how an unconventional (but realistic) threat model (Section 4.1.2) of an attacker colluding with a weak insider can enable strong attacks.

Ethics and Disclosure: We have disclosed preliminary findings and are in the process of providing detailed reports to vendors and cloud providers involved.

4.1 Background and Motivation

In this section, we provide the background on stateful firewalls and the deployment model we consider. Then, we highlight motivating examples of evasion vulnerabilities against a commercial firewall to motivate our work.

4.1.1 Background on Stateful Firewalls

A firewall is the central network device that stops or mitigates unwanted access to the private (protected) networks from other untrusted external networks such as the Internet. We specifically consider a layer-3 stateful firewall (FW) in this work. A stateful firewall decides whether to drop or forward a network packet based on the 5-tuple defining a connection (srcip, srcport, dstip, dstport, proto), and the configured rules. Typically, a stateful firewall is configured with rules of the following form: $\{$ interface, srcip, srcport, dstip, dstport $\rightarrow$ action$\}$ where interface

refers to an interface where incoming TCP packets are matched to (also, can be a wildcard). A firewall configured with each rule keeps the states of network connections (i.e., a TCP connection state) and tracks a state during a connection's lifetime based on the 5-tuple. Further, a firewall also monitors both incoming and outgoing packets across interfaces to update the connection state accordingly. This connection state may include details such as the sequence (seq) and acknowledgment (ack) numbers of the packets traversing the connection.

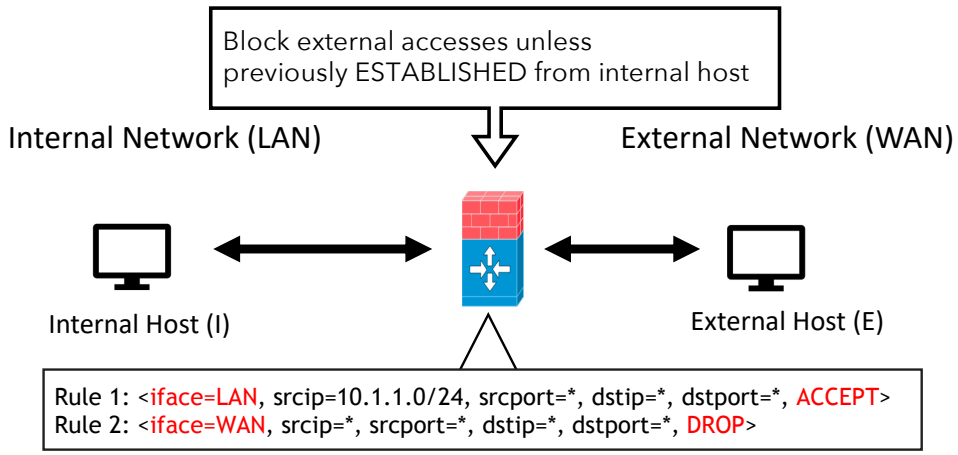
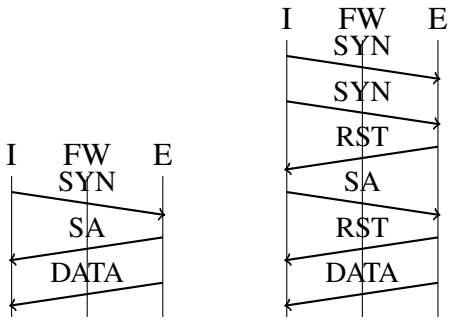


Figure 4.1: A Stateful Firewall in an Enterprise Network

Deployment model: In an enterprise network, stateful firewalls (Figure 4.1) are typically configured to drop all external packets that do not belong to an established connection initiated from an internal host. For instance, the firewall must allow internal hosts to initiate a connection to `google.com` and also allow the return "data" packets from `google.com` to arrive at the host that initiated the connection. This is the "stateful" part; if we only had simple stateless rules to allow traffic from internal hosts to `google.com` and block incoming traffic from the Internet, then the return data packets from `google.com` would never arrive.

More generally, suppose a TCP packet with srcip:A, srcport:Ap, dstip:B, dstport:Bp enters via an external interface, the firewall checks whether the connection from srcip:B, srcport:Bp, dstip:A, dstport:Ap is in the established state. If it is, the packet will be forward and otherwise, this packet will be dropped.

The enterprise scenario we consider differs from other scenarios such as censorship. In the censorship scenario, FWs are typically configured with the "default-allow" policy for packets originating from both directions. However, these firewalls will inject RST packets (to terminate the connection) if a client accesses "blocked" content. As such, the evasion attacks we focus on are orthogonal to those considered in censorship circumvention (e.g., [69, 179]).



(a) Scenario 1 **(b) Scenario 2**

Figure 4.2: Packet sequences played against a firewall (anonymized vendor, FW-1); Scenario 1 is an identical sequence from the motivating scenario (Figure 1.4b in Section 1.1), but we present here again for ease of reference

4.1.2 Motivating Scenarios

Note that the logic of how the firewall decides to a particular action to the current state plays a critical role in determining whether a packet is dropped or not. Therefore, a flawed implementation on how a firewall tracks a connection state can have a detrimental effect on the security posture. To motivate this, we use concrete evasion attacks we uncovered with a real commercial FW-1.

Before discussing the concrete attack scenarios, we first showcase *vulnerable* sequences of packets (which can be exploited for attacks). Specifically, this leads to a firewall allowing a DATA packet from an untrusted external host (E) to an internal host (I).

To explain these suspicious sequences, recall a normal (standard) packet sequence we have

shown in Figure 1.4a from Section 1.1. This is a sequence a firewall expects to see. Let us elaborate on this standard packet sequence. An internal host wants to access `google.com` and initiates a connection setup by sending a TCP SYN. `Google.com` acknowledges this by sending a SYN-ACK (SA for short), followed by an ACK from an internal host, thereby completing a proper three-way handshake. At that point, an internal host and `google.com` can freely exchange DATA packets. Now, we show two strange sequences of TCP packets, that will start allowing a DATA packet from an untrusted external host; these strange packet sequences drive a connection state to some limbo (vulnerable) state and that is when the firewall also starts to accept a DATA packet from untrusted external hosts.

Scenario 1 (Incomplete handshake): As seen in Figure 4.2a, the FW-1 allows a DATA packet from `google.com` "even before" a three-way TCP handshake has been completed; i.e., the firewall is not checking whether the last ACK packet from an internal host has been sent or not. Such a simple error highlights that implementing even a very basic stateful semantics of checking for a complete handshake can be erroneous. In practice, this problem is much worse than what it appears as such an error can manifest in so many different ways (i.e., polymorphic variants of this attack as we will see in Section 4.5.2).

Scenario 2 (SYN retries + Teardown): We now show a more complex and a different sequence (Figure 4.2b) from the Scenario 1. This scenario exploits an implementation error in how a FW-1 FW handles a combination of SYN retries and connection teardown (While prior works on censorship evasion (e.g., [69, 179]) find similar attacks, their focus is orthogonal as their deployment and the system model differs. Their findings do not directly apply to our setting (Section 4.5.1).) Here, an internal host first sends two subsequent SYN packets (like SYN retries). Then, an external host sends a RST packet, which drives this connection state to some "limbo" state. After two non-traditional TCP packet exchanges (an internal SA followed by an RST), a firewall allows a DATA packet from an external host! One may wonder whether all of these 5 packets are necessary for a firewall to allow a DATA packet. In fact, that is the case as each packet in a

sequence modifies the connection state. Further, only after the first two SYN packets, the firewall does not allow a DATA packet for a FW-1 (however, a FW-3 stateful firewall does, which is motivating our work). These sequences are discovered by our tool and as we discuss in Section 4.4.1, we specifically only output semantically-distinct attacks (from a black-box perspective).

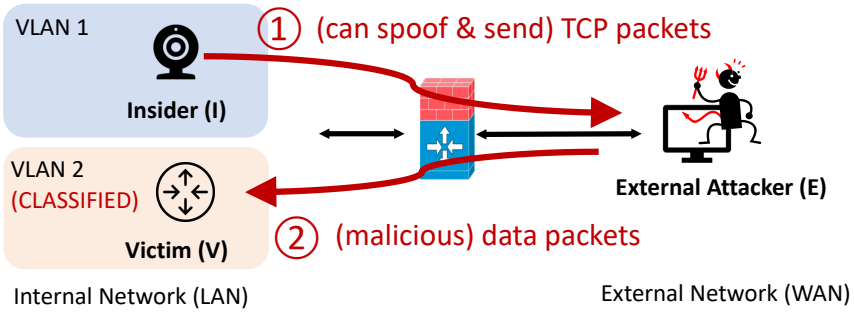


Figure 4.3: Attack scenario setup

Attack scenarios: Now, let us think about how such erroneous implementation can lead to concrete evasion attacks. One precondition for concrete attacks for the above two scenarios is that an internal host needs to send specific TCP packets (e.g., the first SYN packet in Scenario 1) in coordination with an external attacker. But, this isn't too difficult as any internal host can easily *spoof* the source IP-port and send it outwardly. Consider a case where we have a compromised insider such as an IoT device (e.g., a smart printer) in the intranet [45, 52]. In fact, first compromising these IoT devices is increasingly gaining traction for hackers due to IoT devices' prevalence in today's enterprise network [49] and their lack of built-in-security [45]. In fact, it is recently reported that Russian-state hackers, Strontium (APT28), have been caught attempting to hack IoT devices (e.g., an office printer, a video decoder) to gain entry points into their targets' internal networks [47]. Such an insider (Figure 4.3) may lack direct access to the target victims as it is not located in the same VLAN as the target victims (e.g., router, end host). However, this insider colludes with an external attacker and exchange a sequence of pre-defined TCP packets (including Step ①). Finally, a firewall allows a DATA packet directed at the target victim (Step ②). The best micro-segmented network with VLANS using the best

practices today [43] is also vulnerable to these attacks.

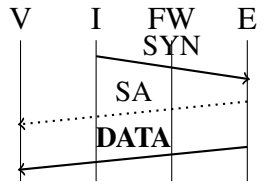


Figure 4.4: Scenario 1 mapped to an attack setup

Figure 4.4 shows the Scenario 1 mapped to a concrete attack. In this attack, an insider first sends a SYN packet with a source IP-port that of a victim, followed by a SA from an attacker. Then, an external attacker can circumvent the firewall policy by sending a DATA packet (containing malicious payload) at a victim. It is outside the scope to precisely guarantee whether all victim software stacks actually accept and process this data packet as such. As such, we observe that there are many cases (e.g., routers, IoT devices) that will accept the data packet. At this point, the attacker gained an entry into a highly-classified VLAN. An attacker could either compromise this target victim or use this victim as another stepping stone to enable more sophisticated multi-stage attacks.

Attack characteristics: Having described motivating examples, we now derive several characteristics of these attacks:

- *Semantic evasion attacks are subtle*: These attacks exploit subtle implementation nuances in a firewall's logic of tracking a per-connection state. Specifically, to make these attacks to work, one needs to carefully construct packet headers with the above TCP flags and seq numbers. Moreover, these attacks may be specific to each firewall vendor's implementation.
- *Semantic diversity of evasion opportunities*: As we saw brief examples for FW-1, as these attacks the fundamental issue in tracking per-connection states, there tend to be multiple such attacks exploiting diverse mechanisms (e.g., handling teardown packets, incomplete handshake, SYN retries). Further, even within attacks that exploit a similar mechanism, there are multiple polymorphic variants that are semantically different [146] (detail in Section 4.5) that

explore the different stateful semantics.

The above suggests that we need a general and robust framework to uncover such evasion attacks. Given the subtle and implementation-specific nature of these attack opportunities and stateful behaviors involved, strawman solutions such as randomly generating packet sequences are inefficient (Section 4.5).

4.2 Pryde Problem Overview

In this section, we formulate the problem of enabling a model-guided approach and formulate our problem. We provide an overview of the Pryde workflow and discuss the key challenges in realizing our workflow.

4.2.1 Threat Model

We begin by scoping the adversary's goals and capabilities.

Adversary goals and capabilities: The attacker's goal is to circumvent the firewall and send a DATA packet to an "unreachable" internal victim. We assume the following attacker capabilities and constraints.

- *Send constructed packets*: An external attacker can craft TCP packets and send them to internal hosts.
- *A colluding insider with minimal privileges*: The attack may optionally have a "weak" insider that can spoof the source of the TCP packets and send them to external hosts. A "weak" insider cannot directly send packets to the victim; e.g., internal firewalls or VLAN policies may prevent this.
- *Firewall-specific knowledge*: The attacker does not know the rules that the firewall is configured with. We assume the attacker knows the firewall vendor/version; if not this can be obtained by known fingerprinting mechanisms such as banner grabbing [44]. We assume the

attacker has no visibility into the internal implementation or code. However, we do assume that the attacker can have offline "black-box" access to the firewall (e.g., obtain a virtual firewall appliance [40]).

4.2.2 Problem Formulation

Input and output: Pryde works in a black-box setting where it does not require access to the source code or internal logic. Pryde takes as inputs: (1) firewall binary or virtual appliance; and (2) a deployment and threat model defining different entities (i.e., an insider, an external attacker) and their capabilities (e.g., an insider can spoof and send TCP packets). The output of Pryde is a set of semantically-distinct concrete evasion strategies (we present the definition in Section 4.4.2). Specifically, each concrete evasion strategy is an ordered sequence of *input TCP packets* mapped to the corresponding sender (i.e., an insider or an external attacker); an example was shown in Figure 4.4.

Scope: In this work, we focus on sending a DATA packet from an external attack to an "unreachable" internal victim (as defined by the policy). We acknowledge that not all victims may actually accept and process this data packet. However, we observe that there are many cases (e.g., routers, IoT devices) that will accept the data packet. Note that the attack's goal after this circumvention (e.g., installing back-doors or lateral movement) is outside our scope. Our attack is a fundamental first step that can enable future attacks.

Challenges: There are two main challenges in enabling our vision. First, the input space is too *large* for an unstructured search (i.e., random search). Specifically, as we deal with an adversarial scenario, we need to consider *sequences of packets* where each packet can come from diverse sets (e.g., a sequence of non-standard TCP flags, out-of-window packets, a flow with a flipped direction). Second, there may be *multiple such evasion strategies* that could exploit different features or code paths. While we cannot guarantee coverage, our goal is to discover as many attacks as possible and also attacks that are semantically different (from the point of view of the

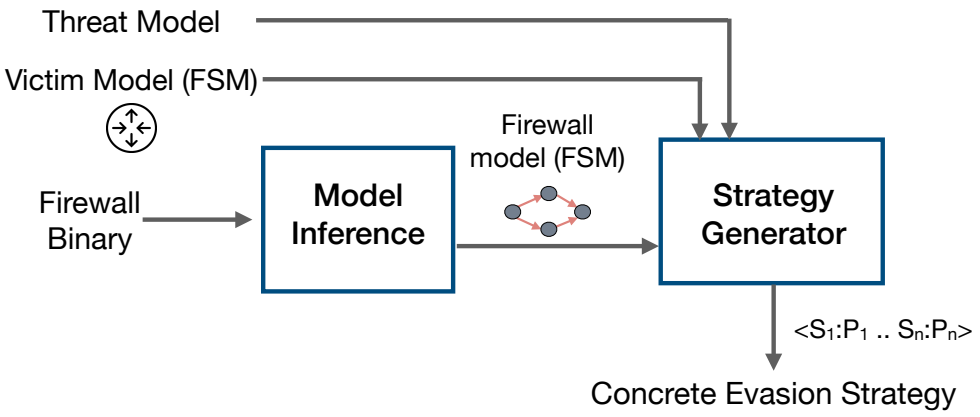


Figure 4.5: Pryde System Overview

black-box analysis).

4.2.3 High-Level Design

A case for a model-guided approach: The evasion attacks we consider exploit nuanced firewall-specific aspects. For instance, Scenario 2 (Section 4.1) incorrectly allows an external DATA packet, after SYN retries and teardown packets. That is, identifying such attacks require carefully-constructed sequences of TCP packets, triggering internal state transitions that will not be exercised by normal TCP sequences. As a result, strawman solutions (e.g., random fuzzing) will not discover many of these attacks for many firewalls (Section 4.5.1). Instead, we adopt a *model-guided approach*, where we first infer a behavior model of the stateful semantics. We do note that this model has to be specific to each firewall implementation, since the connection handling semantics of different firewalls may be different. Having a model enables us to systematically search over this state-space to discover semantically distinct attack opportunities.

A case for a two-phase approach: One option of a model-guided approach is to couple threat model encoding with model inference. Unfortunately, this is not *extensible* as our system and deployment assumptions change; e.g., modeling a weak insider option would require us to re-learn the model for each possible scenario of the insider's capabilities. By decoupling model inference from attack generation, our workflow is *extensible* to future threat and deployment

models. Thus, Pryde consists of two logical modules (Figure 4.5):

1. *Model Inference (Section 4.3)*: Given a black-box firewall implementation/binary, the Model Inference engine outputs a Finite-State Machine (FSM) model. This model describes the input and output packets of the firewall in a given connection state. Specifically, Alembic from Chapter 3 shows the feasibility of blackbox model inference for stateful firewalls [146]. However, in designing Alembic, we made several simplifying assumptions and focus on inferring a firewall model under typical or normal packet sequences. Thus, we cannot directly use Alembic for the context of Pryde; e.g., how a firewall behaves in presence of out-of-window packets that "interfere" with an existing connection. We address key challenges in extending Alembic to infer model under more general or anomalous packet sequences.

2. *Strategy Generator (Section 4.4)*: Given a firewall model, we need a systematic way to uncover of evasion attacks under the interactions of different entities (i.e., attacker, victim, insider, and the firewall). To this end, in this module, we formulate these system interactions using SMT and use Z3 [90] to build a custom model checker. We model the problem similar to bounded model checking [82], where we check if a bounded length path sequence exists. To uncover semantically different attacks, after finding one attack, we refine our constraints in the model checker to uncover more semantically-different attacks.

4.3 Model Inference

In this section, we discuss the design of the ModelInference module. Specifically, we recognize that while Alembic (in Chapter 3) is a good starting point, we first elaborate on why it is fundamentally insufficient for Pryde. Then, we discuss our extension to Alembic to build the ModelInference module.

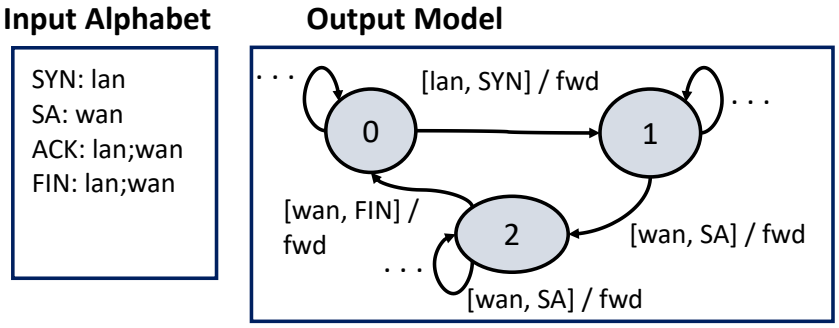


Figure 4.6: An example of an output model and the corresponding input alphabet used by Alembic (lan for internal and wan for external due to space)

4.3.1 Limitations of Alembic

Let us first recall the design choices we made for Alembic (Chapter 3). Then, we can pinpoint about why it is insufficient for Pryde. At a high level, Alembic was designed for the verification and testing workflow, and, hence, assume mostly TCP-compliant (non-adversarial) workloads.

Given this assumption on deployment, Alembic takes an input alphabet (Σ) describing a "scoped" (TCP-compliant) set of packet types of interest (e.g., TCP SYN from an internal host, I, to an external host, E, TCP SA from E to I) as shown by Figure 4.6. Further, we propose using specific optimizations to reduce the size of an Σ. For instance, rather than considering the entire possible space of TCP headers such as sequence (seq) and acknowledge (ack) numbers (32-bits each), their system re-writes these seq and ack number of TCP packets during the actual inference to adhere to the TCP semantics. By doing so, Alembic does not have to search the space of seq and ack numbers. Given this background, we discuss two key dimensions that render Alembic insufficient.

1. *Need to consider diverse input alphabets*: We designed Alembic (Chapter 3) to model the firewall for mostly TCP-compliant workloads. However, we need to consider adversarial scenarios (e.g., out-of-window packets) that "interfere" with the TCP-compliant connection states. For instance, Scenario 2 (Section 4.1) would not have been discovered if we hadn't considered an internal SA. While this is just a simple example, we need a systematic way to

generate input alphabets to reason about potential evasions.

2. *Support for rewriting logic*: As this tool, Alembic, makes optimizations to reduce the search space by rewriting seq and ack numbers during the inference. Unfortunately, we need to consider adversarial cases where such re-writing logic may not help us to uncover certain types of evasion attacks. We need to come up with a *general* way of handling seq and ack headers during the inference to support various types of input alphabets.

4.3.2 Generating Evasion-Centric Input Alphabets

We discuss how we generate input alphabets (Σ) for the evasion attacks. A strawman solution is to just generate all possible packets as Σ. Unfortunately, the size of input will grow exponentially as we need to consider possible combinations of directions, TCP flags, and those TCP packets that adhere to the TCP semantics and those that do not. Instead, our idea is to come up with an ensemble of *independent* Σ where each model learned with a given Σ sheds light on the potential evasion scenario (i.e., TCP states interfering with packets with the reverse direction). That way, our method is systematic and extensible to future Σ we may consider.

Further, evasion can happen with or without interfering packets. Hence, we first set a *basic* Σ that can reason about attacks using just "non-traditional" sequences of packets (i.e., a sequence of non-standard TCP flags). Then, we showcase how we generate Σ to reason about interference. In this work, we only consider interference from packets that share the *same* bi-directional tuple; i.e., a TCP packet from lan has source A and destination B and a packet from wan has source B and destination A. This is a conscious decision as from observations/anecdotes suggest the firewalls mostly have flaws in processing the state for packets with the same 5-tuples. (It is easy to extend our design to check if two connections with different source and/or destination interfere with each other.)

Basic input alphabet: Figure 4.7 shows the two basic input alphabets (Σ). We also denote abbreviations for input symbols or packet types (e.g., SA and $\mathrm{SYN\text{-}ACK}$, DA for DATA).

Data Injection (DI)	Data Injection with Teardown (DI-T)	
SYN: lan	**SYN:** lan	**R:** lan; wan
SA: wan	**SA:** wan	**RA:** lan; wan
A: lan; wan	**A:** lan; wan	**F:** lan; wan
DA: wan	**DA:** wan	**FA:** lan; wan

Abbreviation:
SA : SYN-ACK
R : RST ; RA: RST-ACK
F : FIN ; FA: FIN-ACK
DA : DATA

Figure 4.7: Basic Input Alphabet for Alembic

- *Data Injection (DI)* : This Σ helps to reason about potential circumvention just using the connection setup packets. This Σ has SYN from lan, $\mathrm{SYN\text{-}ACK}$ from wan, ACK from lan and wan, and DATA from wan (left column in Figure 4.7).

Interference input alphabet: We now need to consider Σ to also reason about potential evasion from interference with non-compliant TCP packets; e.g., packets that do not belong to the same connection (e.g., [69, 179]). Specifically, we observe from anecdotes/prior works that the firewalls can incorrectly map the state with (1) TCP connection with flipped direction (e.g., SYN from lan vs wan); and (2) packets with out-of-window seq numbers. These cases are "subtle" such that these packets share the same 5-tuple. However, if a firewall was implemented correctly, it should have not been considered so.

In this work, we consider four possible types of interference (IX) input alphabet. Given each IX set, we append them to the basic Σ to reason about firewall's behavior when it encounters both types of packets. While we come up with a representative Σ, our design is easily extensible for a new Σ. Further, we consider interference from connection setup-relevant packets.

1. *Reverse directions (*$\mathrm{IX}^{\mathrm{dir}}$*)* : These TCP 3-way handshake packets have flipped direction only; i.e., SYN coming from the wan network, $\mathrm{SYN\text{-}ACK}$ from lan, and ACK from wan. The seq and ack numbers are *not* out-of-window.
2. *Reverse direction and random seq/ack numbers (*$\mathrm{IX}^{\mathrm{dir}}_{\mathrm{rand}}$): These TCP 3-way handshake packets have flipped direction and also are out-of-window. Specifically, the seq & ack numbers are randomly initialized.
3. *Reverse direction and random seq/ack numbers (*$\mathrm{IX}^{\mathrm{dir}}_{\mathrm{conn}}$): This case is similar to the previ-

ous case. However, these out-of-sequence packets themselves form a connection; i.e., their seq/ack are are in-window among themselves.

4. *Packets with out-of-sequence* ($\mathrm{IX}_{\mathrm{rand}}$): These TCP 3-way handshake packets have the same direction but are out-of-window and randomly initialized.

A generative model for input alphabets: The generative model for the input alphabet is formed by considering each of the four basic Σ independently. For each basic input alphabet (Σ), we append the IX set and reason about the firewall's behavior. This gives a total of 2 basic Σ and $\times$ 5 (1 for basic and 4 for interference) $= 10$ input alphabet for each firewall. However, note that during the model interference, the model may not converge for certain firewalls; i.e., Alembic assumes that the underlying model is deterministic and fail if this is not the case. Hence, if the model does not converge for the basic Σ, we do not model for the corresponding *input alphabet* appended with IX set.

4.3.3 Extending the Inference Algorithm

We now discuss how we enable the model inference under these diverse input alphabets.

Baseline rewriting logic ($\mathrm{RWLogic}()$**):** We briefly discuss the seq/ack rewriting logic of Alembic to reduce $|\Sigma|$ At a high-level, their tool tracks seq and ack of the transmitted packets and rewrites them during the inference to adhere to the correct semantics. For instance, suppose the underlying L* plays this packet sequence of length 2: 1) SYN from A to B from the lan network (SYN:lan), and 2) SYN-ACK from wan. If the firewall forwards the first SYN to a wan interface, then the seq number of the transmitted SYN is used to update the seq and ack number of the TCP packets with a source B and a destination A. We refer to this entire logic as $\mathrm{RWLogic}(seq = X)$ where X is the initial seq number.

Rewriting logic to support interference set: We now discuss how we adjust this logic to handle the "interference" packets. The beauty of this extension is in the generality of this design that makes it extensible and general for future Σ to consider. Note that from the IX sets from

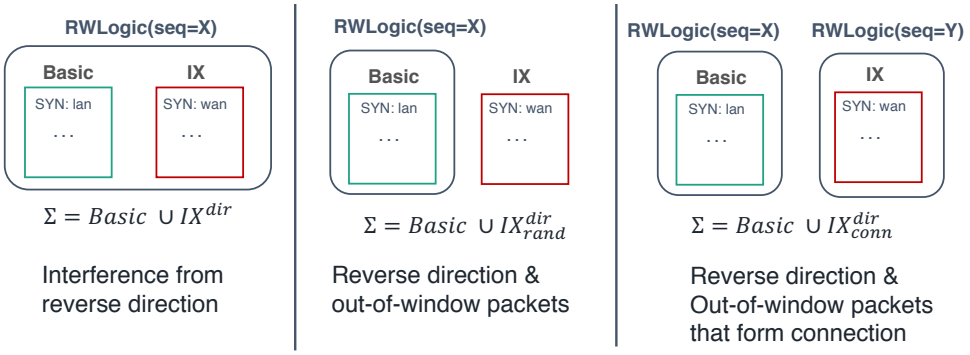


Figure 4.8: Rewriting logic to handle interference (IX) set

Section 4.3.2, there are cases where we need to subject only a certain group of packets to rewriting, or subject multiple groups of packets to independent rewriting (i.e., reasoning when two connections with the same 5-tuple but initialized with different seq number such as $\mathrm{IX}^{\mathrm{dir}}_{\mathrm{conn}}$).

For a systematic design, we group packets accordingly and subject each group to a corresponding rewriting logic (as shown in Figure 4.8). For instance, if we consider an interference set where only a direction is reversed (i.e., $\mathrm{IX}^{\mathrm{dir}}$), then we would subject both the basic and the IX set to have the seq/ack numbers in-sync (the left side of Figure 4.8 shows both sets being subject to $\mathrm{RWLogic}(X)$). Now, consider an interference set where these packets have a reverse direction and have out-of-window seq/ack numbers are (i.e., $\mathrm{IX}^{\mathrm{dir}}_{\mathrm{rand}}$). Then, packets in the basic Σ are subject to $\mathrm{RWLogic}(X)$ and packets in the IX set are only randomly initialized and not re-written during the inference. Similarly, for IX where packets themselves form a connection, then we will independently rewrite seq and ack for this set (i.e., $\mathrm{RWLogic}(Y)$). The internal of modified implementation of Pryde keeps track of which "set" each packet belongs and decides if/how the rewriting logic is applied. These are all exposed configurable parameters.

4.4 Attack Strategy Generator

In this section, we discuss how we used the inferred models from Section 4.3 to generate concrete evasion strategies. We discuss how we encode the entire system model and the interaction between different entities (Section 4.4.1). Then, we demonstrate how we achieve coverage across

distinct attacks (Section 4.4.2).

4.4.1 Encoding the System Model

Strategy Generator takes an input of a firewall (FW) model (from Section 4.3) and the system model. The system model is defined by (1) a model of a victim, and (2) the threat model that defines each entity (i.e., insider, attacker) and their capabilities w.r.t. the packets that they can send. Specifically, we encode into the model checker that an insider can *spoof* the source IP and address of another internal host (e.g., victim).

The output is a *concrete evasion strategy*, which is an ordered sequences of *located input packets* (a concept borrowed from prior work in network verification [126]) mapped to the corresponding sender (i.e,. an insider or an external attacker).

At a high-level, a *located input packet* that comes to either of an interface of a firewall and we define it below:

Definition 8 (A located input packet)**.** *A located input packet, σ, is defined by the following tuple $(intf, srcip, srcport, dstip, dstport, tcp, data, pre, seq, ack)$: (1) $intf$, an incoming interface (i.e., internal or external), (2) $srcip$, a source IP, (3) $srcport$, a source port, (4) $dstip$, a destination IP, (5) $dstport$, a destination port, (6) tcp, TCP flags, (7) $data$, a Boolean indicating the presence of a* DATA *(payload), (8) pre, a prefix indicating whether the seq/ack numbers were rewritten during the inference to follow the TCP semantics, (9) seq, a sequence number , (10) ack, an acknowledgement number. For (9) and (10), we use a variable such as X, X+1 to denote a relation across packets.*

A model of a firewall (from Section 4.3) is a Mealy machine and is defined by the following tuple $(\mathrm{Q}, \Delta, \mathrm{O}, \Phi)$: (1) Q, a set of states, (2) Δ, a set of input packets, (3) O, a set of output packets, and (4) Φ, a set of transitions. Similarly, a model of a victim is defined in the same manner as a firewall.

Encoding a firewall as a function of time: As mentioned, a concrete evasion strategy is an *ordered sequence* at discrete timesteps; i.e., we need to model the progression of time as a function of timesteps. For instance, when a firewall gets an event (i.e., get a located input packet), the timestep T advances to $T + 1$. Such an event changes the "state" of the entire system. Therefore, we use the following functions to describe a state of a firewall at a given timestep, T:

1. $\mathrm{State} : \mathrm{Q} \times \mathrm{T} \rightarrow \mathrm{Bool}$. Boolean function that indicates if a given state of the firewall, $s \in \mathrm{Q}$, occurs at a timestep T;
2. $\mathrm{Input} : \Delta \times \mathrm{T} \rightarrow \mathrm{Bool}$. Boolean function that indicates if an input packet, $\sigma \in \Delta$, occurs at a timestep T;
3. $\mathrm{Output} : \mathrm{O} \times \mathrm{T} \rightarrow \mathrm{Bool}$. Boolean function that indicates if an output packet, $o \in \mathrm{O}$, occurs at a timestep T;
4. $\mathrm{Trs} : \Phi \times \mathrm{T} \rightarrow \mathrm{Bool}$. Boolean function that indicates whether a specific transition, $\phi \in \Phi$, occurs at a timestep T. A particular transition is determined by an input packet (σ), output packet (o), and a current state (s).

We determine the possible sending entities based on the pre-specified thread model. As dictated by the threat model, an insider can spoof the source IP as a victim. Our encoding is extensible and more attributes can be easily be added.

Encoding constraints: We briefly describe how we encode these entities. Specifically, we encode a firewall function using propositional logic with the following constraints:

- At a given $timestep$ T, we encode the following restrictions: (1) exactly one state occurs, (2) at most one input packet occurs, (3) at most one output packet occurs, and (4) exactly one transition happens.
- To encode a firewall input 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 occurrences of a specific state and an arrival of a located input packet at the same timestep T. After a transition ϕ_j happens, then the state of

a firewall changes and as a result, we observe a corresponding output packet (defined by a Mealy machine). This can be represented as:

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

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

We also encode the victim's model using a similar logic. If an attacker's packet reaches the victim, then the next input is dictated by the victim's model (where in our current victim, a victim accepts all TCP packets). We discuss how this model could be extended in Section 6.3.1.

Encoding the goal: The goal is to find an ordered sequence of σ that leads to the firewall to be "evadable" at a given timestep T; i.e., a DATA packet from an external attacker reaches an internal victim.

4.4.2 Discovering Semantically-Different Attacks

To discover concrete attack strategies (i.e., sequences of a located input packet mapped to a corresponding sender), we leverage the idea from a bounded model checking (BMC) [82] where we find counterexamples with a bounded length. BMC is a common technique to find bugs in software that can be identified within a few iterations (i.e., timesteps in our case). By default, the solver terminates upon finding one counterexample. To find a new counterexample, we must add additional constraints to block this counterexample and make a new call to the solver. However, this procedure would find many attacks that may be semantically-identical. Since the model-checker can be time-consuming, instead of outputting thousands of semantically-identical attacks and then do post-processing, we made a design decision to encode additional constraints that block semantically-identical attacks during the search. This procedure is repeated until no more counterexamples are discovered.

We now discuss the invariant of the attacks we output and the refinement strategy we use to enable the discovery of semantically-distinct attacks.

Loop-free invariant: To efficiently search over the state space of a firewall's model, we want to output an attack string that uses a minimum of packets in traversing the state-space of a firewall. Hence, we encode a loop-free invariant into our model.

Definition 9 (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 input packet.*

Refinement strategy: Given this loop-free invariant, when we discover an attack packet string, a, composed of a sequence of located input packets, $\{p_1 \cdots p_n\}$, we exclude the exact packing string match. Hence, our refinement strategy corresponds to exclude equivalent strings.

Semantically-distinct attacks: Having defined the loop-free invariant and the refinement strategy, we can define semantically-distinct attacks. Note that, we can only provide this definition given the same input template (where the input space is identical).

Definition 10 (Semantically-distinct attack)**.** *Given two loop-free attack strings, a and a', they are semantically distinct if $a \neq a'$.*

By construction, the attack sequences (strings) generated by our tool are loop-free (from Def. 9). Additionally, as we do an exact string matching as a refinement strategy, all attacks that we output are *semantically distinct*.

4.5 Evaluation

System implementation: We implemented Pryde in Java atop Learnlib [162], an implementation of L* [62] for the Model Inference. We also built a custom `Python` based model checker using the Z3 SMT solver [90]. We implement other supporting modules for packet generation

using *Scapy* [27]. Additionally, we have an automated framework to spin up the ModelInference in Amazon EC2 or Cloudlab [94].

Setup: We use 4 off-the-shelf firewall implementations (i.e., FW-1, FW-2, FW-3, FW-4); three of them are proprietary firewalls and one has an open-source implementation (but we emulate it in a black-box manner). Two proprietary firewalls (i.e., FW-1, FW-2) were from the Amazon EC2 marketplace [40] and were set up in Amazon EC2. We ran FW-4 and FW-3 in VMs in VirtualBox [35] in CloudLab [94]. We ran the Strategy Generator to find attacks of length 1 to 7. To test concrete attacks, we set up a sandbox network with an insider, a victim, an external attacker, and the stateful firewall (configured to only allow TCP traffic from external hosts on already established connections as discussed in Section 4.1). We inject the packets via attacker and insider as dictated by the concrete attack strategy. For each firewall, we consider the following candidate input alphabets to identify evasion opportunities:

1. $\mathrm{Baseline}$: Basic Σ without involving interference packets
2. $\mathrm{IX}^{\mathrm{dir}}$: Interference by reverse direction only;
3. $\mathrm{IX}^{\mathrm{dir}}_{\mathrm{rand}}$: Interference by reverse direction and out-of-window packets with random seq and ack numbers;
4. $\mathrm{IX}^{\mathrm{dir}}_{\mathrm{conn}}$: Interference by reverse direction and out-of-window packets that adhere to the connection semantics w.r.t. seq and ack numbers;
5. $\mathrm{IX}_{\mathrm{rand}}$: Interference by out-of-window packets with random seq and ack numbers.

For each template, we run the model inference and the attack generation for 1) Σ involving connection setup packets (i.e., Data Injection from Section 4.3.2), and 2) Σ for Data Injection with Teardown.

4.5.1 Aggregate Summary of Attacks

We first start with an aggregate summary and discuss the effectiveness of the strawman solutions.

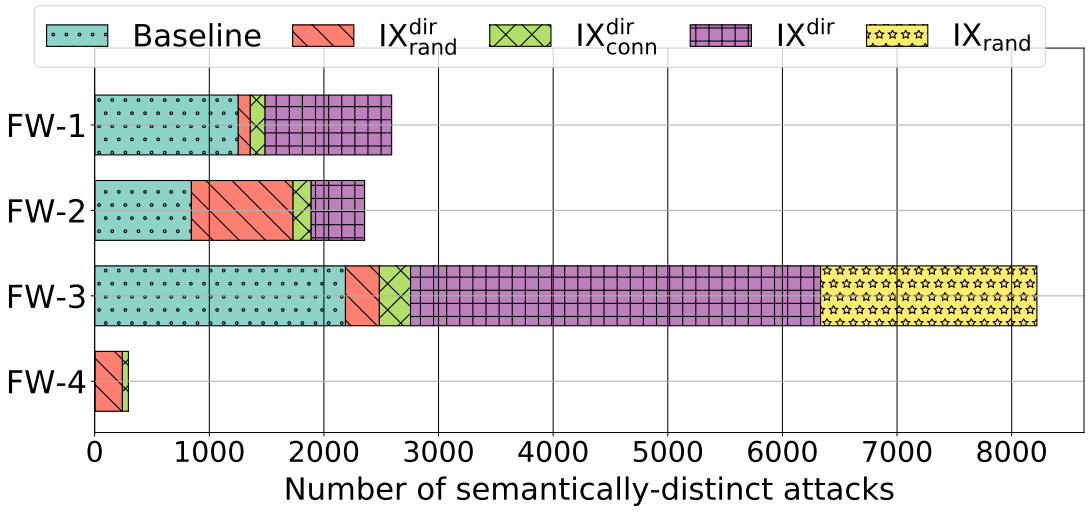


Figure 4.9: Aggregate summary of semantically-distinct attacks found against 4 firewalls across all input templates

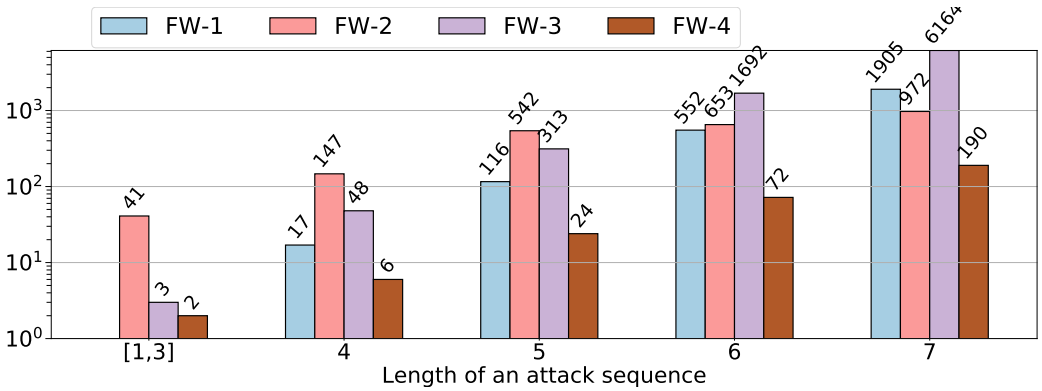


Figure 4.10: Breakdown of the distinct attacks we found for each attack length against all firewalls (Y-axis on a log scale)

Aggregate summary: Figure 4.9 shows the number of successful evasion attacks across all input alphabets. We found 2,591 semantically distinct attacks against FW-1, 2,355 against FW-2, 8,220 against FW-3, and 294 against FW-4. These attacks are loop-free and semantically distinct given Σ (Section 4.4.2). (For attacks across templates, we compressed identical attack strings.) From the Baseline templates, we have 1 attack (out of 294) for FW-4, 1253 (out of 2,591) for FW-1, 844 (out of 2,335) for FW-2, and 1,253 (out of 8,220) for FW-3. The attacks found using

this template are the ones that come from non-traditional sequences of the TCP packets. The rest of the templates help Pryde to discover attacks that come from the TCP packets (with identical 5-tuple) inferring the seq/ack numbers and/or other TCP connection with the reverse direction. For completeness, we also present the number of states for the inferred models in Table 4.1.

	FW-1		**FW-2**		**FW-3**		**FW-4**	
Template	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)
Baseline	8	23	3	12	4	56	2	2
IX^{dir}	14	27	14	15	4	63	2	3
$\text{IX}^{\text{dir}}_{\text{rand}}$	11	36	5	16	10	63	57	N/A
$\text{IX}^{\text{dir}}_{\text{conn}}$	15	50	7	11	10	78	30	N/A
IX_{rand}	10	10	3	11	18	62	55	247

Table 4.1: Number of states for inferred models (N/A means that the model inference did not converge); (1) involves only connection setup, DI, and (2) involves teardown packets, DI-T

The number of distinct attacks is correlated with the number of states (i.e., the complexity of the stateful semantics) and whether that attack can be discovered with the bounded length. We see a relatively smaller number of attacks for FW-4 as (1) the size of the inferred FSMs are smaller in contrast to FW-3 (i.e., 2 for Baseline with teardown vs. 56 for FW-3), or (2) for one large FSM (more than 200 states), we did not find an attack within a bounded length. Here, the take away is that there are hundreds to thousands of attacks leading to circumvention; i.e., patching one such code-path or sequences will be insufficient.

We also summarize the successful attacks based on the length of an attack sequence (Figure 4.10), where the y-axis is in a log-scale. We see a magnitude higher number of attacks for larger attack lengths. While all attacks are equally important, the attacks with longer sequences are likely more *subtle* in exploiting the implementation error/nuances of the firewalls (more detail in Section 4.5.2).

	Attack length							Total
	1	**2**	**3**	**4**	**5**	**6**	**7**	
# generated attacks	1	5	25	125	625	3,125	15,625	19,531
With an insider								
FW-1	0	0	0	0	0	1	2	3
FW-2	1	5	25	125	625	3,123	15,618	19,522
FW-3	0	0	0	0	0	0	0	0
FW-4	0	0	1	3	15	91	586	696

Table 4.2: Number of raw attacks found using random fuzzing

Comparison with strawman solutions: We consider a random fuzzing strategy that randomly generates packet sequences of lengths 1 to 7; the last packet in a sequence is a DATA packet from an external attacker. Hence, we only have one attack for a length of 1. For an attack sequence of a length, $L+1$, we generate $5\times$ the number of attacks for a length of L (giving a total of 19,531 sequences). Now, to pick each TCP packet in a sequence, we randomly sample values from "valid" TCP flags (i.e., SYN $\cdots$, and a DATA packet), a direction (i.e., internal vs. external), and concrete values of seq/ack numbers. Despite the strategy being called the random fuzzing, we only generate "valid" TCP packets (i.e., being generous). Further, as we lack information on the state each packet traverses to enforce the loop-free invariant and the refinement strategy (Section 4.4.2), we only report the "raw" number of attacks. (It turns out non-trivial and there is no one-to-one mapping to project this randomly-generated sequence to our inferred models.)

Note that using an insider was a pre-condition found by Pryde and this unconventional (but realistic) threat model is one of our insights. However, even if we consider the fuzzing strategy with an insider (i.e., being extremely generous to a fuzzing strategy), this strategy is highly ineffective. Specifically, for FW-3 and FW-1, the strategy discovers only 0 to 3 raw attacks (from 20K generated ones). Table 4.2 shows the results. The results, 3 attacks against FW-1 and 0 against FW-3, contrast with 2,591 *distinct* attacks we discovered against FW-1, and 8,220 against FW-3. For FW-4, the random fuzzing found 696 raw attacks (may not necessarily be

distinct). This is natural as the state space of FW-4 is quite simplistic (i.e., only 3 states for the IX^{dir} with teardown packets). Hence, it is relatively *easy* to get to a goal state. FW-2 has a close to 100% success rate. That is, as we will see in Section 4.5.2, FW-2 allows a DATA originating from an external network (even with an explicit "drop" rule). Hence, it is very easy for any strategy to generate working evasion strategy for FW-2. At a high-level, this strategy is ineffective and not robust across firewall implementations.

We briefly also evaluate the strategies found by a related work [69] on censorship evasion, which is an orthogonal problem to our own. The system model in this body of work [69, 179] is considerably different as these censorship firewalls need to allow users accessing (un-censored) contents and, hence, has a default-allow policy. However, we still evaluated the 24 published strategies from Geneva. To map their attacks to our setting, an external web server maps to our internal victim, serving content, and their internal evader maps to our external attacker (evading an enterprise firewall). Across all 4 firewalls, none of these 24 strategies worked (i.e., a victim doest not receive a DATA packet). This is even true for FW-2 as the initial SYN from an external attacker is dropped (due to the default-drop policy). We discuss more about this body of work and also broadly, about applying genetic algorithms or model-free approaches for our problem context in Section 6.3.

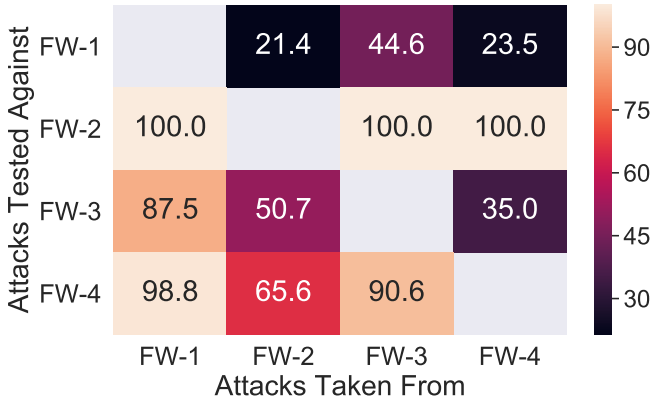


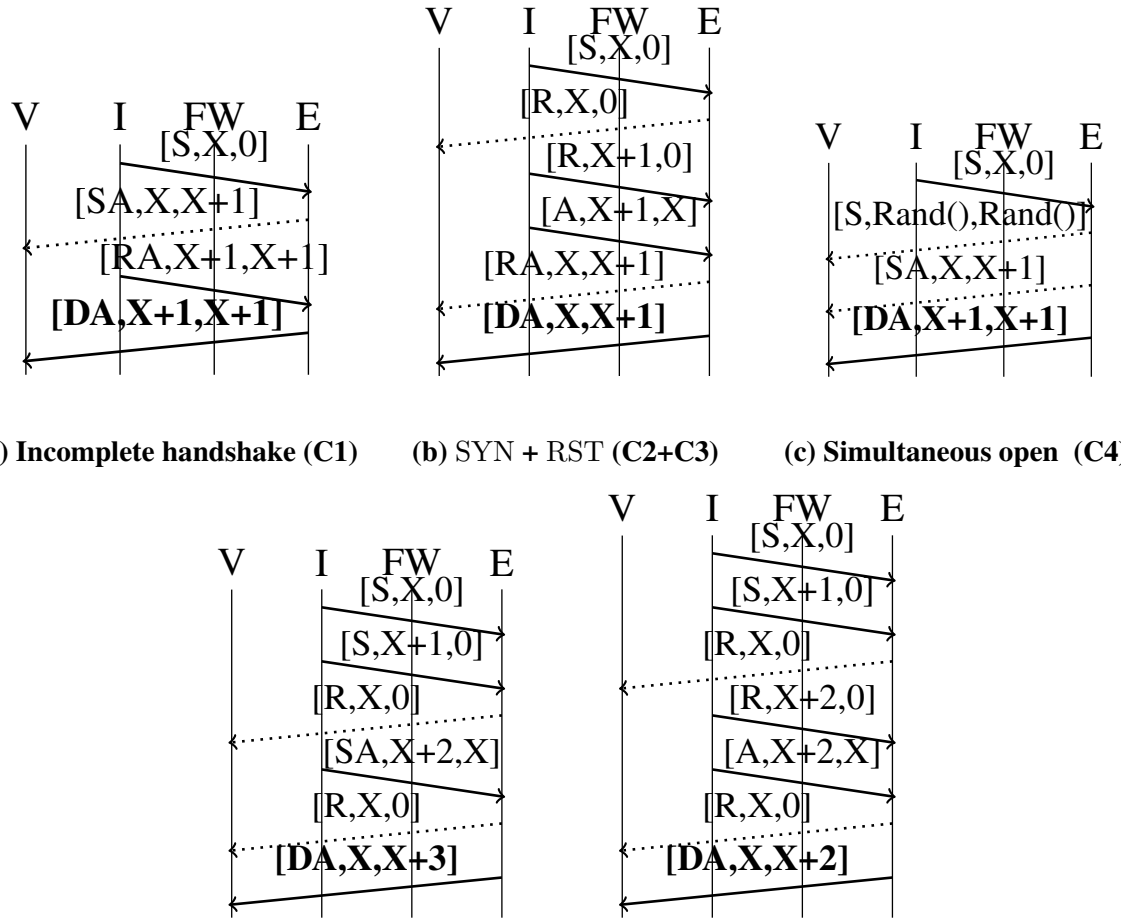
Figure 4.11: Cross-validating the discovered attacks by taking successful attacks against a firewall (x-axis) and testing on a firewall (y-axis) and reporting the attack success rate

Pairwise overlaps of successful attacks: First, we took the successful attack sequences from each vendor and replayed the sequence on other vendors. Figure 4.11 shows the results. For FW-2, attacks from the other three vendors lead to successes. This is because the FW-2 forwards a DATA packet to an internal host in all states (more details in Section 4.5.2). However, other than FW-2, we see low success rates for attacks seen in FW-4 and FW-2 on other firewalls; only 23.5% of attacks from FW-4 work on FW-1. We revisit this when we look at the structure of these attacks in-depth.

4.5.2 Structure of Evasion Attacks

Clustering attack sequences: To help us shed light on the structure of the uncovered attack sequences, for each firewall vendor we cluster the packet sequences as follows. In our clustering formulation, each data point is an attack sequence composed of an ordered sequence of located input packets (Def. 8). From each located packet, for the clustering purposes, we exclude the specific values used for seq/ack numbers but a prefix (that indicates whether the seq/ack numbers was re-written to comply to the TCP semantics). For each pair of sequences, we compute the Levenshtein edit distance. Given this metric, we run a complete-linkage hierarchical clustering algorithm, with a pre-specified target number of clusters. As the attacks differ across vendors, we used a different number of clusters (3 to 7) for each firewall vendor.

For each cluster, we report a concrete attack sequence with the shortest attack length as a canonical example. We also depict other polymorphic variants within the cluster as required. Similar to Section 4.1, we use timing diagrams to specify these canonical attacks. In our diagrams, V refers to the victim, I is the insider, and E is the external attacker. A "dotted" line indicates whether a non-data packet reaches a victim. A bold line means that a DATA packet reaches a victim (i.e., successful circumvention). Further, each label in a line specifies the (TCP flag, seq, ack) from a located input packet; we use abbreviations for TCP flags (e.g., S for SYN, DA for DATA).



(a) Incomplete handshake (C1) **(b)** SYN **+** RST **(C2+C3)** **(c) Simultaneous open (C4)**

(d) SYN **retries (C5)** **(e)** SYN **retries variant (C5)**

Figure 4.12: Evasion attacks against FW-1 across 5 clusters.

FW-1: From 2,591 attacks, we learned 5 clusters of size 2057, 163, 147, 144, and 80, respectively described below:

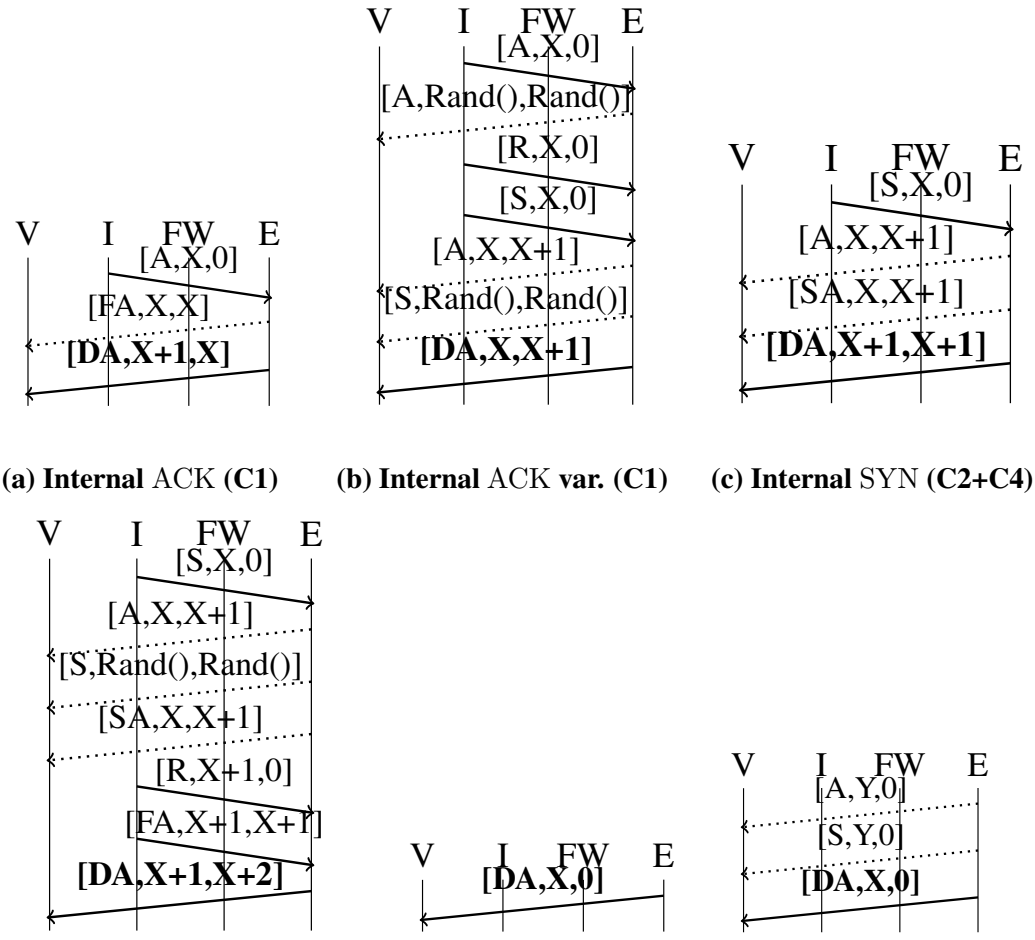
- *C1) Incomplete handshake and variants.* Scenario 1 (Figure 4.4) from Section 4.1 is the shortest-length attack in this cluster. Here, the firewall allows a DATA packet just after seeing an SYN from an insider followed by a SYN-ACK from an attacker. A natural question is whether patching this specific sequence may remove this vulnerability. Unfortunately, this is not the case. Figure 4.12a shows other polymorphic variants that is more subtle and involves the connection state being "disrupted" by a teardown packet (i.e., RA). We also find hundreds

of variants traversing other parts of the firewall state-space; i.e., patching this problem can be non-trivial as an attacker may use other sequences.

- *C2+C3) SYN disrupted by* RST *or* RST-ACK *and variants.* The next two clusters contain attacks involving an initial SYN packet disrupted by an external RST packet (C2) or a RST-ACK packet (C3). The shortest sequence in this cluster is 6, indicating this attack is subtle; i.e., random fuzzing cannot discover these. Figure 4.12b shows that after an insider sends a SYN followed by a RST packet from an attacker, an insider and an attacker exchange three additional TCP packets, leading to circumvention of a DATA packet. There are many variants of this basic attack as well (not shown for brevity).
- *C4) Simultaneous open and variants.* The fourth cluster with 144 attacks exploits how the FW-1 handles the case where two SYN packets are concurrently sent from both directions. (This was found using the IX templates.) Figure 4.12c shows that in the shortest attack sequence. After the first SYN from an insider, the attacker sends a SYN packet, which drives the firewall to another state (i.e., simultaneous open). After that point, the attacker sends a SYN-ACK followed by a DATA packet, reaching the victim. Again, we find many variants that explore the other regions of the state space using a variety of TCP flags (e.g., FIN-ACK, RST, and even DATA packets).
- *C5) SYN retries and variants.* The last cluster of 80 attacks exploits possibly incorrect handling of connection state after SYN retries. Figure 4.12d shows the shortest attack of length 6. We may think that to exploit SYN retries, we need a SYN-ACK to drive the firewall to an incomplete handshake state (similar to C1). However, we also find an interesting variant (Figure 4.12e) that does not involve any SYN-ACK packet to exploit the SYN retries feature!

One invariant we observe here is that the first SYN packet that needs to be sent from an insider. However, as we will see shortly, this is not the case for other vendors (i.e., FW-2).

FW-2: We found 4 clusters of size 824, 806, 422, and 303, respectively. Recall that (Section 4.5.1), FW-2 allows a DATA packet from an external attacker even with an explicit drop rule



(a) Internal ACK **(C1)** **(b) Internal** ACK **var. (C1)** **(c) Internal** SYN **(C2+C4)**

(d) Internal SYN **(C2+C4)** **(e)** DA **allowed (C3)** **(f) External** ACK **(C3)**

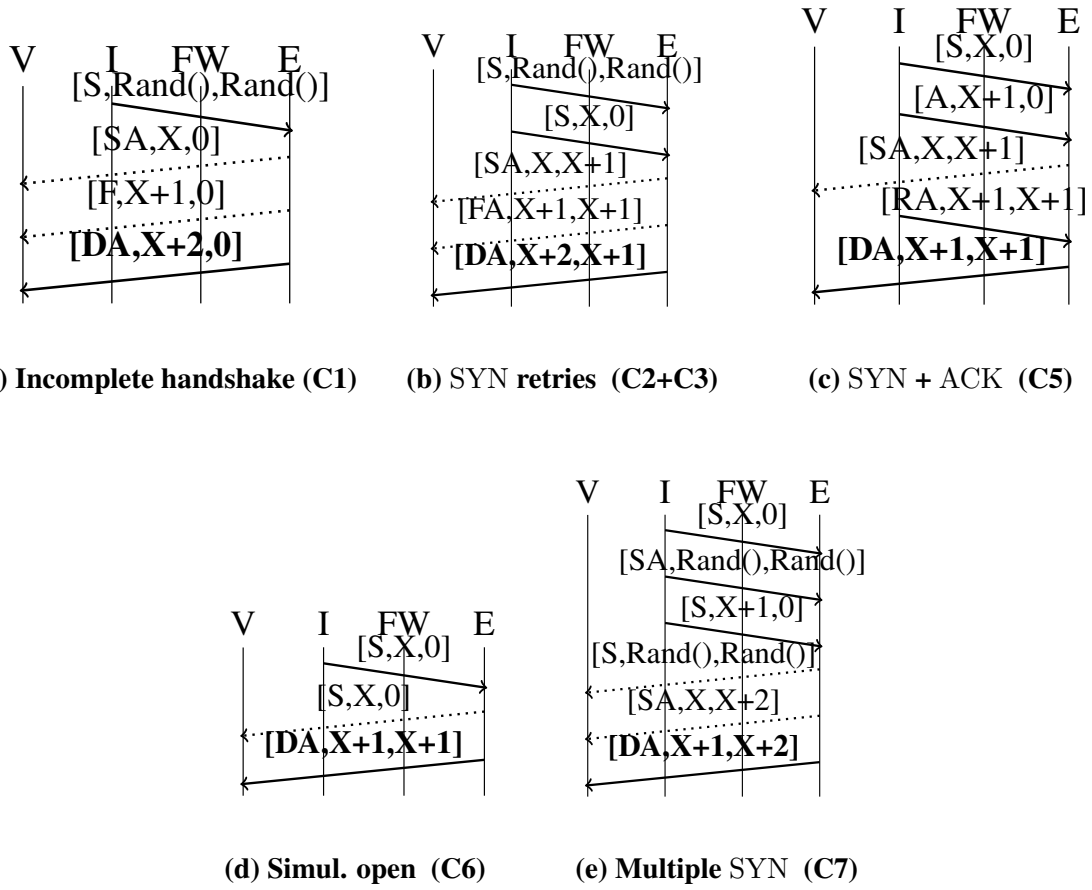
Figure 4.13: Evasion attacks against FW-2 across 4 clusters

(Figure 4.13e).

- *C1) Internal* ACK *and variants.* Attacks in this cluster use an external ACK from an attacker as the first packet. Figure 4.13a shows the shortest example. After an internal ACK followed by an external FIN-ACK (FA) packet, an attacker can circumvent and send a DATA packet. It is surprising that an ACK transitions the connection state without a SYN packet! This is the largest cluster and again has many variants (not shown).
- *C2+C4) Internal* SYN *and variants.* The second and the fourth clusters entail using an internal SYN packet followed by non-traditional packet sequences. Figure 4.13c shows one shortest

example and Figure 4.13d shows an attack of length 7. This is interesting as for the other 3 firewalls, having the first SYN was a requirement but for FW-2, this is just 2 clusters out of 4.

- *C3) External TCP packets with* ACK *flags and variants.* This cluster involves a first TCP packet with an ACK flag (e.g., a DATA packet with an ACK bit or an ACK packet). The shortest attack involves one DATA packet (Figure 4.13e), but there are numerous variants involving a range of lengths.



(a) Incomplete handshake (C1) **(b)** SYN **retries (C2+C3)** **(c)** SYN **+** ACK **(C5)**

(d) Simul. open (C6) **(e) Multiple** SYN **(C7)**

Figure 4.14: Evasion attacks against FW-3 across 7 clusters

FW-3: We identify 7 clusters of sizes 7,621, 212, 198, 63, 58, 37, and 31, respectively.

- *C1+C4) Incomplete handshake and variants.* The attacks in these clusters exploit a connection state being disrupted after an incomplete handshake. Figure 4.14a shows an example where after the initial SYN and SYN-ACK exchanges, a firewall seeing a FIN-ACK packet leads

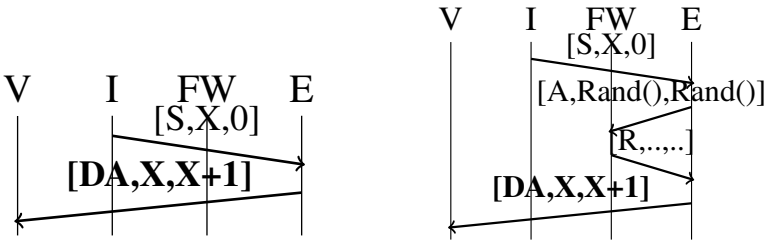
to a circumvention. Cluster 4 is also a special case where the packet disrupted an incomplete handshake is a FIN-ACK packet (Figures not shown).

- *C2+C3)* SYN *retries + an external* SYN-ACK *and variants.* The attacks in these clusters exploit a connection state being disrupted. Figure 4.14b shows an example where after SYN retries, followed by an external SYN-ACK packets and other TCP packets lead to a circumvention.
- *C5) Internal* SYN+ACK *(optional) variants.* The shortest attack in this sequence is identical to that of a FW-1's attack. Specifically, after a SYN followed by a SYN-ACK packet, the FW-3 allows an external DATA packet.

  Other attacks in this cluster exploit a combination of an internal SYN and ACK packets. Figure 4.14c shows such an example. This cluster is quite interesting as these attacks are neither simultaneous open, SYN retries nor an incomplete handshake, but rather some strange packet combinations.
- *C6) Simultaneous Open and Variants.* Figure 4.14d shows an example that only involves 3 attack packets. That is, after the SYN exchanges, the firewall directly allows an external DATA packet. This is in contrast with the attacks against FW-1 (Figure 4.12c) and FW-4 (left of Figure 4.16) exploiting simultaneous sequence; these require longer sequences. However, in the case of FW-3, only after SYN exchanges, an external DATA packet is allowed! There are many variants that also required a longer attack path (now shown).
- *C7) Multiple* SYN *packets and Variants:* Attacks in this cluster involve multiple SYN packets in both directions. Figure 4.14e shows such an example. Explaining this fully is outside our scope; we posit that each packet is responsible for affecting the connection state, and hence, critical in enabling an attack.

FW-4: We clustered FW-4 attacks using 3 clusters. From 294 attacks, we learned 3 clusters of sizes 199, 84, and 11, respectively.

- *C1)* SYN+ *(optional)* ACK *and variants.* Many attacks in this cluster contain some combi-



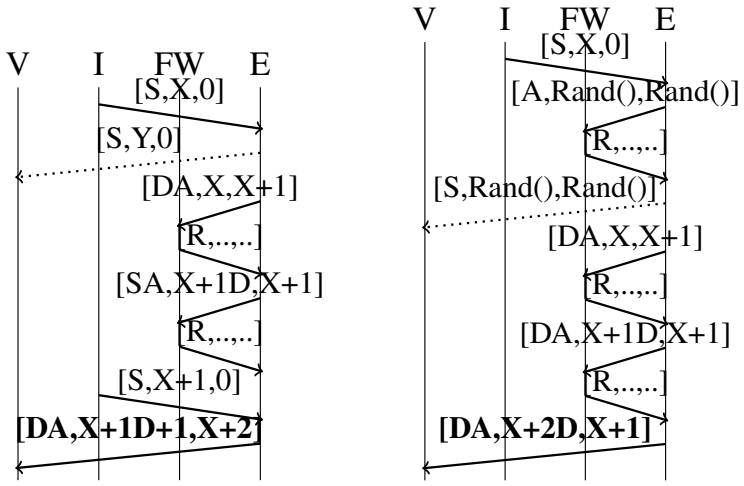
(a) Incomplete handshake(C1) **(b)** SYN**+**ACK **and var. (C1)**

Figure 4.15: Evasion attacks against FW-4 exploiting SYN **+ (optional)** ACK **from (C1)**

nation of internal SYN and ACK packets. Some also exploit an incomplete handshake (Figure 4.15a). After the initial SYN packet from an insider, the FW-4 forwards a DATA packet from an attacker to otherwise an unreachable victim. The right side of Figure 4.15 shows another attack where 3 packets are injected. That is, after the initial SYN followed by an ACK from an attacker, the FW-4 replies with a RST packet. However, an attacker can send a DATA packet. This was flagged as a distinct attack from the previous one as the state space traversed differs.

- *C2) Simultaneous open and variants.* Attacks here exploit the simultaneous open mechanism (Figure 4.16a). Again, the shortest attack length is 6, indicating the subtlety required (and contrasting with FW-3 which had an attack sequence length of 3 as shown in Figure 4.14d). Interestingly, a FW-4 sends RST packets when it sees unexpected TCP packets (unlike, FW-1, for instance).

- *C3)* SYN+ *multiple* DATA *and variants.* The last cluster is interesting in that these attacks use multiple DATA packets (Figure 4.16b). The intermediate DATA packets are required to drive the connection state but are dropped by a firewall (who replies with a RST packet). However, eventually, the firewall allows the third attempt! While omitted for brevity, we find multiple variants.



(a) Simul. open (C2) **(b)** SYN**+multiple** DATA **(C3)**

Figure 4.16: Evasion attacks against FW-4 from (C2) and (C3)

4.6 Other Related Work on Firewall Policy Checking

We now discuss the related work specific to Pryde. Specifically, we discuss orthogonal efforts that test and verify whether the firewall *ruleset* is correctly configured (e.g., [54, 56, 184]). These efforts are orthogonal as Pryde (and other related efforts we presented in Chapter 2) finds attacks that exploit the subtle implementation nuances as opposed to testing or verifying the correctness of firewall ruleset. Many of them use abstractions such as Binary Decision Diagrams (FIREMAN [184]) or a directed graph [55] to verify the FW ruleset. Further, the work by El-Atawy et al. [54] generates test cases to identify misconfiguration by designing a mechanism to reduce the search space. At a high-level, these works are orthogonal to Pryde since they do not focus on discovering implementation errors but on how the rules are configured.

4.7 Countermeasures

In the short term, we envision two fixes. First, vendors could use our generated models and attack strategies and identify bug fixes. While fixing such logic bugs completely may be difficult, as a

starting point, we could focus on fixing bugs up to a length X. Here, the models learned from AmpMap could help narrow down the exact code region or path for any given attack sequence. We envision doing this iteratively; i.e., after patching specific vulnerable sequences, we can rerun AmpMap to validate and identify new evasion attacks. Second, operators can use our attacks to synthesize policies for traffic normalizers [114]. Some of our post-processing analysis for summarizing the patterns of attacks may help in this process (e.g., generating signatures). A longer-term option would be to use some type of program synthesis or formal verification techniques to generate the FSM handling parts of the firewall that are correct by construction.

4.8 Summary

Stateful firewalls are the "workhorse" of operational network security but are surprisingly hard to implement correctly. As such, vulnerabilities in the semantics of the stateful processing can lead to fundamental sources of evasion attacks that can manifest even if the policies are configured correctly. Our work on Pryde automatically synthesizes evasion strategies with a model-guided approach by taking as input only a black-box firewall implementation. Specifically, our model-guided workflow is inspired given that these attacks exploit nuanced implementation errors. Pryde build upon Alembic to enable a scalable model inference to reason about adversarial scenarios. Pryde is extensible and can be used for analyzing a variety of scenarios, though in this work as a starting point we focus on the circumventing a DATA to a victim host. Our analysis of multiple production-grade firewalls reveals that: 1) there are more than hundreds (or a thousand for some cases) of distinct attack sequences for each firewall; and 2) these attacks are subtle that would be difficult to discover if we had done them manually or used structure-free approaches.

Chapter 5

AmpMap: Accurately Measuring Global Risk of Amplification Attacks

The Problem: Many recent high-profile Distributed Denial-of-Service (DDoS) attacks rely on *amplification* [154, 165]. If a source IP address can be spoofed, any stateless protocols in which the response is larger than the query can be abused. While there are a variety of best practices to mitigate this situation [3, 4, 5] given that spoofing is possible, they are unevenly applied. Spoofing the victim's IP may be avoidable in a future Internet (e.g., [61]), but it continues to be possible from a large number of ISPs [15]. Finally, there continue to be many public-facing servers that can be exploited for amplification [165]; many servers do not apply best-practice mitigation (e.g., rate limiting, restricting access).

As networks evolve and server deployments change, the potential for amplification attacks changes over time. For instance, new avenues for amplification emerge (e.g., botnet, gaming protocols) and unexpected vectors for known protocols are discovered [24]. In light of the continued threat of amplification, we argue that we need an Internet-scale monitoring service that can systematically and continuously measure the empirical risk of amplification [11, 18]. We envision a service that periodically maps each server to query patterns yielding high amplifica-

tion and quantifies these amplification factors (AF). Such a framework can serve as an empirical foundation for cyber-risk quantification that many have argued for [10, 14]. Furthermore, it can inform remediation efforts such as throttling servers, generating signatures, informing protocol changes, and provisioning of defenses [10, 14].

At first glance, it seems that we can use or extend existing scanning services that look for and enumerate open/public servers for different protocols (e.g., Censys [95], ZMap [96], `openresolver` [13] monitoring open DNS resolvers, `shadowserver` [29] reporting on open CharGen, LDAP, QOTD, and SNMP servers, among others). For instance, we can (1) multiply the number of open servers with previously reported amplification factors (AF) [10, 165], or (2) extend these scans to probe servers using a set of "known" query patterns (e.g., send ANY requests to DNS servers) to account for per-server factors (rather than using a single global amplification factor for all servers). Unfortunately, these have fundamental shortcomings (Section 5.1.2). These either assume that the amplification that servers yield is homogeneous or that the servers share an identical set of query patterns. In reality, we see significant and unpredictable variability in amplification across servers (including within servers running the same software versions) and query patterns that yield amplification. Thus, these approaches are inaccurate for estimating the empirical risk and for informing remediation efforts.

At the other extreme, we can envision a brute-force approach of sending all possible protocol-compliant queries to servers for each protocol. Unfortunately, the search space of possible queries is large (e.g., NTP has multiple 32-bit fields). We can also consider simple fuzzing or existing heuristic-based optimization techniques but they all have fundamental limitations as the relationship between the packet field values and amplification can be quite complex. This highlights a fundamental tension between the overhead of such an amplification-monitoring service and its utility.

The Solution: In this chapter, we present AmpMap [147], a framework for measuring the risk of amplification with a low network footprint (i.e., $<$ 1.5K queries per server) that accounts for

both the server- and query-specific variability. Our approach builds on key structural properties of amplification-inducing query patterns.

- First, we observe that distinct amplification-inducing query patterns overlap in terms of values in protocol fields. This locality structure suggests that if we find one such pattern, we can potentially uncover other related patterns by changing one field at a time (i.e., assume that each field in N-dimensional space is "independent").
- Second, we use smart sampling strategies to explore the search space of large (e.g., 16 or 32 bit) fields based on the insight that these either do not affect amplification (e.g., $\mathrm{timestamp}$ for NTP), or when they do, have contiguous structure (e.g., edns payload for DNS).
- Finally, even though servers/implementations are diverse, they share some similarities. This helps us further reduce overhead and improve fidelity by sharing insights across servers. (While we acknowledge that these insights may not be universal for all protocols, these hold in practice for many protocols that have been popular targets.)

Findings and Evaluation: Leveraging these structural properties, we implemented AmpMap, validated our parameter settings in lab settings, and ran real-world measurements. Our key findings (Section 5.4) are :

- *Uncovering new patterns and polymorphic variants:* In addition to confirming findings from prior work (e.g., GetBulk for SNMP [5], ANY or TXT for DNS [5, 165, 176]), we discovered new patterns and polymorphic variants (from known ones). Table 5.1 summarizes these findings. We highlight some of the interesting or high amplification-inducing patterns. For NTP, apart from the $\mathrm{MONLIST}$ request, we discover "get restrict" and "if stats" can also incur more than 500$\times$ amplification factor (AF). For SNMP, apart from GetBulk [5, 165], we find that GetNext can incur AF up to a few hundred! For DNS, we also uncover "multiple" patterns (e.g., URI, SRV, CNAME records) that collectively incur 21.9 $\times$ more risk than a popular-known pattern (ANY requests). While some of DNS patterns have been pointed by (mostly) the operational community (e.g., A, RRSIG [166, 176, 180]), many have not been documented

	Known patterns	AmpMap-discovered	
		New patterns	**Polymorphic variants**
DNS	edns:0, recordtype: ANY [3], TXT [28]	edns $\neq$ 0, recordtype: LOC, SRV, URI $\cdots$	rd:0 (off) dnssec:0 (off) ednspayload:<512
NTP	MONLIST [4, 165]	if stats if reload get restrict peer list peer list sum	
SNMP v2	GetBulk [5, 165]	GetNext Get	Vary OIDs; number of OIDs
Chargen	Character request	None	None
Memcached	Stats [5]	None	None
SSDP	Search request [5, 165]	None	ssdp:all upnp:rootdevice $\cdots$

Table 5.1: Summarizing known, unforeseen, and polymorphic query patterns found using AmpMap

to the best of our knowledge. We also discover polymorphic variants due to server diversity; e.g, for GetBulk request, SNMP servers can incur a magnitude "higher" AF with certain OID (object identifier) and the right number of OIDs to query for.

- *Variability across servers and protocols:* We observe significant variability with the AF that each server can yield; e.g., the AF can vary between 0 to 1300 for NTP. This confirms we cannot assess amplification risk by looking at mega-amplifiers or simply counting the number of servers. We also observe substantial variability in the AF distribution across protocols; e.g., DNS servers can yield AF above 100 but 60.4% for Chargen. Such variability across multiple dimensions calls for the need to do periodic measurements rather than one-time analysis.

- *Empirical risk quantification* : By analyzing our data, we unfortunately find that just disabling

the top few known patterns is far from enough; e.g., blocking EDNS0 and ANY or TXT for DNS still leaves 17.9× the residual risk from "other" patterns (Table 5.6). Further, using an additive risk metric (Section 5.1), we highlight the imprecision of the risk estimated by prior work. Even if we focus on the known patterns (e.g., GetBulk for SNMP), existing techniques underestimate SNMP risk by 3.5× and overestimate Memcached risk by 5.6K× and DNS by 1.9×. If we consider new patterns, then the inaccuracy gets worse; e.g., DNS risk is underestimated by 11.9×.

Having summarized our findings, we now put them in a historical context. Specifically, while UDP-based amplification attacks have been known for decades in the security community [154] and exploited at scale [11, 18], AmpMap takes a systematic approach in understanding the attack landscape. In doing so, AmpMap uncovers new query patterns (i.e., new vulnerabilities) that can be exploited for UDP-based amplification attacks and confirms the known vulnerabilities that have not been patched. Further, using AmpMap, we question our prior understanding that only a few patterns yield high amplification by uncovering many other new patterns that can collectively incur even higher amplification that the know ones combined. Given the complexity of the amplification attack landscape and variability of attack vectors across server instances, we need an AmpMap-like Internet health-monitoring framework that can automatically and systematically map out these attacks. Our findings can inform remediation efforts such as generating signatures, throttling servers, and informing protocol changes.

Ethics and Disclosure: We have also disclosed the newly discovered patterns to relevant stakeholders such as CERT, vendors, and IP address owners (Section 5.5.2).

5.1 Background and Motivation

We start with a background on amplification attacks. We then motivate the need for empirically measuring amplification risk and discuss why strawman solutions are insufficient.

Primer on amplification: In an amplification attack (Figure 1.5 from Section 1.1), the attacker spoofs a victim's source IP and sends a small query/request (e.g., 60 bytes) to one or more public servers that act as *amplifiers*. These amplifiers send large responses to the victim. The amplification factor (AF) is the ratio of the query/response sizes; e.g., $\frac{|r|}{|q|} = 100$ in Figure 1.5. AF is also referred as BAF (bandwidth AF) in prior work [10, 165]. (We do not report packet amplification ratios for brevity.) Amplification attacks are well known [154] and have been exploited at scale (e.g., [24, 31, 32]). For example, one of the **query patterns** that induce high amplification for DNS is $\langle$ id:*, recordtype:ANY, edns:0, payload:(1000,65535) $\cdots \rangle$. Here, edns is set to version 0, allowing a DNS server to use the non-default "payload" size and send large responses (default payload is 512-bytes) . The payload is set to greater than 1K (to overwrite the default 512-bytes), and recordtype is set to ANY (all records for a given domain).

5.1.1 Motivating Use Cases

We summarize two motivating use cases as argued by prior academic and policy efforts (e.g,. [10, 14, 165]). For both use cases, there are two relevant aspects for each server/amplifier: (1) *which query patterns* cause large amplification, and (2) *how much amplification* each query pattern induces.

U1) Assessing cyber risk: Network operators need to know whether, and by how much, their deployments are susceptible to amplification. Policymakers need a risk assessment to focus their remediation efforts on the highest priority risk. Given a query pattern, p, for a protocol, $proto$, and a set of servers, S, we define a simple additive risk metric as follows:

$$RiskMetric(p, S) = \sum_{s_i \in \mathrm{S}} \mathrm{AF}(s_i, p) \tag{5.1}$$

Then, given a set of patterns, P, the total risk then is the summation of the risk for each pattern, $p \in P$. Even though this does not consider other factors [10] (e.g., outbound link capacity), it is an instructive metric to quantify risk.

U2) Inform defense efforts: Operators need to know which query patterns induce amplification to take appropriate defenses (e.g., block or throttle responses). Similarly, protocol designers and vendors need to know these patterns to (1) guide the design of future protocols, and (2) assess whether particular remediation (e.g., disabling a feature) can reduce the risk. Lastly, ISPs and operators need to know the degree to which servers are susceptible to amplification to inform capacity provisioning for defenses. For this, the per-pattern risk can also help prioritize the remediation efforts to focus on the largest threats first.

5.1.2 Case for a Measurement Service

Given these use cases, we can consider some seemingly natural strategies derived from (or extended from) from prior work in amplification analysis (e.g., [10, 85, 165]):

- *S1) Scan for open servers* : Using a count of the number of open servers, we can multiply this number by a fixed known AF (e.g., 556 for NTP [33]). For instance, if there are 1M open NTP servers, this approach would multiply 1M by 556 AF; for a 50 bytes request, this translates to 27.8 billion bytes. Such information can be used for risk quantification (U1) and for informing network operators of their servers (U2) akin to existing efforts (e.g., [10]).
- *S2) Probe servers using fixed patterns* : (S1) assumes that servers have identical risk and does not account for multiple patterns. A more advanced strategy is to probe servers using previously known patterns and record their AFs (e.g., DNS [175], NTP [85]). Then, we can use this to assess risk (U1) and construct signatures (U2). There can be different options for choosing which patterns to probe (e.g., taking the known patterns, taking the top-K patterns from random sampling).
- *S3) Customize S2 for different server software*: (S2) did not account for the variability of query patterns across servers. If servers with the same software setup have similar patterns, then we can run (S2) once for each "software setup" (e.g., Bind 9.3, Dnsmasq 2.76). That way, we can reduce the number of probes we send.

To understand if these strategies are effective, we run a small-scale measurement study using DNS (as an example) as its amplification properties are seemingly well understood [33, 165]. We identify a set of 172 queries based on three fields ($\mathrm{recordtype}$, edns, rd or recursion desired) that are known to affect amplification [3, 5, 165]. (We generated 172 queries using combinations of 43 values of $\mathrm{recordtype}$=$\{$A, NS, CNAME, $\cdots\}$, edns=$\{0, 1\}$, and rd=$\{0, 1\}$.) (As we will see later, that these 3 fields do not represent the full set of fields that affect amplification. Rather, we use this as an illustrative set of query patterns to highlight why these strategies are imprecise.) Then, we pick a random sample of 1K DNS servers from Censys [95], send each of the 172 queries, and record the AF per query. We also obtained the version string (if available) for each server using Nmap.

In this dataset, we observe 94 unique patterns that incur $\geqslant \delta$ AF (δ=10) with a total risk of 125.8K AF (using Eq. 5.1); if these servers are connected to a mere 10 Mbit/sec connection, 125.8K translates to 918 Gbps across 1K servers.[1] Using this "ground truth," we evaluate the above strategies using two metrics: (1) the risk estimation accuracy (for U1); and (2) the number of query patterns missed (for U2).

Strategies		% Error in Risk (U1)	# missed patterns (U2)
S1	Scaling by number of servers	4.5× ↓	N/A
S2	Using known patterns	5.7× ↓	90 (out of 94)
	Top-K from random samples	20× ↓	86 (out of 94)
	Top-K from ground-truth data	3.6 × ↓	84 (out of 94)

Table 5.2: Effectiveness of S1 and S2 in enabling use cases

Table 5.2 summarizes these metrics for S1 and S2. For S1 of multiplying by a known AF factor, we use a factor of 28 as reported earlier [3]. For S2, we considered three possible instantiations: (1) using known query patterns from prior works (edns 0 and $\mathrm{recordtype}$ ANY or

[1] 60 bytes/query × 128.5 avg AF / server × 1K servers × 8 bits/byte × 14,880 query/sec (using 10 Mbps and a frame size of 84 bytes).

TXT [3, 176]), (2) using the top-10 queries across servers w.r.t. the AF values across servers after randomly sampling 20% of the possible values of three fields space; and (3) using the global top-10 patterns from the entire data. Note that (2) and (3) are extremely generous; in practice, we do not know the global top-10 a priori, and the actual space of queries is much larger than just 172 queries. We see that S1 of scaling server count under-estimates the risk by 4.5×. Depending on the scaling factor, the risk may also be significantly over-estimated. S2 also under-estimates the risk (U1). We also see that S2 misses many query patterns (U2).

We also observe that this aggregate estimation error across 1K servers translates to large percentages (%) of residual risk for "each server" (if we had used S2). Consider the cumulative distributive function (CDF) of the % of the residual risk for each server. 50% of the servers would have: (1) $\geqslant$ 68% residual risk (if we had blocked the top-10 patterns from the ground-truth, which is infeasible in practice), (2) $\geqslant$ 72% residual risk (if we had blocked only the known patterns), and (3) $\geqslant$ 82% residual risk (if we had taken top-10 patterns after random sampling the header space). The trend does not really get better, even if we had used other top-K (e.g., 20).

Finally, Table 5.3 shows the ineffectiveness of S3 for top-5 version (ranked by the number of servers that have at least one query that induces $\mathrm{AF} \geqslant \delta$ in the dataset). Here, we define that servers have identical software setup if they share the same vendor and a major version.

	% Error in Risk Estimation (U1); (# of missed patterns / # total patterns) (U2)				
	Microsoft 6.1	**Dnsmasq 2.52**	**Dnsmasq 2.40**	**Dnsmasq 2.76**	**Bind 9.9**
Using known patterns	14.4× ↓ (76/80)	2.7× ↓ (27/31)	6× ↓ (38/42)	3.8× ↓ (44/48)	8.8× ↓ (72/76)
Top-K from random samples	8.7 × ↓ (70/80)	3.6 × ↓ (27/31)	44.2 × ↓ (41/42)	31.6 × ↓ (45/48)	7 × ↓ (66/76)
Top-K from groundtruth	4.5× ↓ (70/80)	1.2× ↓ (21/31)	3.8× ↓ (31/42)	1.7× ↓ (38/48)	6× ↓ (66/76)

Table 5.3: Effectiveness of S3 that does per-version analysis

To understand why these strategies are inaccurate, we analyzed this data further. To explain our analysis, we define some terms. Given a server, s_i, let Q_i be the set of queries that incur AF $\geqslant \delta$; i.e., Q_i is the set of queries that elicit large responses. Given n servers, let Q be the union of $Q_1 \cdots Q_n$; i.e., Q is the union of all amplification-inducing queries.

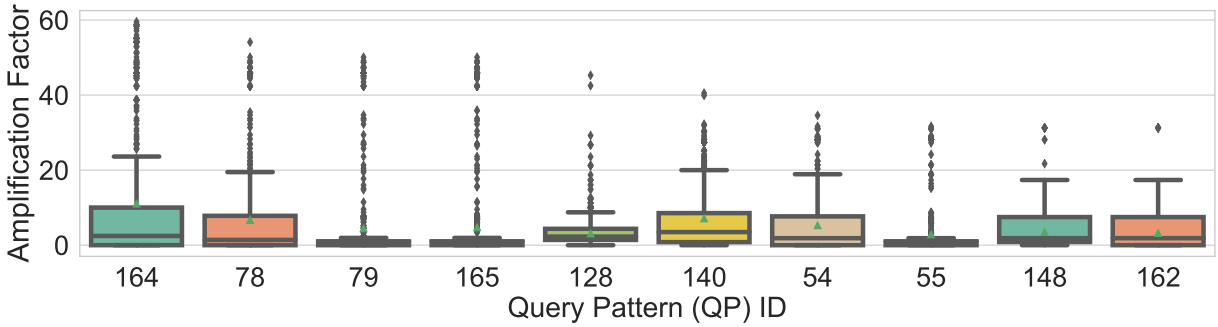
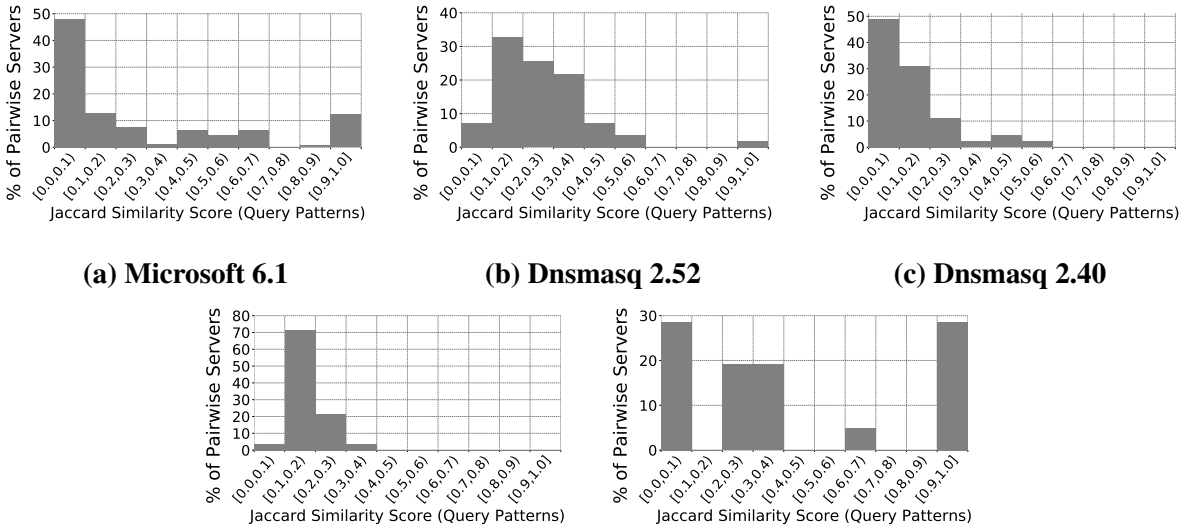


Figure 5.1: Diversity of AF given a query across servers

Variability in magnitude across servers: Figure 5.1 shows the distribution of the AF value across servers. Due to space, we only show this for 10 queries that induce the highest AF if sorted by the AF across our data-set. For a given q, the standard deviation ranges from 3.9 to 17. Looking beyond the global top 10, if we just consider maximum AF for each server (across all 172 queries), there is significant variability as well with a standard deviation of 16.7. This also holds for servers sharing the same software versions (not shown).

Variability in query patterns across servers: If only a small subset of patterns induce amplification on *all* servers (i.e., Q_i are identical), then (S2) and (S3) would have been sufficient. To this end, we analyze the similarity (or lack thereof) of query patterns across servers in two ways. Let $\text{TopK}(Q_i)$ denote a set of Top-K queries when Q_i is sorted by the AF value. Then, we analyze: (1) How similar are high-amplification query patterns between every pair of servers (i.e., $\text{TopK}(Q_i)$ from $\text{TopK}(Q_j)$)? (2) How similar is a server-specific query pattern set $\text{TopK}(Q_i)$ to the global $\text{TopK}(Q)$? We compare the top-K queries where K=10. Note that we are not just looking at the "maximum" query (i.e., K=1) as we want to consider multiple patterns. We observe the same trend holds for varying Ks such as 5, 20 (not shown).



(a) Microsoft 6.1 **(b) Dnsmasq 2.52** **(c) Dnsmasq 2.40**

(d) Dnsmasq 2.76 **(e) Bind 9.9**

Figure 5.2: Histogram showing the Jaccard similarity scores between Top-10 query patterns of pairwise servers

If we look at the histogram of similarity score for K=10, more than 60% of server pairs have low similarity scores $\leqslant 0.2$ and only 4% of server pairs have above 0.8 similarity scores. This trend is also similar for servers with "identical" software (Figure 5.2); e.g., for Microsoft 6.1, more than 45% have similarity $\leqslant 0.1$. For the question (2), compared to the global $\mathrm{TopK(Q)}$, we find that more than 70% of servers' $\mathrm{TopK(Q}_i\mathrm{)}$ have $\leqslant 0.2$ similarity scores.

Taken together, these results suggest that we cannot homogeneously attribute the same risk per pattern and across servers. Furthermore, we cannot just look at a single server instance (or one per software version) for our use cases. Given this empirical variability across servers, query patterns, and the AF values, we argue that we need an active measurement framework that can quantify the risk and inform defenses for amplification attacks.

5.2 AmpMap Problem Overview

The previous results showed us that we need a measurement service. Next, we formulate the goals for such a service we call AmpMap and discuss the challenges in realizing them.

5.2.1 Problem Formulation

We consider S servers implementing protocol $proto$. For each server, $s \in S$, our goal is to uncover as many *distinct amplification-inducing query patterns* as possible (e.g., say $\mathrm{AF} \geqslant \delta$=10) while keeping our network footprint low. These per-server patterns output by AmpMap can inform our use cases such as assessing risk and informing defenses. Intuitively, each pattern is a template for describing protocol queries, where each field takes a value or a contiguous range; queries in the same pattern trigger similar protocol behavior and hence, have similar AFs (formal definitions in Section 5.3.3).

We obtain the list of open servers implementing a given protocol from public services (Shodan [30], Censys [95]). We prune out inactive protocol servers or servers owned by military or government. Each protocol is defined by a set of `fields` ($F = \{f_1 \cdots f_n\}$), and a set of `accepted values` for each field ($AV(f_1) \cdots AV(f_n)$). We obtain the protocol format from protocol specifications (e.g., RFCs). For example, DNS defines fields such as dnssec, id, and their accepted values; e.g., dnssec takes a value from $\{0, 1\}$. A valid query of $proto$ is a list of values for each field (f_i=$v_i \in AV(f_i)$) and a valid query returns a response. To avoid malformed queries that may impact server operation, we only consider valid queries. We do not include derived fields (e.g., checksum, count-related fields). Some fields take a value from a set of strings (e.g., domain for DNS, OID for SNMP). For these, we sample values; e.g., for DNS, we take popular domains from different industry sectors (e.g., education, health care) and with different features (DNSSEC-enabled vs. not). To this end, we keep the set of values for these fields to be small (a few tens). For the fields that take a "list" of values (e.g., OID list for SNMP), we also specify a "length" of a list as an input (Section 5.3).

To keep our footprint and impact on servers low, we impose a query budget for each server, B_{total} (400–1500 range as we will discuss in Section 5.4). We also consider additional precautions e.g., limit rate per server (1 query per 5 s), number of servers probed concurrently, and avoid invalid/malformed requests (more details on precautions in Section 5.5.1).

Scope: We focus on *stateless* and *unicast* protocols (e.g., UDP) and stateless amplification strategies. Thus, stateful protocols (e.g., TCP-based) and broadcast or multicast protocols (e.g., [135]) are out of scope. Additionally, stateful attack strategies that seed entries to a server and subsequently launch a high AF query are outside our scope; e.g., we do not consider an attacker who registers his own domain for DNS with many records to amplify the attack.

Fields: $F = \{f_1, f_2, f_3, f_4, f_5\}$

Accepted values for each field: $AV(f_i)$

1. f_1 takes a value from 0 to 1; $AV(f_1) = [0, 1]$
2. f_2 takes a value from 0 to 99; $AV(f_2) = [0, 99]$
3. f_3 takes a value from 0 to 65535; $AV(f_3) = [0, 65535]$
4. f_4 takes a value from 0 to 7; $AV(f_4) = [0, 7]$
5. f_5 takes a value from 0 to 1; $AV(f_5) = [0, 1]$

Figure 5.3: Simplified protocol definition to highlight challenges of uncovering amplification queries

5.2.2 High-Level Challenges

We now discuss three key challenges in achieving our goal. To illustrate these concretely, we consider a simplified protocol inspired by the structural properties of real protocols. The protocol is shown in Figure 5.3 and consists of 5 fields with their accepted values. Figure 5.4 represents the structure of *amplification-inducing query patterns* for a single server s_1 varying two of these fields, f_2 and f_3, while fixing values of the other three fields. The left side is when f_1=0, and the right side is when f_1= 1. In both cases, f_4 and f_5 are 0 and 1000, respectively. Each such "red" (darker) region in these heatmaps is a potential *query pattern*. Even this relatively simplified protocol highlights several key challenges. We observe these challenges across protocols we surveyed (especially for more complex protocols like DNS and NTP). The challenges we discuss

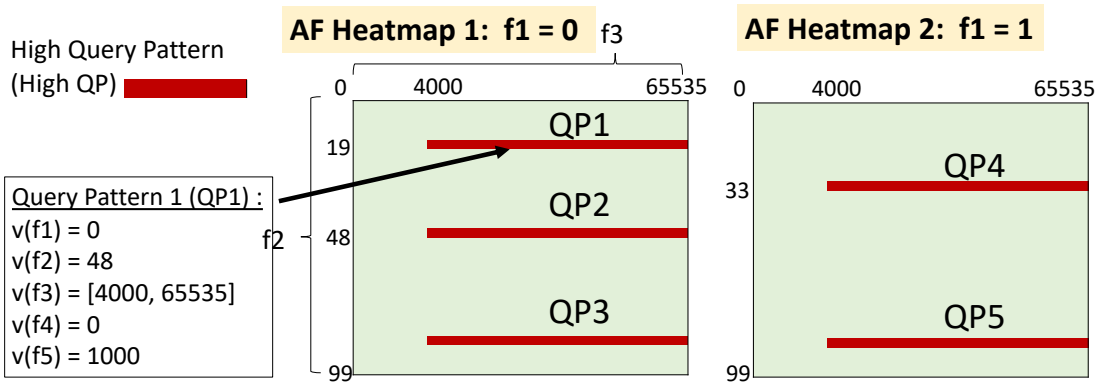


Figure 5.4: Query space for one server, server 1 (s_1). QP_i refers to a query pattern

below for AmpMap map directly to three high-level challenges we discussed in Chapter 1:

- **C1:** We observe a *large query space* of $2 \times 100 \times 65K \times 8 \times 2 >$200M values; i.e., it is infeasible to exhaustively explore this space.
- **C2:** Even for a single server, we observe the structure of amplification can be *complex* as the fields in a query are dependent on each other and need to be simultaneously set. For instance, in QP_1 (Figure 5.4), both f_2 and f_3 need to be set to 48 and [4K, 65535], respectively, to yield high AF. Intuitively, in real protocols, such behavior occurs as certain flags need to be set to trigger a relevant behavior; e.g., in Section 5.1.2, we need to set edns to 0 and/or rd to 1 for *certain* servers to yield large AF. Also, note the relationship between the query and AF does not necessarily have a nice continuous structure. Worse, our goal is to uncover as many patterns as possible in this complex, multi-field search space, making the problem even more challenging.
- **C3:** As we saw in Section 5.1.2, across servers, the exact AF for a given query may differ and the set of query patterns also may differ. Figure 5.5 shows the structure for three servers (including s_1) for the case when f_1 is set to 1. In our simplified protocol, queries in $\mathrm{QP1}$ for s_1 incurs high AF for s_2 (i.e., $\mathrm{QP1}$) but not for s_3. Due to a server configuration and the view of data a servers has (e.g., the number of peers for the NTP server), s_3 does not have any query patterns that cause high AF.

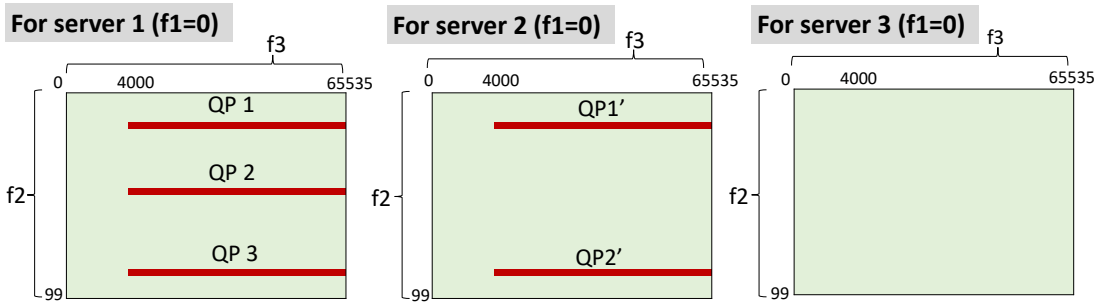


Figure 5.5: Query space across multiple servers, only showing the case when f_1=1 (i.e., Heatmap 1 as in Figure 5.4)

5.3 AmpMap Overview and Design

In this section, we discuss our key insights regarding **structural properties** of amplification patterns common to many protocols that enable a practical design. We start with a single server case (Section 5.3.1) and use that to build a multi-server solution (Section 5.3.2). Then, we provide an analysis of why AmpMap can discover amplification-inducing patterns and compare it with other strawman solutions (Section 5.3.3).

5.3.1 Single-Server Algorithm

Before we explain our insights, let us consider two seemingly natural baselines and see why these are not practical (we empirically confirm this in Section 5.4).

1. *Random fuzzing:* We can randomly pick a field value to construct a query. Unfortunately, achieving coverage across "distinct" pattern would be prohibitively expensive (analysis in Section 5.3.3); e.g,. if there are 10 patterns and the density of each pattern to the total query space is 0.1 (ϵ), then we need $>$ 29K queries to discover all patterns.
2. *Heuristic optimization techniques:* Existing heuristic optimization techniques (e.g., simulated annealing) may find only a few patterns. However, these are ill-suited to achieve coverage as these getting stuck in "local" optima.

5.3.1.1 Single-Server Insights

Next, we present our insights to make the problem tractable. At a high level, these insights were derived from a combination of simple analysis, local server experiments, and the measurements we saw in Section 5.1.2.

Insight 1 (**I1**): ***Amplification-inducing query patterns exhibit locality and overlap in the values for fields.***

Intuitively, we observe that query patterns often share a subset of specific field values. This suggests that given a query, q, in one of the amplification-inducing query patterns, we may not need to change all N fields at a time. Rather, we can discover other nearby patterns by sweeping one field at a time. Conceptually, we can view the query space as a `logical graph` and look for "neighboring" queries that differ in the value of just one field to discover other patterns. Figure 5.6 shows a logical graph representation of the query space for the abstract protocol (in Figure 5.4). In this graph, each node is a query and an edge between two queries, q and q', means they differ in only one field(e.g., f_2). For instance, from a query in $\mathrm{QP1}$, a simple per-field search approach as described above can discover queries in $\mathrm{QP2}$ and $\mathrm{QP3}$ by changing f_2. To discover $\mathrm{QP5}$, we need to search f_1 from a query in $\mathrm{QP3}$.

Insight 2 (**I2**): ***If the density of amplification-inducing queries is*** $> \epsilon$***, then random sampling will likely find one such query using*** $\geqslant \frac{1}{\epsilon}$ ***queries.***

This is a very simple probabilistic analysis insight. If the overall density of the queries that give high AF is ϵ, then the probability of picking one such query is ϵ. Then, the expected budget to find one such query is $\frac{1}{\epsilon}$; i.e., if a probability of a picking an amplification-inducing query is $\frac{1}{1000}$, then we need an expected budget of 1000 samples. This analysis suggests a viable path to find at least one query in one of the amplification-inducing query patterns, which can subsequently be used to exploit the above locality structure.

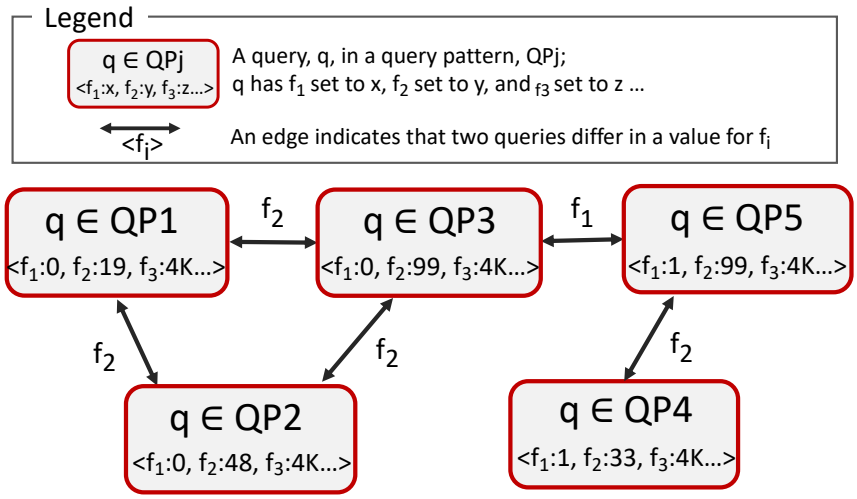


Figure 5.6: Viewing the query space as a logical graph (for the abstract protocol shown in Figure 5.4)

Insight 3 (**I3**): ***Fields with large accepted value ranges either do not affect amplification or exhibit contiguous range structure w.r.t.*** AF**.**

Even if we use **I1** and only need to vary one field value at a time, we still may require a high query budget as some fields take a very large set of accepted values. Fortunately, many of the large-range fields tend to not affect amplification. If they do, we observe that there is a large contiguous "range" (e.g., f_3 with [4K, 65535]) that exhibit similar behavior. For instance, as long as the EDNS payload is set to a large value (i.e., 4096), an edns feature will allow large responses. Thus, instead of exhaustive sweeping, we can sample values for large fields. Specifically, we use a logarithmically-spaced sampling strategy so that if the ranges are sufficiently large, then we will get at least one query from a contiguous range.

5.3.1.2 Single-Server Workflow

Putting this together, we present our workflow for a single server (left side of Figure 5.7). We present the pseudocode of the main function in Algo. 3. Recall that given a fixed query budget, B, we want to maximize coverage of distinct query patterns. In choosing B, we want to strike a

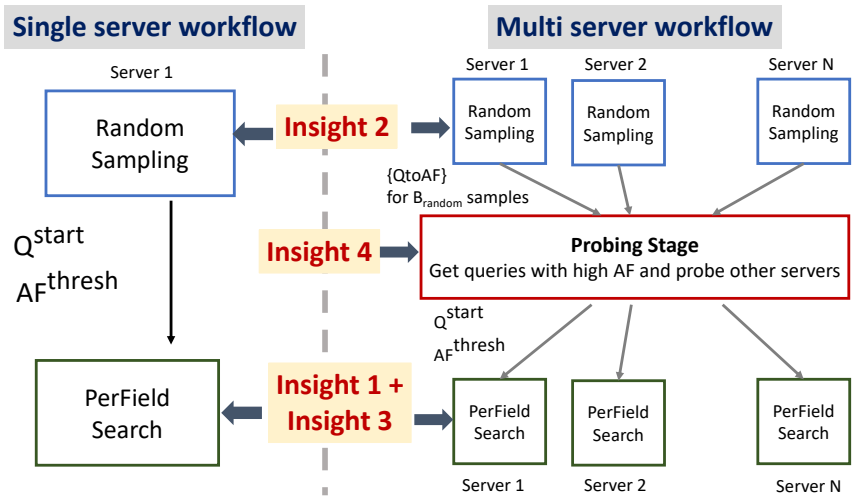


Figure 5.7: AmpMap Workflow

balance between coverage and network load. Our goal is not to find optimal parameters, but to use reasonable ranges that work well in practice. For relatively complex protocols like DNS, we empirically find that 1200-1500 is a good operating range as we see diminishing returns beyond this (Section 5.4.7). For simple protocols with a smaller search space, this property still holds.

RandomSample Stage (Line 2 in Algo. 3): Given a fixed B_{total}, the algorithm randomly samples B_{rand} queries with the goal of discovering an amplification-inducing query (**I2**). The discovered queries are the starting points to run the next phase, per-field search, to improve coverage. For choosing a B_{rand}, we empirically observe that choosing 10% to 45% of the total budget is sufficient (Section 5.4.7). This is because we just need to find one (or a handful) query that induces amplification given the locality structure. As we will later, we can use multi-server experiments to make this further robust to potential mis-estimation of the B_{rand} needed for a server; i.e., even when the random search fails to find a feasible starting point (Section 5.3.2).

Per-field search (Line 4 in Algo. 3): We then run the `Per-field search` leveraging **I1**. Algo. 5 shows a detailed pseudocode. It takes an input of $QtoAF$ which contains each query to the AF from the random stage. We also need to determine other relevant input parameters.

- *Starting queries for per-field search (Q^{start})*: We pick top K queries w.r.t. the AF values. As

we just need one or two queries with high AF, we find choosing 1 is sufficient.

- *Threshold to prune low* AF *queries (*AF^{thresh}*)* : If neighboring queries have AF below AF^{thresh}, the per-field search prunes them from further exploration. If the value is too low, the search will degenerate into an exhaustive search. If too high, the search terminates without exploration. As a practical trade-off, if the maximum AF is above $2 \times \delta$, we make the threshold to be δ (i.e., 10). If it is below $2 \times \delta$, then we use a threshold equal to some fraction of the maximum AF observed in the random stage (e.g., half).

Using each query from Q^{start}, the per-field search searches the neighboring queries by varying one field value (SEARCHNEIGHBOR$(...)$ in Line 7. It uses a log-sampling for large fields and exhaustive search for other fields. Further, for fields that take a set of strings as an input (e.g., domains for DNS), we recommend inputting an accepted set as a small set (i.e., few tens). This is a conscious decision as such fields tend to not have a "contiguous" structure w.r.t. the AF and each concrete value has a distinct semantic. Hence, we need to treat these fields as small fields (where we do an exhaustive search). For fields that take a list as an input (e.g., SNMP's variable bind list takes a "list" consisting of OID), we search over both the item (OID) and the size of the list (i.e., 0-256). For this field, it is worthwhile to see how the protocol behaves when this list size is large. Hence, we recommend putting non-small value (i.e., $\geqslant$256) so that we can log sample the values.

Avoiding already-visited pattern: We have one more practical challenge as each query pattern consists of tens of thousands of queries. Some field take ranges (e.g., f_3 had [4000, 65535] to denote a pattern). If we naively explore, we may redundantly explore other queries in the same query pattern, wasting our query budget. To avoid this, we heuristically detect if we have already explored a pattern to decide if we can skip exploring this further. To do so, we infer the contiguous range of a field that incur above-the-threshold AF as we sweep each field. When we need to explore a query, q', we first check whether q' has already been visited (ISNEWPATTERN$(\cdots)$, Line 5) and only explore if it was not. During the per-field search, we refine the inferred pattern

structure as we get a new range that contains the old range. The search terminates if the budget is exhausted or there are no more queries to explore.

Let's look at a concrete example using the abstract protocol (Section 5.2). Suppose we are currently exploring a query q, $\langle$ f1:0, f2:48, f3:6000 $\cdots\rangle$, from a QP 2. When it is a turn to explore f_3. we *log sample* f_3 to obtain the AFs and find that [5K, 65535] has contiguously "high" AFs. Then, we use this range to describe the pattern structure (i.e., $\langle$ f1:0, f2:48, f3:[5K, 65535] $\cdots\rangle$). We first check whether this is contained in already-visited patterns and if not, continue exploring further. For completeness, we show the full pseudo code in Algo. 3.

Algorithm 3: AmpMap algorithm for a single server

Input : B: query budget, $AV(f_i)$ for $i = 1, .., n$: accepted value for each packet header field

Output: $QtoAF$: maps each query to corresponding AF

`/* Step 1: Random Search */`

1 $QtoAF$ = RUNRANDOMUPDATEMAP(B_{rand})

2 Q^{start} = FINDTOPKQUERIES($QtoAF$,$K = 1$)

3 AF^{thresh} = COMPUTETHRESH($QtoAF$) `/* Step 2: Local Search */`

4 LOCALSEARCHUPDATEMAP($QtoAF$, Q^{start}, AF^{thresh})

5.3.2 Multi-Server Algorithm

We now discuss how we extend the insights and workflow from a single-server case to handle a multi-server case.

5.3.2.1 Multi-Server Insights

Insight 4 (I4): ***While servers exhibit variability, some share a subset of amplification-inducing queries.***

Recall the abstract protocol on multiple servers in Figure 5.5. In that example, the queries

in $\mathrm{QP1}$ for s_1 also incur high amplification for s_2 but not for s_3. While these servers are not "identical" in all query patterns that induce amplification, a subset of these servers can share a subset of query patterns (even if the specific AF values may differ). We also have observed this in our small-scale experiment in Section 5.1; while the similarity of query patterns between a pair of servers is low, it is not always 0. This is natural as these servers implement the same protocol. Leveraging this insight, we can maximize the "insights" across servers; i.e. if we find a query that incurs high AF in one server, we add an additional *Probing stage* that tests these queries to other servers. Then, we add these queries to the pool of "potential" starting points for each server. This can boost coverage across servers while accounting for server heterogeneity, as we still run a per-server per-field search.

5.3.2.2 Multi-Server Workflow

As before, we run the RandomSample Stage per server as in the single-server case. The key addition is a new stage called the `Probing stage` (Figure 5.7), that ensures that the insights are shared across servers. Specifically, using the high-amplification queries found for each server (from the RandomSample Stage), we test them on other servers to increase the chance of finding good starting queries for each server.

Probing Stage: Turning this idea into practice, we take all queries that give "high" AFs across servers from the RandomSample Stage. Then, we pick a small number of queries to probe other servers (i.e., B_{probe} queries). A relevant question is how many queries to use for B_{probe}. We observe that anywhere between 5% to 30% of the total budget is sufficient where we chose 10% (validation in Section 5.4.7). Specifically, we do not want to assign too much for this value to ensure that we have sufficient available budget for other (critical) stages; i.e. the Probing stage is designed to "supplement" the RandomSample Stage for certain servers where the RandomSample Stage was could not discover amplification-inducing queries. The next relevant question is *how* to pick these probing queries. Consider a strategy where we just pick top-X queries w.r.t.

the AF. This strategy may "overfit" to a specific query pattern or certain servers with "many" AF-inducing queries. We want to use a *diverse set of probing queries*. To this end, we take all queries with AF above the threshold (e.g., δ), and then run a simple K-means clustering where we conservatively set the number of clusters K (e.g., 20). (To run K-means clustering, we define our custom distance function. We normalize N fields and then bin the large fields.) To achieve diversity of patterns, we sample queries such that we have at least one query from each cluster and for the remaining ones, we uniformly sample queries proportional to the cluster size. While the number of clusters does not really affect the coverage, the fact that we *use* probing queries boosts the coverage (Figure 5.18 in Section 5.4.7).

The rest of the algorithm mirrors the single-server approach to pick starting points and run the per-field search. However, the input parameters (i.e., Q^{start}, AF^{thresh}) are server-specific to account for server diversity. The only difference is that the top-K starting points are based both on the original set of random queries and the new additional B_{probe} queries. We show the pseudocode in Algo. 4 for completeness.

5.3.3 Analysis of Our Approach

To make analysis easier, we make two simplifying assumptions: (1) We only consider a single-server case; and (2) The ratio of the number of high AF queries to the total number of possible queries, d , is known.

Definitions: We first give necessary definitions for the formal analysis. We use query ranges to denote a set of queries. Particularly, we write a query range QR as $\langle f_1 : [v_1^l, v_1^r], f_2 : [v_2^l, v_2^r], \ldots, f_n : [v_n^l, v_n^r]\rangle$, where $v_i^l, v_i^r \in AV(f_i)$ and $v_i^l < v_i^r$ for $i = 1, ..., n$. A query range represents a set of queries in a natural way. A query $q = \langle f_1 = v_1, .., f_n = v_n\rangle$ is in QR (written $q \in QR$) iff $v_i^l \leqslant v_i \leqslant v_i^r$ for $i = 1, .., n$.

Given a constant δ, a δ-high query pattern (or simply high query pattern if δ is clear from the context) QP is a query range $\langle f_1 : [v_1^l, v_1^r], f_2 : [v_2^l, v_2^r], \ldots, f_n : [v_n^l, v_n^r]\rangle$ satisfying the

Algorithm 4: AmpMap algorithm for multiple servers

Input : B_{total}: query budget

$AV(f_i)$ for $i = 1, .., n$: accepted value for each packet field

S: a set of servers

Output: PerServerQToAF : maps each query to corresponding AF

$PerServerQToAF$ = {} /* Step 1: Random Search */

for $s \in ServerSet$ **do**

  RunRandomUpdateMap(B_{rand},$PerServerQToAF[s]$)

/* Step 2: Pick probes based on current obs. */

Q^{probe} = PickProbes($PerServerQtoAF$, B_{probe}) // Run additional probes per server

for $s \in S$ **do**

  $ProbeQToAF_s$ = SendQuery(Q^{probe})

  $PerServerQToAF[s]$.insert($ProbeQToAF_s$)

/* Step 3: Local search for each server */

for $s \in S$ **do**

  Q_s^{start} = FindTopKQueries($PerServerQToAF[s]$,K)

  AF^{thresh} = ComputeThresh($PerServerQToAF[s]$)

  LocalSearchUpdateMap($PerServerQToAF[s]$, Q_s^{start}, AF^{thresh})

return *PerServerQToAF*

following two conditions: 1) all queries in the query range induce high AF. That is, $\forall\ q \in \mathrm{QP}$, $AF(q) \geqslant \delta$; 2) the specified range of each field in QP is a maximal in terms of inducing high AF. That is, $\forall\ i = 1, .., n$, $v_i'^l$ and $v_i'^r$, if $v_i'^l < v_i^l \leqslant v_i^r \leqslant v_i'^r$ or $v_i'^l \leqslant v_i^l \leqslant v_i^r < v_i'^r$, then $\exists$ a query $q \in \langle f_1 : [v_1^l, v_1^r], \ldots, f_i : [v_i'^l, v_i'^r], \ldots, f_n : [v_n^l, v_n^r] \rangle$ such that $AF(q) < \delta$.

Given a protocol, $Proto$, we assume that the set of all high query patterns of $Proto$ is unique. We denote the set of all amplification-inducing query patterns as $\mathbb{P}_{Proto}$.

Given a $proto$ and a total budget, Q, the covered high query pattern by Q, denoted $co(Q)$, is the set of high query patterns of $proto$ where each high query pattern shares at least one query

with Q. That is, $co(Q) = \{\text{QP} \in \mathbb{P}_{proto} | Q \cap \text{QP} \neq \varnothing\}$. Based on this definition, we can now formally state the goal of AmpMap. Given a server s running protocol $proto$, AmpMap seeks to maximize the size of $co(Q)$.

5.3.3.1 Analysis of Strawman Approaches

Here, we analyze the expected budget for different strategies for the one-server case.

Exhaustive Search: An exhaustive search enumerates valid queries of the protocol. While this can discover all patterns, the budget is prohibitively large: $E(B) = \prod_{i=1}^{N} |AV(f_i)|$.

Random Search: For pure random search, the expected number of queries to cover all high query patterns is: $E(B) = \int_0^\infty (1 - \prod_{i=1}^{|\mathbb{P}|}(1 - e^{-p_i t}))dt$. Here, p_i is the probability of picking a query in the i-th high query pattern [103]. The expected budget increases exponentially as $|\mathbb{P}|$ increases.

5.3.3.2 Analysis of AmpMap Approach

Under some simplifying assumptions we can analyze the expected budget to discover all patterns. To make analysis easier to present, we make three simplifying assumptions: (1) Each field, f_i, is of homogeneous size F; (2) Each distinct pattern just has one query; and 3. We know the number of distinct patterns, NumPatterns.

In reality, our goal is to discover as many as possible. At a high-level, we can show that our worst-case run time is linear in the $\text{NumPatterns} \times F$.

First, note that given d, the density of queries that give high AF, the expected budget to find one query in one of the patterns is $\frac{1}{d}$. Second, note that the number of queries required to sweep the all neighboring queries from a given query is $F \times NumField$.

Given these preliminaries and our assumptions on the "locality" structure, we can consider the best-case and worst-case analysis to discover all patterns. The best-case is when all patterns form a fully connected clique, where two queries in two distinct patterns are neighbors. This

means, that when we start from a query in a q_1, we will discover all other $\mathrm{NumPatterns}$-1 patterns in just one sweep. The worst case is when all 4 distinct patterns ($\mathrm{QP1} \cdots \mathrm{QP4}$) form a chain. That is, we need to do one sweep to discover an additional mode. Note that we are guaranteed to find another pattern (Observation **1**) because all patterns can be reached by sweeping each field. Hence, we need to do $\mathrm{NumPatterns} - 1$ sweep. Since we assume we know what is $\mathrm{NumPatterns}$, our search will terminate when we discover all patterns. Taken together, the best-case run-time is $\frac{1}{d} + F \times NumField$, and the worst-case run-time is $\frac{1}{d} + (\mathrm{NumPatterns} - 1) \times F \times NumField$.

5.4 Evaluation

In this section, we present findings from our Internet measurements for 6 UDP-based protocols (DNS, NTP, SNMP, Memcached, Chargen, SSDP) and local testing for 3 protocols (QOTD, Quake, RPCbind). In contrast to a scoped experiment (Section 5.1.2) which was used to motivate AmpMap, the results we present here are more complete; i.e., we cover more protocols, servers, and search over the packet header space (opposed to sending a fixed set of queries). We also validate our design against strawman solutions and parameter choices.

Measurement setup: We use nodes from Cloudlab [94] where 1 node is used as a controller, and 30 as measurers. (We restricted our node usage to 31 per experiment, as Cloudlab is a shared platform across institutions.) For these 6 protocols, we scanned 10K *sampled* servers for each protocol: DNS with OPT records for EDNS, NTP, SNMP, Memcached, Chargen, SSDP. For DNS, we scan the servers obtained from Censys and hence, these are mostly open recursers. (We can easily extend AmpMap to handle authoritative servers.) As the protocol format for SNMP's Get, GetNext and GetBulk requests differ, we treated each as a separate protocol and ran separately. Similarly, we ran separate runs for NTP's mode 7 (private), mode 6 (control), and mode 0-5 (normal). We obtained public server IPs from Censys [95] and Shodan [30].

We randomly sampled IPs from these lists and pruned out inactive servers (e.g., not respond to *dig* for DNS) or owned by military or government. For certain protocols (SNMP, NTP) that have different modes of operation with distinct formats (e.g., SNMP has GetBulk, GetNext, and Get), we consider two notions of active server, whether the server responds to (1) "any" of the modes (OR filter); or (2) "all" of them (AND filter). We present results for both schemes, using AND/OR superscripts to denote each (e.g., SNMP $^{\text{AND}}$).

	# IPs scanned	# pruned IPs (b)		# IPs taken	# IPs in DB	% IPs scanned
	(a)	invalid proto	gov't or mil.	(c)=(a)+(b)	(d)	(c) / (d)
DNS	10K	18,698	15	28,713	8.02M	0.36
NTP	10K	4317	5	14,322	8.4M	0.17
NTP $^{\text{AND}}$	3,083	234,374	7	237,464	8.4M	0.28
SNMP	10k	4,933	3	14,936	2.16M	0.69
SNMP $^{\text{AND}}$	10K	60,187	9	70,196	2.16M	0.33
Memchd	10K	11,736	9	21,745	63K	3.5
Chargen	10K	68,065	6	78,071	83K	9.4
SSDP	10K	78,617	3	88,620	2.16M	3.3

Table 5.4: Statistics on (a) the # of IPs we scanned per protocol, (b) the # of pruned IPs, (c) the # of raw IPs we needed from the DB ; (d) the # of total public-facing IPs as is (Shodan and Censys); and (e) the % of IPs we scanned

To finish our measurements in a few days and restrict the number of (shared) nodes we use, we target 10K servers per protocol. Table 5.4 shows: (1) the number of IPs we needed from Shodan/Censys to get our final server lists, (2) the total number of public-facing IPs for each protocol (as of May 30, 2020) from Censys (DNS) and Shodan (others); and (3) the % of IPs we scanned. When we refer to "servers" to present our results, we are referring to "sampled" servers rather than the entire Internet servers.

In our experiments, each server is pinned to a measurer. We do not spoof IP addresses and we send legitimate queries and listen to responses. We impose a limit of 1 query per 5 s for each

server with a timeout of 2 seconds (i.e., 7 seconds per query). This gives approximately 3 days to complete for 10K servers as 30 measurers can handle 500 servers at a given time (each run takes 3 hours, 7s$\times$1500 queries, and need 69 hours to handle 10K servers (not accounting for timeouts). Our load is low: 48 kbps (egress) across all measurers and 1.6 kbps per measurer. If we assume an average AF of 5, then we incur 240 kbps in ingress bandwidth.

Protocol specifics: For protocols with more than 10 fields (DNS,NTP, RPCbind), we used a query budget of 1500 queries per server, setting 45% for RandomSample Stage and 10% for the Probing stage. For simpler protocols, we used a budget of 400 queries with the same split. For QOTD, Quake, RPCbind, we set up a single Cloudlab server running the protocol. Some fields such as domain fields for DNS took strings. As discussed in Section 5.3.1.2, we picked 10 popular domains[2] spanning different continents, industry sectors, and enabled features. For SNMP, we pick v2's OIDs based on the RFC up to depth 4 (i.e., A.B.C.D). For certain fields that take as input a list of values (character for Chargen, OID for SNMP), we also search over the length of the list.

5.4.1 Protocol and Server Diversity

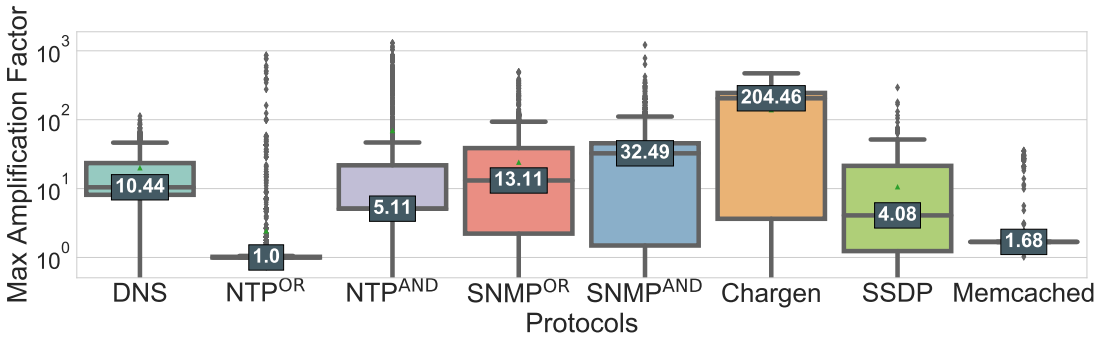


Figure 5.8: Boxplot showing the distribution of the maximum AF achieved by each server given a protocol

[2]berkeley.edu, energy.gov, chase.com, aetna.com, google.com, Nairaland.com, Alibaba.com, Cambridge.org, Alarabiya.net, Bnamericas.com

Finding 1: ***There is significant variability in the maximum amplification a server can yield across servers.***

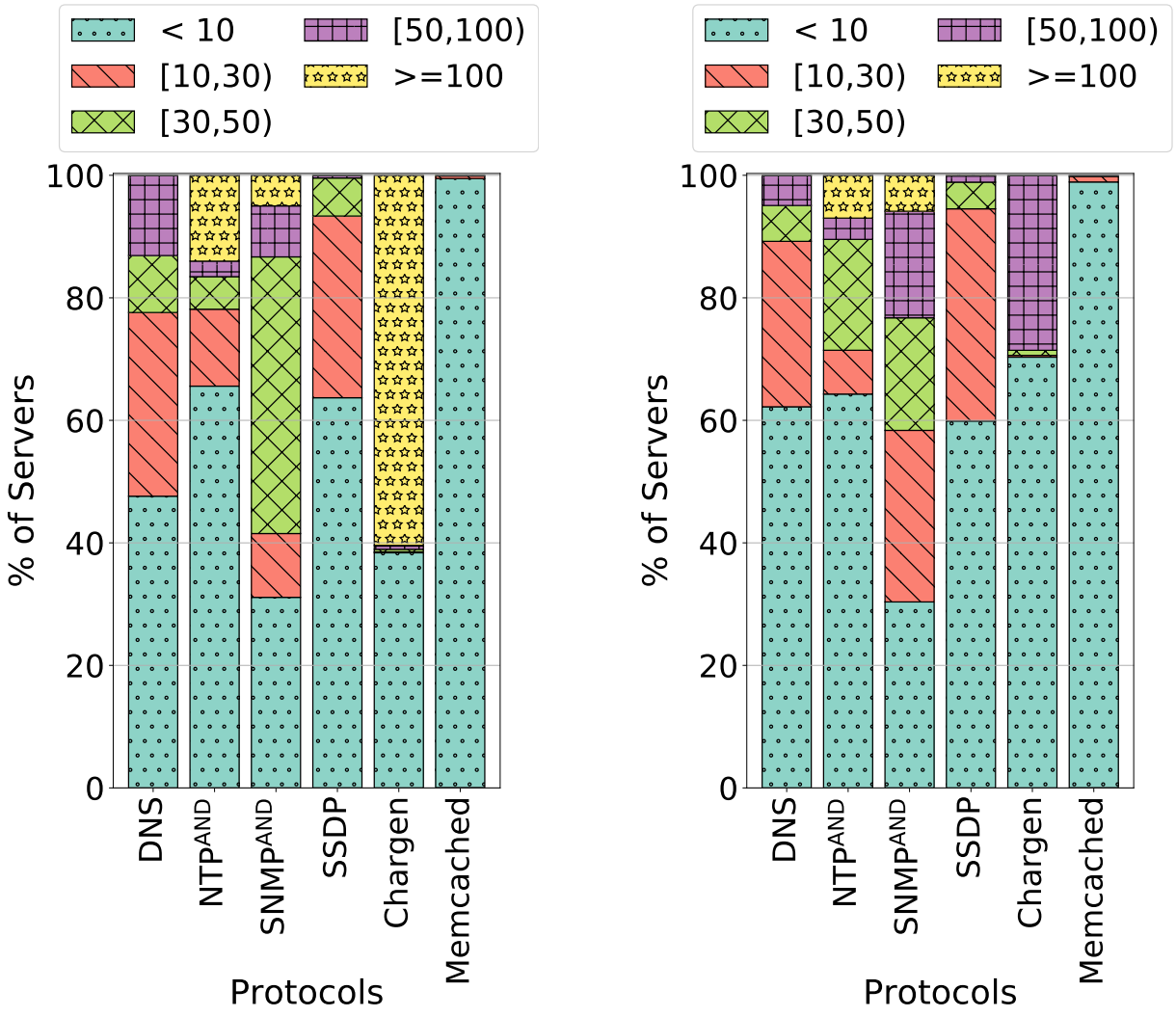
Figure 5.8 (y-axis in log-scale) shows the distribution of the the maximum AF achieved by each server for each protocol. For many protocols, we observe a long tail in the distribution. For instance, while the median for SNMP $^{\text{OR}}$ is 13.01, the maximum is 495. While the median is 1 for NTP $^{\text{OR}}$, the maximum is 860. Similarly, for NTP $^{\text{AND}}$, while the median is 5.11 the maximum is as large as 1300! This confirms we cannot simply count the number of open servers or attribute the same risk to each server.

Finding 2: ***There is substantial variability in the maximum*** AF ***distribution across protocols.***

Figure 5.9a shows the maximum AF distributions with varying AF ranges (e.g., 10-30) across protocols; these experiments ran in May - June 2020. For SNMP and NTP, we only show the results for AND schemes for brevity. First, protocols vary in the percentage of potential amplifiers with $\text{AF} > 10$: 52% for DNS, 34% for NTP $^{\text{AND}}$, 69% for SNMP $^{\text{AND}} \cdots$ 0.6% for Memcached. Further, protocols differ in the most common AF ranges ($\geqslant 10$) that servers can yield. AF range for DNS is concentrated on 10 to 30 but above 100 for Chargen. For NTP $^{\text{AND}}$, 14% of servers give above 100 AF. These results suggest that measuring the risk should take into account the AF distribution per protocol.

Finding 3: ***There is variability across time in the*** AF ***distribution across servers for different protocols.***

Figure 5.9b shows maximum AF distribution from measurements done in `2019` (opposed to `2020` for Figure 5.9a). These visually highlights the differences across two years. For instance, only 7% of NTP $^{\text{AND}}$ servers yielded $\text{AF} \geqslant 100$ in 2019 vs. 14% in 2020. 90th percentile of DNS servers induced above 30AF in 2019 but 59 AF for 2020 (almost doubled) using the identical



(a) May-June 2020

(b) May-June 2019

Figure 5.9: Summary across servers and protocols (from 2019 and 2020 runs)

domain lists.[3] We acknowledge that as we sample servers, we cannot attribute the root cause of differences (i.e., change in server list vs. change in the actual attack landscape). However, such variability is the reason that calls for the need to do continuous (periodic) measurements rather than a one-time analysis.

[3]Across these two experiments, there are minor differences in the parameters (e.g., 53% random in 2019 vs. 45% in 2020, searching over 0-32 as a list size in 2019 vs. 0-256 in 2020) but they do not really affect the results.

5.4.2 Assessing Amplification Risks

	Known Pattern	Risk Quantification		Result
		Prior Work	AmpMap	
DNS	EDNS:0, ANY [3, 165]	287K	149K	1.9× ↑
	EDNS:0, ANY [3],TXT [28]	Not Known	183K	N/A
DNS (domains w/o DNSSEC)	ANY, TXT [176]	Not Known	126K	N/A
NTP $^{\text{OR}}$	MONLIST [4, 165]	5,569K	13K	427× ↑
NTP $^{\text{AND}}$	MONLIST [4, 165]	5,569K	635K	8.8× ↑
SNMP $^{\text{OR}}$	GetBulk [5, 165]	64K	223K	3.5× ↓
SNMP $^{\text{AND}}$	GetBulk [5, 165]	64K	317K	5× ↓
Chargen	Request	3588K	1399K	2.9× ↑
SSDP	Search [5, 165]	308K	126K	2.7× ↑
Memcached	Stats [5, 25]	100M [5]	18K	5.6K × ↑

Table 5.5: Contrasting the risk extrapolated from prior works and measured by AmpMap for 10K servers

---

Finding 4: ***Even for known patterns, extrapolations (e.g.,[85, 165]) mis-estimate amplification risk.***

Table 5.5 summarizes the known patterns and their corresponding risks assessed using AmpMap and prior works [3, 165] (same risk used in Section 5.1.2). For AmpMap, given a pattern for each protocol (e.g., MONLIST for NTP), we calculate the total risk across 10K servers using Eq. 5.1. We find that the baseline techniques from prior work has significant mis-estimation; i.e., NTP is 427× overestimated, SNMP v2 is 3.5× underestimated, and Chargen is 2.9× overestimated. The large inaccuracy for NTP is because the previously reported AF of 556 [165] does not generalize to majority of NTP servers. Our findings confirm a follow-up study of NTP amplification [85],

which specifically focuses on the MONLIST feature. Similarly, the reported average of the worst 10% servers for GetBulk requests (SNMP) is 11.3 [165]. However, the average of the worst 10% is 90 (7.9×) for SNMP $^{\text{OR}}$ and 97 (8.6×) for SNMP $^{\text{AND}}$. This is because the prior analysis does not account for polymorphic variants (i.e, tweaking the OID and the number of OIDs to request for).

	New Patterns	**Risk Quantification Patterns**
DNS	EDNS≠ 0 or ≠ ANY	3274K (21.9 × known pattern)
	EDNS≠ 0 or ≠(ANY or TXT)	3127K (17.1 × known pattern)
NTP $^{\text{OR}}$	reqcode ≠ MONLIST (20,42)	43K (3.3 × known pattern)
NTP $^{\text{AND}}$	reqcode ≠ MONLIST (20,42)	663K (1 × known pattern)
SNMP $^{\text{OR}}$	GetNext	61K (0.27 × known pattern)
	Get	10K (0.04 × known pattern)
SNMP $^{\text{AND}}$	GetNext	101K (0.32 × known pattern)
	Get	11K (0.03 × known pattern)
SSDP	None	0
Memcached	Get, Gets	33K (1.9 × known pattern)

Table 5.6: Amplification risk from new patterns whose risks will be missed by prior works

---

Finding 5: ***Prior recommendations (e.g., [85, 165]) miss many query patterns and leave substantial residual risk.***

We now quantify the risks from new patterns that will be "missed" by prior analysis (Table 5.6). For DNS, there are other combinations of edns and recordtype fields that yield considerable amplification. The total risk from these other patterns (e.g., recordtype:LOC, URI) across 10K servers is 3,274K, which is 21.9× larger than the risk of known patterns (149K)! Figure 5.10 shows a bird's-eye view of the residual risk. We observe similar trends for other protocols; e.g., for NTP, a collective risk from other features (e.g., "get restrict") is 276× higher risk than the known risk. For simpler protocols like SSDP, we do not find new patterns.

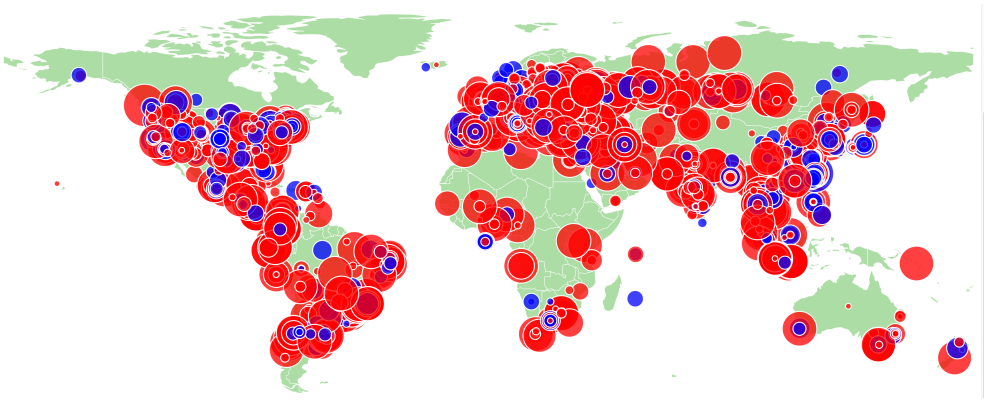


Figure 5.10: Visualizing the DNS residual risk when known patterns (P): edns:0,recordtype:ANY—TXT, are blocked. The size of the circle $\propto$ the max AF of each server and red circles denote when the delta is $\geqslant$ 20%

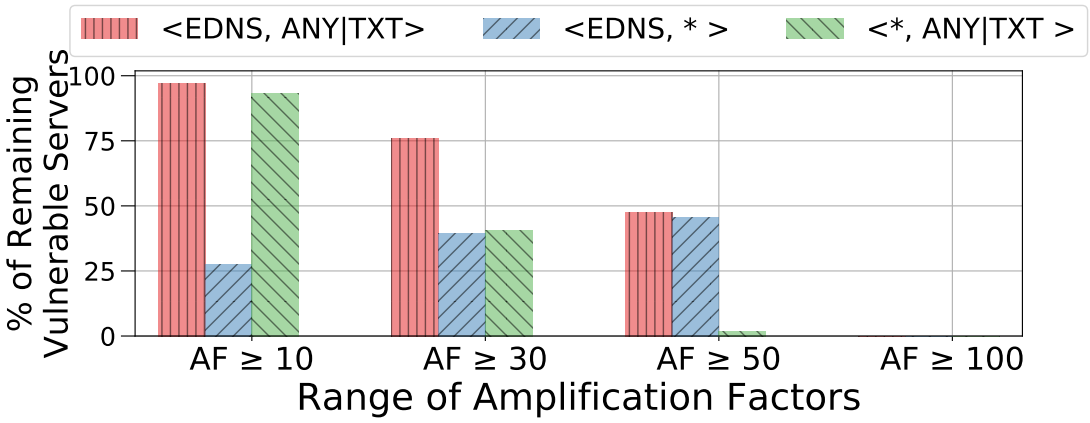


Figure 5.11: % of DNS servers that remain susceptible to amplification even if we use recommendations by prior works to block query patterns; i.e., $\langle$ EDNS, ANY—TXT $\rangle$ is a filter that blocks queries EDNS:0 and ANY|TXT

Next, we conduct *what-if* analysis to analyze what percentage of servers are susceptible to amplification if we were to block known patterns. Given that prior works do not provide concrete "signatures", we consider a few possible interpretations; i.e., a combination of edns:0 and recordtype:ANY, TXT. Figure 5.11 shows that even with edns0 and (ANY or TXT) blocked, more than 97% of servers still can yield AF greater than 10. For NTP (mode 7), even with

MONLIST as a signature[4], 30.5% servers can still yield AF $\geqslant$ 10 and 4.8% $\geqslant$ 100! We observe similar trends for SNMP. However, prior recommendations achieve high coverage for SSDP, Chargen, and Memcached.

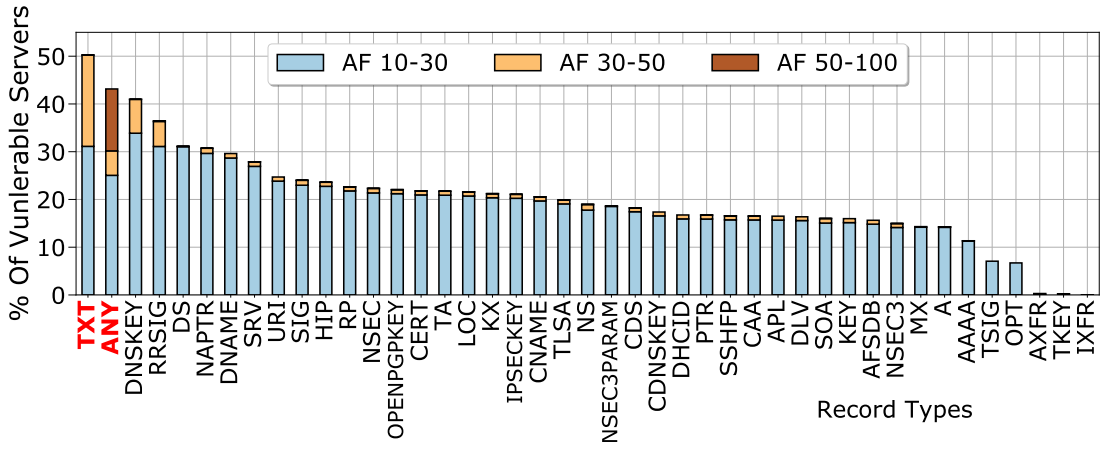


Figure 5.12: The variability of field values (for a specific field, recordtype**) that contribute to high amplification. Apart from known ones (**recordtype**:ANY, TXT), many other** recordtype **values can lead to large** AF

5.4.3 In-Depth Analysis on DNS

The previous discussion suggests there are many patterns not highlighted by prior work. We analyze this further next. We focus on DNS first and defer other protocols to Section 5.4.4-Section 5.4.6.

We start with a recordtype field as this field determines ANY vs. NS records. Figure 5.12 shows the percentage of servers that can induce considerable AF for each possible value of this field. While the top-2 record types are TXT and ANY (pointed by prior work), more than 20% of our sampled servers can yield more than 10 AF with 19 other recordtype values (e.g., URI, HIP, RP, LOC, CNAME). Some of these (e.g., NAPTR) incur very high AF especially if used

[4]A follow-up paper mentioned the possibility of other settings that induce amplification, they did not specify which request types [85].

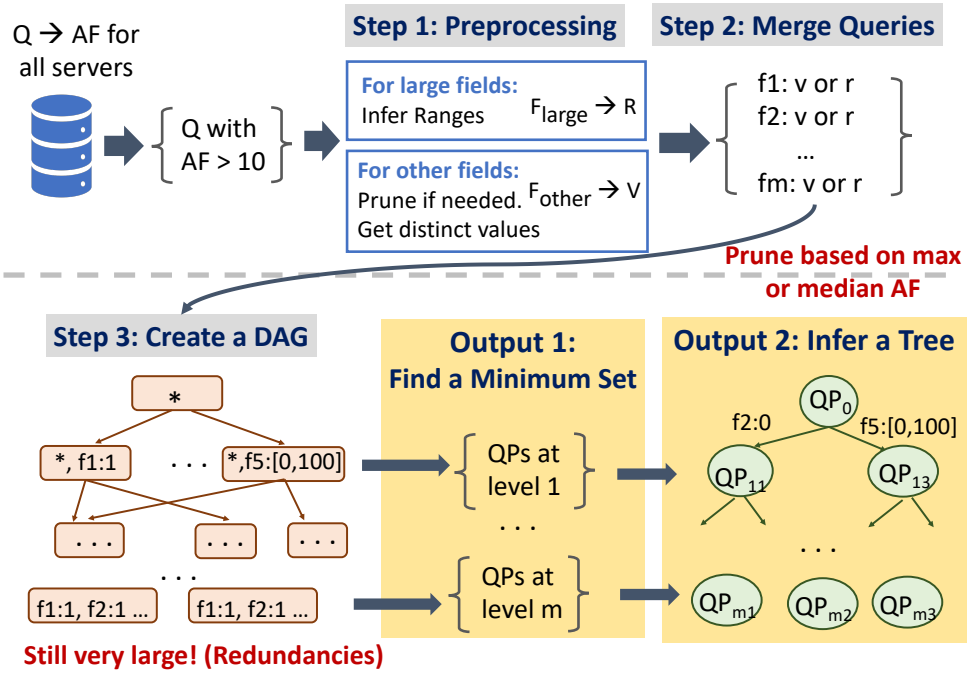


Figure 5.13: Steps to obtain query patterns to shed light on the patterns of amplification

in conjunction with the dnssec (DNSSEC-OK) set. While many DNSSEC-related recordtype values (e.g., RRSIG, DNSKEY, DS) can yield high AF [175], we also observe many recordtype values "unrelated" to DNSSEC (e.g., NAPTR, SRV). This is significant—even if we block ANY, TXT queries, there are "many other" types that can induce high amplification.

Summarizing and analyzing query patterns: The above analysis only considers one field. In practice, many other combinations of fields are susceptible, and we want to understand the structure of amplification-inducing query patterns (QPs). For this summarization, we considered a number of standard data mining techniques (i.e., hierarchical clustering, K-means clustering, decision trees) but found none were suitable. (For instance, clustering assume that we know the number of clusters or the right distance metric/threshold. Similarly, given the large combinatorial space, decision trees produce uninterpretable outputs.) Given these limitations of standard techniques, we designed a custom heuristic (Figure 5.13). Starting from AF-inducing queries across all servers, we generate a set of candidate patterns where some fields are set to concrete values or ranges, and others are wildcarded. Specifically, for large fields (e.g., id, payload for DNS)

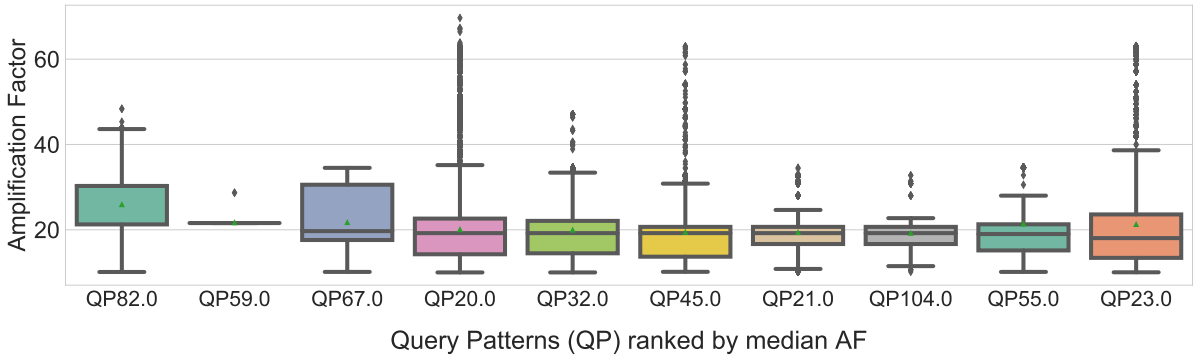
we identify candidate *ranges* by dividing the accepted values for a large field into exponentially-spaced bins (e.g., $\{[0, 10], [11, 100]\ ...\}$. Then, for *each server*, we generate a bit vector (e.g., 1111) to represent these bins; it is set to 1 if a server has a query ($\mathrm{AF} \geqslant 10$) using a field value that belongs to the bin range. Finally, given a set of bit vectors for all servers, we take candidate vectors that are observed across $\geqslant 10\%$ of servers. We prune out fields that appear to have no effect on amplification; we count the number of queries (with $\mathrm{AF} \geqslant 10$) by checking if wild-carding the field makes the AF value histogram follow a uniform distribution. We then generate candidate patterns by generating all combinations of values and ranges. From these candidates, we prune out QPs with $\mathrm{AF} < 10$ based on the "maximum" or the "median" AF. We represent the QPs as a logical `Directed Acyclic Graph` (DAG), with these QPs are leaf nodes (Step 3, Figure 5.13). We create a parent node by taking one of the nodes in the current $level$ and wild-carding one field; the root of the DAG is a node where all fields are wildcards. Given this DAG, we consider two analysis:

1. *Minimum set cover per level (Output 1, Figure 5.13)*: We compute the minimum set-cover of QPs at each level that logically covers all leaf nodes; i.e., the set of QPs obtained at level 10 represents the *minimum set of* QPs to describe QPs using only 10 fields as concrete values or ranges.
2. *Hierarchical analysis (Output 2, Figure 5.13)*: To see dependencies across fields, we create *a tree* where the edge is annotated with the field and its value which became concrete as we increase the level (an example in Figure 5.15).

We run the above procedure separately for: 1) domains with DNSSEC support, and 2) domains without support.

Corollary 1: ***Many unexpected patterns lead to high*** AF***; e.g., with*** dnssec ***off and unrelated to ANY records.***

DNSSEC-related patterns: Figure 5.14a shows a boxplot of top 10 QPs w.r.t. the median AF



(a) Rank based on median AF

ID	Field values
QP 82	⟨edns:0, payload:*,recordtype:RRSIG, ad:1, rd:*, rcode:8 · · · ⟩
QP 20	⟨edns:1, payload:*, recordtype:*, ad:0, rd:1, rcode:* · · · ⟩
QP 32	⟨edns:1, payload:*, recordtype:TXT, ad:0, rd:1, rcode:* · · · ⟩

(b) Describing query patterns (QPs)

Figure 5.14: DNS: Top 10 query patterns for a particular depth where 8 fields are left as concrete values of ranges

when 8 fields are left concrete (level 8) in Figure 5.14. QP 82 incurs the largest median AF of 30 with ⟨edns:0, payload:*, record type:RRSIG, rd:* · · · ⟩; here, it is not necessary to have rd set to 1 and shows that having recordtype RRSIG can also cause high AF. The rank-2 QP has edns set to 1 and not 0 (a known pattern); several servers that yield high AF had edns not set to 0. Further, as we find many recordtype values that lead to high AF (also seen in Figure 5.12), this QP has recordtype set to *. Further, as a side note, when we were pruning out fields that appear to have no effect on AF (Figure 5.13), dnssec (DNSSEC-OK) got pruned out; however, we have observed that setting this bit to 1 on certain queries can induce high AF on a subset of servers.

Non-DNSSEC patterns: For certain servers, domains without DNSSEC support can also yield high AFs. Specifically, the median AF for the top-1 QP is 21 with ⟨edns:1, record type:TXT, rd:1 · · · ⟩. This confirms that TXT records can cause high AF [176]. We also observe recordtype values such as DS appear among the QPs; some are attributed to anomalous servers.

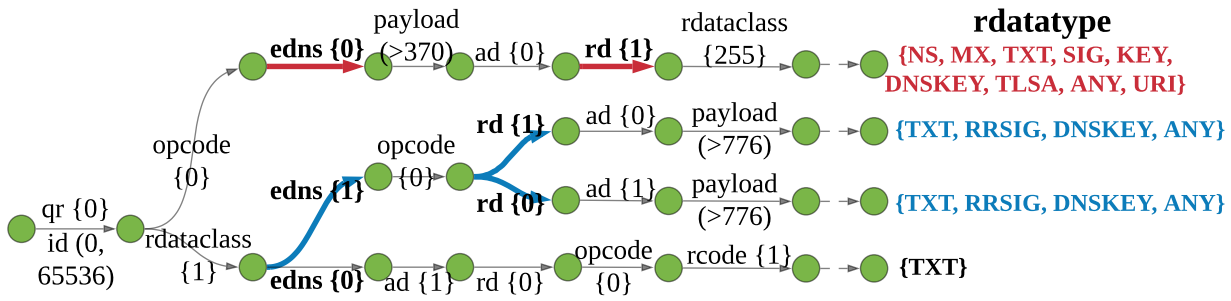


Figure 5.15: Tree showing how the query patterns change across levels. An edge means a field value transitioned from a wildcard (*) in level L to a concrete value or range in the next level, $L + 1$

Corollary 2: ***There are many query patterns that while not "max" provide high enough amplification. This can render an existing mitigation (i.e., [119]) ineffective.***

At each level of DAG, more QPs are concentrated at AF between 10 and 20. At the leaf nodes, 699 QPs produce a median AF of 10 to 20 while only 47 above 20 AF. Purely focusing on one pattern or a handful to drive the mitigation plan will be insufficient.

Corollary 3: ***There are complex dependencies across field values inducing high*** AF ***change based on other fields.***

The DAG output (Figure 5.13) shows complex dependencies across field values that yield high AF. Specifically, Figure 5.15 shows a subset of a tree (for DNSSEC-related) where the QPs are filtered based on the "median" AF. If we consider a top branch with edns:0 and rd:1, with NS, MX, $\cdots$ TLSA, URI record types cause high AF. Some other combinations (i.e., blue edges) will cause different recordtype values to induce high AF. Surprisingly, we find a non-trivial number of servers that yield high AF even when rd (recursion desired) to 0 (off)! These suggest that (1) there are many combinations of *multiple* fields values that lead to high AF, and (2) this finding generalizes to many servers (as QPs are kept if the "median" AF across servers is

$\geqslant$ 10 AF). Further, if we consider a tree where QPs are pruned based on the maximum AF (less aggressive pruning), we see even more combinations leading to high AF (e.g., OPENPGPKEY, SOA for recordtype).

Further, we observe that not all servers behave according to specifications further adding to variability in QPs. For instance, when edns is used, the response should be chopped to the specified payload value. Unfortunately, for many servers, this is not the case; i.e., 88 servers (out of 10K) yield AF above 50 with payload $<$ 512. Also, in our 2019 run, we saw 311 AF for one server (for SRV record) where we saw many IP fragments. This server went offline shortly after the experiment. While DNS over UDP does use IP fragmentation to deliver large payloads [22], this makes defenses more difficult as they miss key fields such as port information [7].

Corollary 4: ***Given the variability of query patterns, blocking Top-K percentage of patterns still leave significant residual risk; i.e., the 50th percentile of servers have 80% or more residual risk even with blocking 20% of query patterns (infeasible in practice).***

We now analyze the percentage (%) of the residual risk if we had used the top-K percentage (%) of QPs to block these queries. For this analysis, from the inferred QPs (Figure 5.13), we do not prune them based on the maximum or median AF; we need to know all QPs that lead to high AF for each individual server. We take the top-5 and 20% of these 11K QPs (sorted by their median AF) and use them to block amplification-inducing queries from each server. Unfortunately, we observe that even blocking the top-20% QPs (which is infeasible in practice) still leaves 50% of the servers with 80% risk or higher (96.7% or higher risk if we block top-5 %).

Corollary 5: ***Many DNS vendors are affected.***

Table 5.7 shows the affected vendors with servers that can yield AF $\geqslant$ 10. We only show vendors with more than 20 servers. We discuss our efforts to notify these vendors of the vulnerability in Section 5.5.2.

Vendor	# of total servers	# server (AF $\geqslant$ 10)	% of servers (AF $\geqslant$ 10)
Bind	946	236	24.9%
Dnsmasq	917	819	89.3%
Version:recursive-main/*	522	12	2.3%
Microsoft	261	250	95.8%
PowerDNS	78	50	64.1%
unbound	40	26	65%

Table 5.7: Statistics on the affected DNS vendors

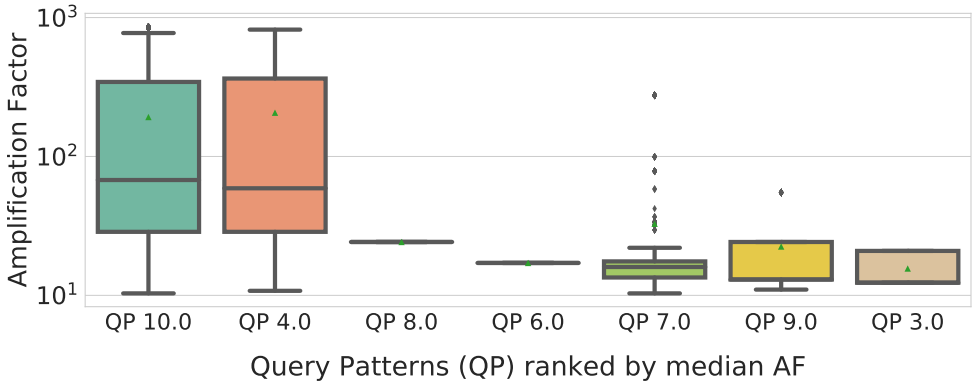


Figure 5.16: NTP top query patterns where the top-2 are MONLIST **patterns. Other top** QPs **have peer list, if reload, peer list sum, and peer stats as** reqcode**.**

5.4.4 Amplification Patterns for NTP

We discuss amplification patterns for NTP; as we do not discover new patterns for mode 0-6, we focus on mode-7 (private mode). Recall that we need to prune candidates QPs based on maximum or median AF (Figure 5.13). As we observe a high variance across AF achieved by different NTP servers, we looked at the QPs where they are pruned based on the maximum. Figure 5.16 shows all QPs where they ranked by the median AF. Apart from MONLIST (QPs 10 and 4), we observe reqcode of `peer list` , `if reload` , `peer list sum`, and `peer stats` from NTP $^{\text{OR}}$; some of these other QPs can yield as large as a few hundred (seen by the

long "tail" in Figure 5.16). From NTP $^{\text{AND}}$ servers, we also observe `mem stats`, `if stats`, and `get restrict`. Our findings for NTPprivate again complements **Corollary 2**. The affected versions (with servers that can yield $\geqslant$ 10AF) are 4.1.1-2, 4,2,4, 4,2,6-8, and 4.2.0. Further, servers that can induce high AF with other request codes (other than MONLIST) are not particularly tired to one single version but span across multiple versions.

Vendors	**# total servers**	**GetBulk**		**GetNext**	
		# server (AF$\geqslant$10)	% servers (AF$\geqslant$10)	# server (AF$\geqslant$10)	% servers (AF$\geqslant$10)
net-snmp	5357	5044	94.2%	3445	64.3%
cisco Systems	594	96	16.2%	60	10.1%
SonicWall	220	21.7	98.6%	27	12.3%
Broadcom Corp.	205	193	94.1%	81	39.5%

Table 5.8: Statistics on the affected SNMP vendors

5.4.5 Amplification Patterns for SNMP

We now discuss patterns for SNMP, which have 3 modes of operations; i.e., GetBulk, GetNext, and Get. We start with `GetBulk`, which is a known pattern [5] (reported average of 6.3 AF [165]). However, in running AmpMap, we discovered *polymorphic variants* that lead to significantly "higher" AF; i.e. we saw an average of 22.4 AF for SNMP $^{\text{OR}}$ and 31.8 AF for SNMP $^{\text{AND}}$. Specifically, an attacker can modify OID value and the number of OIDs to yield higher AF. Generally, we generally observe higher AF for queries with (1) a single-digit OID (near the root) such as 2, 1, 0, and (2) a list containing multiple OID (i.e., 2-15 but above 15). However, given server variability, there are exceptions; e.g., OID of 1.3.6.1.2, and a list size of 1 appears in one of the top-4 patterns. The top-1 QP from the SNMP servers yields a median AF of 35 with $\langle$ community:public $\cdots$ OID:2, numoid: (0,8) $\rangle$. From SNMP $^{\text{AND}}$ servers, the top-1 QP yields 45 median AF with OID:0. We now discuss `GetNext` requests. While only GetBulk has

been highlighted in the prior analysis, AmpMap discovers that a single GetNext request can also yield hundreds of AF (similarly, by varying the OID and the number of OIDs to query). From SNMP $^{\text{AND}}$ servers, 37% servers can yield AF above 10 and 0.74% above 100AF! From SNMP $^{\text{OR}}$ servers, 10% servers yield above 10 AF and 0.14% above 100. However, unlike SNMPbulk, we saw high AFs for various OIDs (e.g., 1.3.6.1.2, 0, 1, 1.3.6.1.3); this is expected because GetNext just requests the "next" variable in the tree, unlike GetBulk which requests a number of GetNext requests. While we also replicated that a local server can yield 15 AF with GetNext (by varying the list size), we posit that we see higher AF in the wild given server variability. Table 5.8 shows the affected vendors for servers that can yield greater than 10AF using GetBulk or GetNext requests; we only show for vendors with more than 200 servers and this combines the results from both SNMP $^{\text{AND}}$ and SNMP $^{\text{OR}}$. Similar to DNS and NTP, this amplification vulnerability affects multiple vendors and not just one.

Lastly, measurements reveal that `Get` requests also can yield tens of AF (but not as large as GetNext). From SNMP $^{\text{OR}}$, 0.73% servers that have AF greater than 10. Unlike GetNext, we observe high AF for OID of 1.3, and 1.3.6.1.3-4.

5.4.6 Amplification Patterns for Other Protocols

SSDP: Amplification risk is inherent with SSDP's "discovery" feature. Our inferred QPs are quite simple. If the leaf nodes are pruned based on the median AF, we see a discovery request with one UUID of "ssdp:all." This is expected as this feature will fetch "all" UUID information. However, for QPs based on the maximum AF, we see many UUIDs leading to $\geqslant$ 10 AF. Again, this confirms the presence of multiple query patterns.

Memcached: We did not find any QPs that lead to above 10 AF other than the "stats" request (a known pattern) from our 2020 run. If we use our runs from 2019, some of the QPs with *get* and *gets* requests did induce above 10 AF. However, it is still the case that "stats" are by far the dominant pattern, and the residual risk from *get* and *gets* requests are negligible. Further, while

the known AF for Memcached is tens of thousands [33], the maximum we find from our 2020 run is 35 AF (we believe many have been patched or taken offline).

Chargen: As Chargen servers respond to any UDP datagram, the QPs learned at the leaf nodes contain all possible characters and lengths. We represented the search space as a list of hex strings where we search over the hex character as well as the length of the hex character.

We validate the existence of amplification-inducing query patterns for three protocols in a lab setting. For these, we confirm the known patterns but do not find additional ones.

Quake: "Get status" message induces AF of 10 in our setting.

QOTD: As this server responds with random quotes, we see higher AF when the list size is smaller and the size of the quote is larger.

RPCbind: The request for the process number running on the server with a correct version ID incurs high AF (i.e., 10).

5.4.7 Parameters and Validation

Given the lack of ground-truth for all servers, we use a combination of local-server experiments, a large-scale simulation, and example measurements for validation. In the local experiment, we randomly sample 2M queries on a local DNS server and measured the AFs to infer the signatures (Section 5.4.3). Our simulator models an amplification function that maps a query to AF based on (1) field types, (2) the # of servers, (3) the # of pattern structures across servers, (4) the # of pattern for each (3). For (3), indicating 100 pattern types instantiates 100 graph structures across servers where each gets mapped to one type. This simulates the variability across servers.

Validating parameters: There are three key parameters: (1) per-server total budget, B, (2) allotting B across different stages (e.g., Probing stage), and (3) the number of clusters for K-means.

To see the impact of total budget (B), we use the local DNS server experiment. Fixing other parameters (50% for B_{rand}), we varied the B from 100 to 2000 (Figure 5.17). To show the

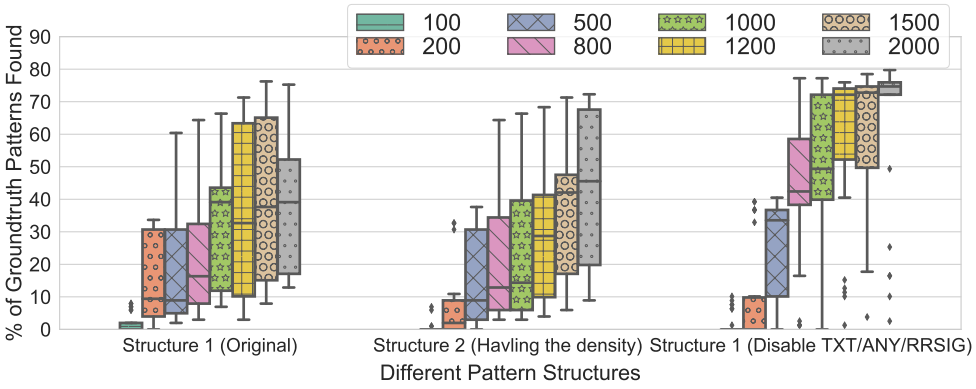
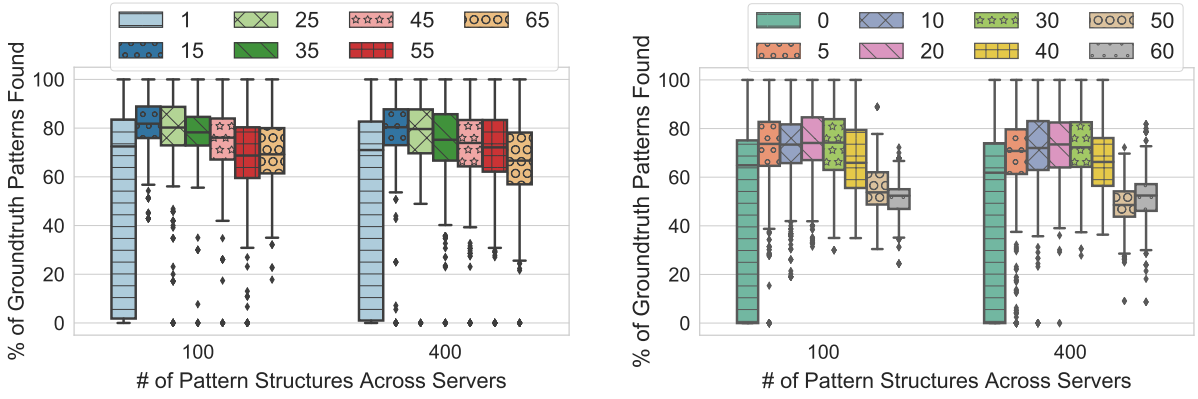


Figure 5.17: Validating the choice of total budget (B)

robustness across multiple pattern structures, we "emulated" different pattern structures given one setup. We emulated the effect of (1) reducing the % of AF-inducing queries by half, and (2) disabling certain patterns (TXT, RRSIG, ANY). Clearly, using only a few hundreds achieves low coverage but starts seeing the "diminishing" return at 1200 or 1500. We chose 1500 for complex protocols (e.g., DNS). This confirms our chosen B is in a sufficiently good "operating region."

To see the impact of the budget across stages, we use our simulator with 1K servers where 30% servers are configured to not induce high amplification (similar to the real-world). To analyze the robustness w.r.t. different levels of diversity, we test against 100 to 400 pattern structures. First, using 50% for B_{rand}, we vary the % of B_{probe} from 0 to 40% (Figure 5.18b). Using 0% for probing hurts coverage but using 5% and 30% is robust across settings. We chose 10% (lower end of the range) as we should spare the budget for other (more critical) stages. Similarly, we vary the B_{random} from 0 to 70% (Figure 5.18a). We observe robustness across 5% to 45%. As it is crucial for this stage to discover at least one AF-inducing query (for most servers), we chose 45% (the higher end). This leaves per-field search with the remaining 45%.

To validate the number of clusters, we use the same simulator and evaluate based on the percentage (%) of servers, which the chosen B_{probe} discovered ($\geqslant$ one) high AF query. Then, we vary the number of clusters from 2 to 200 and observe robustness across these values; i.e., this is not a "crucial" factor.



(a) % of budget for B_{random}

(b) % of budget for B_{probe}

Figure 5.18: Validating the choice of budget allocation

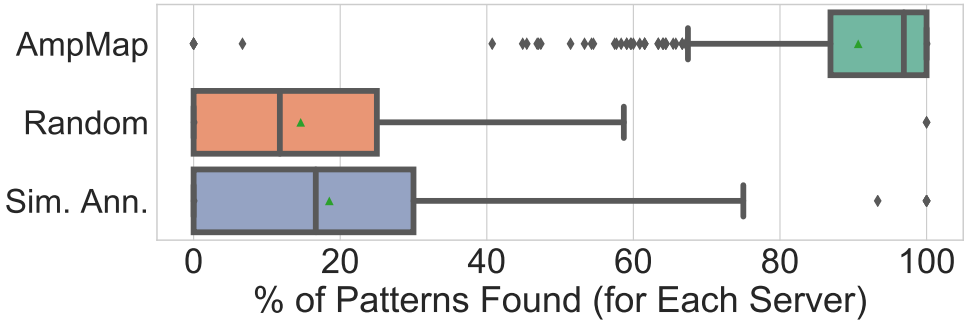


Figure 5.19: Validation of coverage of AmpMap and alternate solutions using 1K server measurements

Comparison with structure-free alternatives: We compare AmpMap vs. two baselines: 1) Simulated Annealing (SA), and 2) pure random search. Specifically, these baselines are structure-free approaches. Our success metric is pattern coverage across a set of servers. We compared these solutions using a small-scale 1K measurements. As we lack the ground-truth for each server, we compare the "relative" performance across these solutions rather than to claim optimality or completeness. Using a budget of 1500 queries, we inferred the signatures combining the output across all solutions. Then, we analyze the coverage for each server. For a given server, we take all queries with $\text{AF} \geqslant 10$ across three solutions, which serves as the basis of comparison for this server. Then, for each strategy, we compute the % of patterns discovered for each server.

Figure 5.19 shows the coverage across 1K servers for each solution. While SA performs better than pure random strategy, the median coverage is 16.7% while the pure random strategy has 11.9% median. AmpMap achieves 97% coverage in this relative comparison.

5.5 Precautions and Disclosure

We carefully considered the impact of our measurements and the disclosure of our findings. We followed the ethical principles (Menlo Report [65]) and the scanning guidelines suggested by prior efforts (Zmap [96]). At a high-level, we adhered to these principles of (1) *minimizing the harm* by taking multiple measurement precautions (Section 5.5.1), and (2) *being transparent in our method and results* by informing relevant stakeholders of our findings and explaining the purpose of our scanning (e.g., when we send out email notifications) (Section 5.5.2).

5.5.1 Scanning Precautions

We took precautions to ensure that there was no harm to the amplifying servers and the network. Our study was approved by IRB under non-human subject criteria. We took care to ensure that our measurements do not burden servers or the Internet.

- We send at most one query per 5 seconds, do not send malformed requests, and cap overall budget per server.
- We do not scan the IPv4 network space but only known public servers obtained from Censys [95] and Shodan [30].
- We do not spoof the source IPs to induce responses to others. Our measurers explicitly receive the responses.

Abuse complaints: We closely worked with the Cloudlab [94] administrators whom we notified of our measurements and the purpose of AmpMap. We only received one abuse complaint from running back-to-back SNMP small-scale experiments (500 servers) on June 3, 2020. This com-

plaint came from a third-party monitoring framework called `greynoise.io` [16]; their goal is to notify the probing activities in the Internet and mass scanners (e.g., Censys [95], Shodan [30]) are also likely to be flagged by them [16]. We resolved this abuse complaint by discussing this with Cloudlab admins. We did not receive any other abuse complaints from our 10K server measurements. Across all 6 protocols, we also ran small-scale runs (300 servers) from our public-facing server. We are not aware that the campus network operators received any abuse complaints from these measurements.

SUBJECT: Vulnerable DDoS Amplifier

BODY: Security researchers at Carnegie Mellon University have been conducting Internet measurements to quantify the risk of amplification distributed denial-of-service (DDoS) attacks. Our team has noticed your system, IP with $PORT$ running $PROTOCOL$, can be abused to create an amplification attack (US-CERT). That means certain network queries can induce large responses (i.e., amplification factor as defined by US-CERT). Note that this may or may not be a result of mis-configuration of the server. An example of a network packet that can cause an amplification factor greater than 30 is: $PACKET INFO $

Please feel free to contact us at ampmap.proj@gmail.com should you have any questions and/or concerns. The details and motivation of our project can be found in $OUR WEB$.

Figure 5.20: A notification email to IP owners

5.5.2 Disclosure

Next, we discuss our steps for responsible disclosure to relevant stakeholders.

Notifying IP owners: We notified the IP owners whose servers can induce AF greater than 30. Following best practices, we obtained the abuse and/or contact email from WHOIS [137]. We include an example notification sent from a project's email, ampmap.proj@gmail.com in Figure 5.20. Table 5.9 shows the number of emails we sent and human (not automated) responses we got; e.g., for DNS, we send 1688 emails and received 17 responses. Example responses

include "Thanks $\cdots$ service detected on *ADDR* has been shutdown the time to install necessary mitigation" and "We were not even aware this was the case. we have disabled SNMP."

proto		# sent	# resp	proto		# sent	# resp
DNS		1688	17	SNMP$^{\text{AND}}$	GetBulk	3709	35
NTP $^{\text{OR}}$	private mode	85	0		GetNext	248	4
NTP $^{\text{AND}}$	private mode	620	1	SSDP		643	5
SNMP $^{\text{OR}}$	GetBulk	2208	27	Chargen		5999	9
	GetNext	55	0	Memcached		11	0

Table 5.9: Statistics on the # of notification emails we sent and the responses we got from system owners

Vulnerability reporting: We have initiated a process of disclosing our findings to the affected parties mediated by the CERT® Coordination Center (CERT/CC). CERT/CC has accepted our coordination request and is in the process of identifying and notifying the affected parties. Our findings require multi-party coordination because unexpected amplification is potentially a protocol issue, and so all relevant vendors need to be notified in a consistent manner.

Notifying the vendors: Our vulnerability reports to CERT/CC specify affected vendors for DNS, SNMP, and NTP.

5.6 Other Related Work on Amplification Attacks and Mitigation

We now discuss other related work on amplification attacks and mitigation. Prior works have pointed out that many network protocols have amplification vulnerabilities [154]. Rossow [165] discovered amplification vulnerabilities in 14 UDP-based protocols via a manual analysis on the code or reverse engineering. Follow-up research also analyzed detailed amplification vector in specific protocols by focusing on a specific set of features (e.g., analyzing DNSSEC

in DNS [175], $\mathrm{MONLIST}$ in NTP [85]). However, using AmpMap, we found many other $\mathrm{recordtype}$ values that can incur high AF. Some have looked at TCP-based amplification [134] which is outside the scope of AmpMap. There is also an active discussion on the mitigation of amplification attacks (e.g., [12, 19]). Further, some orthogonal efforts focus on monitoring [131] and linking [133] DDoS services. Our work is inspired by these prior efforts. To the best of our knowledge, AmpMap is the first to study the problem of *automatically* mapping Internet-wide amplification vulnerabilities precisely.

5.7 Summary

Given the constant evolution of protocols, server implementations, we need a systematic approach to map the DDoS amplification threat. AmpMap bridges this gap by synthesizing structural insights with careful measurement design to realize a low-overhead service called AmpMap. AmpMap can systematically confirm prior observations and also uncover new-possibly-hidden amplification patterns that are ripe for abuse. As future work, we plan to add support for more protocols and expand the scale of measurement to make this a continuous "health monitoring" service for the Internet.

Algorithm 5: Per-Field Search

```
1  Function PerFieldSearch(QtoAF, Q^start, AF^thresh):
2      Q^explore = {Q^start}; PatternsFound = {}
3      while Q^explore is not empty do
4          q ← Extract from Q^explore
5          if IsNewPattern(q.pattern , PatternsFound) then
               /* Search neighbors for a new pattern                    */
6              PatternsFound.insert(q.pattern)
7              tmpQtoAF = SearchNeighbor(q, AF^thresh)
8              QtoAF.insert(QtoAF^neighbor)
9              Q^explore = Q^explore ∪ tmpQtoAF.keys()
10         else
11             MergeQueries(q.pattern, PattersFound)
12     return QtoAF
13 Function SearchNeighbor(q, AF^thresh):
14     NeighborQToAF = {}
15     foreach protocol field f_i do
16         Q_i = {q[f_i ← v_i], for v_i ∈ Values_i}
17         QtoAF_i = SendQuery(q ∈ Q_i)
           /* Merge queries into contiguous ranges with high AF     */
18         HighRanges = InferPatternRange(q, Values_i, QtoAF_i, AF^thresh) /* Find
               representative sample from each range                 */
19         for ⟨v_l, v_r⟩ ∈ HighRanges do
20             patternid = q.pattern[f_i ← (v_l, v_r)]
21             q_n = q[f_i ← rand([v_i, v_r])]
22             NeighborQToAF.append( q_n → AF_n )
23     return NeighborQToAF
```

Chapter 6

Reflections, Limitations, and Future Work

In this chapter, we summarize the lessons and reflections from designing and running our tools (Section 6.1) and discuss the limitations of our proposed solutions (Section 6.2). We conclude the dissertation by identifying the key future research directions (Section 6.3).

6.1 Reflections and Lessons

Reflection #1: Black-box analysis is a necessary and practical alternative. A black-box analysis is necessary for scenarios where it is the only viable option (e.g., proprietary functions, vulnerability assessment for remote services). An interesting question is whether a black-box analysis is beneficial and necessary when other types of analysis (e.g., white-box) are also viable.

At a high level, we find that different types of analysis (e.g., white-box analysis) complement each other. For instance, white-box analysis ensures that the internal operations are performed according to the expectations. On the other hand, the black-box analysis does not have visibility into the internal logic and cannot discover such erroneous transition not visible from outputs. However, we find scenarios where even when the internal state (i.e., code state) is in an expected, correct state, a black-box function can still be exploited. When we were running Pryde, we saw a specific firewall forwarding an external DATA packet behaving as if the connection is

ESTABLISHED, even when a connection state was in a SYN-SENT state (observable through the user-interface).

Further, a black-box approach dynamically analyzes the system's functionality as a whole (i.e., a combination of source code, a configuration, among others) based on only the input-output behavior. For instance, AmpMap uncovered a significant variability of the amplification-inducing query patterns across servers with the identical software version (i.e., same code). We posit that this happens as amplification is not just a function of the source code but many other factors (e.g., configuration settings, the status of the server's data).

Reflection #2: We observe a significant variability of implementations across vendors and versions. Our findings suggest that we cannot merely assume that network functions and services of the same type or with identical software (1) have a homogeneous implementation, and (2) are susceptible to an identical set of vulnerabilities. Otherwise, we will miss out on significant residual risk or miss our critical security implications. We briefly summarize the variability:

- Alembic (Chapter 3): NFs of the same type (e.g., load balancers) have starkly different behavior across vendors. For example, HAProxy load balancer was a connection-terminating NF. In contrast, the PfSense was behaving like a destination NAT.
- Pryde (Chapter 4): The discovered evasion vulnerabilities were highly implementation-specific. For example, FW-2 was forwarding a DATA packet from an external attacker (even with an explicit drop rule). In contrast, the ProprietaryNF firewall waited for exchanges of SYN and SYN-ACK packets with correct seq/ack numbers.
- AmpMap (Chapter 5): We uncover a significant diversity of amplification-inducing patterns across servers. Hence, only focusing on a handful of patterns will leave significant residual for each server. The motivation for building AmpMap (empirically validated in Section 5.1) is the variability of query patterns and the risk across servers.

Reflection #3: Our approaches are highly effective in systematically uncovering new vulnerabilities and confirming known attacks. While prior literature has uncovered evasion

attacks against IDSes [160] and censorship firewalls (e.g., [69, 179]), Pryde takes a systematic approach to uncover evasion attacks against enterprise firewalls. These attacks exploit more fundamental implementation errors in tracking per-connection states. Using Pryde, we uncovered 294 to 8,220 semantically-distinct attacks across four popular firewalls. Furthermore, with our systematic workflow, we showcase how having a weak insider can enable strong attacks. Similarly, while an amplification attack is not a new concept in the security community, AmpMap revealed a non-trivial number of new amplification-inducing patterns and polymorphic variants of the known patterns. For instance, while prior work only highlighted ANY for DNS, more than 20% of our sampled servers can yield more than 10 AF with 19 other $\mathrm{recordtype}$ values. Our findings across tools suggest: (1) such errors are prevalent in the implementations and design of these functions and services; and (2) solely focusing on one feature or one attack vector is insufficient for mitigation.

Reflection #4: We need structure-based approaches to discover these subtle attacks. While Pryde uncovered thousands of attacks, a pure random-based strategy only uncovered 1 to 3 (raw) attacks for two complex firewall implementations. Similarly, for AmpMap, using random-based strategies (i.e., simulated annealing, random search) is ineffective in discovering multiple amplification patterns. These findings suggest that (1) these behavior and vulnerabilities are subtle that simple approaches are ineffective; and (2) we need to leverage domain-specific insights to analyze complex network functions and services.

6.2 Limitations

We now examine the limitations of our proposed solutions.

Limitation #1: Observable packet output: We assume a black-box function or service that takes an input packet and emits packet(s). Hence, our current tools do not support the types of behavior unobservable as an output packet. These include software bugs (e.g., buffer overflow,

privilege escalation), behavior affecting availability (e.g., battery outage), or physical behavior (e.g., a sudden temperature increase). While our insights may be relevant for these settings, we would need different mechanisms to identify the internal state or the cyber-physical aspects.

Limitation #2: Complex device behavior: Our techniques currently cannot capture temporal (e.g., timeout) or quantitative behavior (e.g., rate-limiting, reduced throughput). Supporting these properties would enable our techniques to handle more complex functions and services (e.g., WAN optimizer) and reason about complex bugs (e.g., performance-related bugs).

Limitation #3: Assumptions about the input space: We make certain assumptions about the input space to reduce the relevant search space. For instance, in building Alembic and Pryde, we only consider specific header fields such as IP addresses, ports, sequence, and acknowledgment numbers, but not others (e.g., checksum, TTL). Similarly, we only focus on network-layer behavior or attacks, but not application-layer behavior or attacks across all three tools.

Limitation #4: Knowledge of input format: We assume that we know the input formats (e.g., a format of a DNS packet) to these network functions and services (similar to other black-box approaches [64, 171]). However, there may be scenarios where having such knowledge may be infeasible; e.g., for proprietary protocols, we may not know the protocol-compliant formats.

Limitation #5: Identifying the structural properties: Leveraging relevant structural properties allows us to discover subtle behavior and uncover multiple security vulnerabilities. However, extracting the right kinds of structural properties can be challenging. For example, in building AmpMap, we identified these properties through an iterative process of coming up with hypotheses and validating them through visualization and local setup testing. However, we observe that once we identify these properties, we can build automated solutions general across vendors and implementations.

Limitation #6: Root cause diagnosis: As with any black-box approach, explaining the root cause of the behavior or security implications is currently outside the scope of our work. Recall that using AmpMap, we uncovered a significant diversity of query patterns across server

instances (even those with the same software setups). While we posit that this happens as amplification is also a function of other factors (e.g., exact configuration), we do not have visibility into these factors (e.g., configurations, the status of the data contained in the server). While we can hypothesize a host of factors, fully explaining our observation is outside the scope (of any black-box approach). Similarly, we uncovered hundreds to thousands of semantically-distinct bugs using Pryde. While we can approximate the distinct semantics from the stateful model, we also do not have visibility into the internal workings. Hence, fully explaining the "root cause" of these bugs is also outside the scope.

6.3 Future Work

Our ultimate vision for enabling analysis of these network functions and services is to make networks more secure and future-proof to the growing diversity of exploitable network functions and services. Our work in this dissertation has laid some steps towards achieving our goal. To this end, we now identify two broad research directions:

- Given the complexity of functions, services, and their interactions, we need to **enhance black-box analysis techniques** to reason about complex behavior (Section 6.3.1).
- Given that the ultimate goal is to secure them from future attacks, we lay out future directions toward **securing our modern network infrastructure** (Section 6.3.2).

6.3.1 Enhancing Black-Box Analysis Techniques

Supporting temporal and quantitative properties: Representing temporal or quantitative properties would benefit our tools to uncover performance-related behavior and verify such properties. To represent quantitative properties (e.g., rate-limiting), we posit that we need to incorporate these as part of the inputs to consider (e.g., sessions sent at a certain rate) and monitor relevant properties. Similarly, to handle temporal effects (e.g., timeout), we need to add the

passage of time (e.g., wait for 30 s) to our input space.

While we could extend our current tools to handle this behavior, it may be worth considering more native abstractions than deterministic FSMs. For instance, many have proposed different abstractions to represent quantitative (e.g., [58, 60, 116, 145]) and timing properties (e.g., [59]). Once we pick the abstraction, we can find relevant techniques that extend L* (i.e., [66, 112]). It is not trivial to find one abstraction to model multiple properties at once, and we need to pick the abstraction based on the properties of interest.

Handling more complex device interactions: Our current tool focuses on a single device. However, network systems are composed of complex interactions between multiple functions, services, and protocols. Thus, these scenarios can benefit from analyzing the network infrastructure as a whole (e.g., service function chaining [70], collateral damage between application services). For instance, in building Pryde, we considered a non-interactive victim that accepts any TCP packet. Supporting a stateful victim would enable us to discover more sophisticated attacks such as an attacker exfiltrating data from a protected server.

Supporting a broader set of inputs: Our tools consider reasoning about the behavior using a scoped set of input packets or header fields. Adding support for other fields such as bad checksum, TTL would enable our techniques to discover other behaviors and security vulnerabilities. While these may sound relatively simple, we will run into scalability challenges. Hence, we posit that it may be useful to combine our efforts with structure-free efforts [69] as they can quickly explore more TCP fields. For instance, we envision a pre-processing step where these tools inform which header fields are worth systematically exploring.

Supporting application-layer functions, services, and attacks against them: More complex network functions such as layer-7 load balancers (LB), transparent proxy, or deep packet inspection (DPI) operate at the application layer. To support these network functions, we would need to model the multi-layer interactions. Further, AmpMap focused solely on network layer attacks such as DNS amplification. However, application-layer DDoS attacks are also particularly effec-

tive as they consume server resources and creates more damage, with less total bandwidth [39].

Automatically extracting structural properties: Identifying and leverage the right kinds of structural properties are crucial in enabling our scalable solutions (across three applications). However, we identified these properties from our domain expertise or through an iterative process of validating hypotheses (as mentioned in Section 6.2). Therefore, automatically extracting relevant structural insight to build automated tools to black-box analysis can be a promising direction. For instance, we could develop candidate templates that test for specific properties (e.g., testing independence or similarity across input space). That way, we can automatically synthesize the high-level workflow of a solution given a specific problem.

Supporting more general goals: As mentioned in Section 1.3, AmpMap and Pryde focus on more specific goals compared to that of Alembic. Pryde focuses on network firewalls, and AmpMap focuses on DDoS amplification attacks. However, Pryde's general techniques can be relevant to synthesize attacks against other NFs. That is, we can still leverage the same workflow of first inferring a stateful model and use it to systematically uncover attacks that meet certain properties of interest. Extending Pryde's general workflow for other NFs, beyond network firewalls, can be an interesting future direction. Similarly, it would be interesting to explore whether AmpMap's insights can transfer to other types of attacks (e.g., performance degradation for network services).

Leveraging machine learning techniques: We can leverage machine learning techniques to fine-tune our input parameters automatically. For instance, the current AmpMap algorithm can benefit from parameter tuning, e.g., automatically decide the % spent on the RandomSample Stage based on the density observed so far. Further, machine learning techniques may provide a path forward in handle scenarios where we do not know the input format. Furthermore, the problem that AmpMap tackles can be also viewed as a black-box optimization problem. Hence, one interesting future work is to leverage and customize these techniques for AmpMap's purpose; e.g., derivative-free optimization [100, 120, 163] or Bayesian Optimization that can optimize for

a black-box function. For instance, we would need to customize these algorithms to achieve coverage rather than finding the maximum value and also handle server diversity, and these efforts can benefit from our observations and insights.

Our recent vision paper [140] highlights that GANs are appealing for their ability to infer constraints and correlations in the inputs for synthesizing inputs that meet specific properties (e.g., protocol formats). Hence, it would be helpful to use such approaches to infer protocol format or even learn the relationship between header fields.

6.3.2 Securing Our Network Infrastructure

Informing defenses by automatically patching network functions and protocols: The ultimate goal for uncovering security implications is to automatically patch these vulnerabilities. Building a system that can adaptive and automatically install patches would be fruitful would raise exciting research questions. How would we generate these patches? How do we ensure the correctness or non-interference with normal operations of these devices and services? For example, for stateful network functions, generating a model that undoes the effects of identified vulnerabilities. Further, systematically generate signatures for blocking these input packets (e.g., packet values that incur amplification from AmpMap) would be an interesting future direction. However, this would be challenging as we want a low false-positive rate not to interfere with legitimate traffic.

Developing correct-by-construction function, services, and protocols: Our thesis focuses on synthesizing behavior models for accurate verification or identifying exploitable inputs for legacy network functions and services. However, these legacy functions and services can be ripe for abuse. One interesting direction that would be particularly impactful is to design techniques to make legacy functions, protocols, and services correct-by-construction with *minimum code changes*. An alternate way is to use a clean-slate approach where we build *correct-by-construction* network functions, services, and protocols (similar to goals of [187, 188]). How-

ever, building these would raise interesting research challenges. First, we need to know what properties to enforce to make these functions and services correct-by-construction. Second, enforcing these desired properties requires an expressive language or a framework to describe these high-level specifications. We may also need to explore how to synthesize these specifications automatically. All of these directions would help us move toward securing our network infrastructure with correct network functions, services, and protocols.

Bibliography

[1] Oulu University Secure Programming Group: PROTOS Test-Suite: c06-snmpv1. Technical report, University of Oulu, Electrical and Information Engineering. 22

[2] 5 DDOS Attack Trends to Watch in 2020 . https://www.indusface.com/blog/ddos-attack-trends/. 5

[3] Alert (TA13-088A) UDP-Based Amplification Attacks. https://www.us-cert.gov/ncas/alerts/TA13-088A. 5, 112, 115, 119, 120, 141

[4] Alert (TA14-013A) NTP Amplification Attacks Using CVE-2013-5211. https://www.us-cert.gov/ncas/alerts/TA14-013A. 5, 112, 115, 141

[5] Alert (TA14-017A) UDP-Based Amplification Attacks. https://www.us-cert.gov/ncas/alerts/TA14-017A. 5, 112, 114, 115, 119, 141, 151

[6] Arbor DDoS. https://www.netscout.com/arbor-ddos. 5

[7] Broken packets: IP fragmentation is flawed. https://blog.cloudflare.com/ip-fragmentation-is-broken/. 149

[8] Cloudflare DDoS Protection. https://www.cloudflare.com/ddos/. 5

[9] Cloudflare outage on July 17, 2020. https://blog.cloudflare.com/cloudflare-outage-on-july-17-2020/. 2

[10] *CyberGreen*. https://stats.cybergreen.net/. 6, 8, 9, 15, 19, 113, 117, 118

[11] *DDoS Attacks Get Bigger, Smarter and More Diverse*. https://threatpost.com/

ddos-attacks-get-bigger-smarter-and-more-diverse/134028/. 112, 116

[12] Dns reflection defense. https://blogs.akamai.com/2013/06/dns-reflection-defense.html. 159

[13] *DNS SURVEY: OPEN RESOLVERS*. http://dns.measurement-factory.com/surveys/openresolvers.html. 19, 113

[14] Executive Order 13800 - Strengthening the Cybersecurity of Federal Networks and Critical Infrastructure. https://www.govinfo.gov/content/pkg/DCPD-201700327/pdf/DCPD-201700327.pdf. 6, 113, 117

[15] *Flooding the web: The internet's epic attack amplification problem*. https://www.theguardian.com/technology/2014/feb/24/flooding-the-web-attack-amplification. 112

[16] Grey Noise. https://greynoise.io/about. 157

[17] Haproxy. http://www.haproxy.org/. 29, 63

[18] *Here's how much money a business should expect to lose if they're hit with a DDoS attack*. https://www.techrepublic.com/article/heres-how-much-money-a-business-should-expect-to-lose-if-theyre-hit-with-a-ddos-attack/. 112, 116

[19] How to defend against amplification attacks. https://www.information-age.com/how-defend-against-amplification-attacks-123457736/. 159

[20] How Verizon and a BGP Optimizer Knocked Large Parts of the Internet Offline Today. https://blog.cloudflare.com/how-verizon-and-a-bgp-optimizer-knocked-large-parts-of-the-internet-offline-today/. 2

[21] iPerf Performance Tool. https://iperf.fr/. 65, 66

[22] IPv6, Large UDP Packets and the DNS. http://www.potaroo.net/ispcol/2017-08/xtn-hdrs.html. 149

[23] jsonrpc. https://github.com/briandilley/jsonrpc4j. 62

[24] Memcrashed - Major amplification attacks from UDP port 11211. https://blog.cloudflare.com/memcrashed-major-amplification-attacks-from-port-11211/. 2, 5, 112, 117

[25] Open Memcached Key-Value Store Scanning Project. https://memcachedscan.shadowserver.org/. 141

[26] pfsense. https://www.pfsense.org/. 29, 63

[27] Scapy. http://www.secdev.org/projects/scapy/. 62, 98

[28] Security Bulletin: Crafted DNS Text Attack. https://tinyurl.com/y9zpevuy. 115, 141

[29] ShadowServer. https://www.shadowserver.org/. 113

[30] SHODAN. https://www.shodan.io/. 123, 136, 156, 157

[31] Technical Details Behind a 400Gbps NTP Amplification DDoS Attack. https://blog.cloudflare.com/technical-details-behind-a-400gbps-ntp-amplification-ddos-attack/. 5, 117

[32] The DDoS That Almost Broke the Internet. https://blog.cloudflare.com/the-ddos-that-almost-broke-the-internet/. 5, 117

[33] UDP-Based Amplification Attacks. https://www.us-cert.gov/ncas/alerts/TA14-017A. 118, 119, 153

[34] untangle. https://www.untangle.com/. 29, 32, 63

[35] Virtualbox. https://www.virtualbox.org/. 63, 98

[36] What is an intrusion detection system? How an IDS spots threats. https://www.csoonline.com/article/3255632/what-is-an-intrusion-

detection-system-how-an-ids-spots-threats.html, last accessed August 16, 2020. 1

[37] What Is DNS? — How DNS Works. https://www.cloudflare.com/learning/dns/what-is-dns/, last accessed August 16, 2020. 1

[38] What Is Load Balancing? https://www.nginx.com/resources/glossary/load-balancing/, last accessed August 16, 2020. 1

[39] Application Layer DDoS Attack. https://www.cloudflare.com/learning/ddos/application-layer-ddos-attack/, last accessed August 17, 2020. 167

[40] AWS Marketplace. https://aws.amazon.com/marketplace, last accessed July 31, 2020. 77, 86, 98

[41] boofuzz: Network protocol fuzzing for humans. https://boofuzz.readthedocs.io/en/latest/, last accessed July 31, 2020. 76

[42] 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 July 31, 2020. 4, 75

[43] 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 July 31, 2020. 84

[44] Firewall Penetration Testing: Steps, Methods And Tools That Work. https://purplesec.us/firewall-penetration-testing/, last accessed July 31, 2020. 85

[45] IoT security fail: The weird devices that employees are connecting to the office network. https://www.zdnet.com/article/iot-security-warning-

employees-are-connecting-these-unauthorised-devices-to-your-network/, last accessed July 31, 2020. 83

[46] Katherine Pryde. https://marvel-movies.fandom.com/wiki/Katherine_Pryde, last accessed July 31, 2020. 12, 75, 76

[47] Microsoft: Russian state hackers are using IoT devices to breach enterprise networks. https://www.zdnet.com/article/microsoft-russian-state-hackers-are-using-iot-devices-to-breach-enterprise-networks/, last accessed July 31, 2020. 83

[48] Red Hat Sprucing OpenShift for Network Functions on Kubernetes. https://www.lightreading.com/nfv/red-hat-sprucing-openshift-for-network-functions-on-kubernetes/d/d-id/754828, last accessed July 31, 2020. 4, 75

[49] Rogue IoT devices are putting your network at risk from hackers. https://www.zdnet.com/article/rogue-iot-devices-are-putting-your-network-at-risk-from-hackers/, last accessed July 31, 2020. 83

[50] Sulley: Fuzzing Framework. http://www.fuzzing.org/wp-content/SulleyManual.pdf, last accessed July 31, 2020. 76

[51] The Importance of Using a Firewall for Threat Protection. https://www.websecurity.digicert.com/security-topics/importance-using-firewall-threat-protection, last accessed July 31, 2020. 1, 75

[52] The IoT: Gateway for enterprise hackers. https://www.csoonline.com/article/3148806/the-iot-gateway-for-enterprise-hackers.html, last accessed July 31, 2020. 83

[53] Humberto J. Abdelnur, Radu State, and Olivier Festor. Kif: A stateful sip fuzzer. In *Proc. IPTComm*, 2007. 15, 21, 22

[54] Adel El-Atawy, K. Ibrahim, H. Hamed, and Ehab Al-Shaer. Policy segmentation for intelligent firewall testing. In *Proc. IEEE ICNP Workshop on Secure Network Protocols*, 2005. 76, 110

[55] Hari Adiseshu, Subhash Suri, and Guru M. Parulkar. Detecting and resolving packet filter conflicts. In *Proc. IEEE INFOCOM*, 2000. 110

[56] Ehab Al-Shaer, Adel El-Atawy, and Taghrid Samak. Automated pseudo-live testing of firewall configuration enforcement. *IEEE J. Sel. Areas Commun.*, 27(3):302–314, 2009. 76, 110

[57] Maryam Raiyat Aliabadi, Amita Ajith Kamath, Julien Gascon-Samson, and Karthik Pattabiraman. Artinali: Dynamic invariant detection for cyber-physical system security. In *Proc. ESEC/FSE*, 2017. 25

[58] Rajeev Alur, Loris DAntoni, Jyotirmoy Deshmukh, Mukund Raghothaman, and Yifei Yuan. Regular functions and cost register automata. In *Proc. LICS*, 2013. 166

[59] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, April 1994. 166

[60] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular programming for quantitative properties of data streams. In *Proc. European Symposium on Programming Languages and Systems*, pages 15–40, New York, NY, USA, 2016. Springer-Verlag New York, Inc. 166

[61] David G. Andersen, Hari Balakrishnan, Nick Feamster, Teemu Koponen, Daekyeong Moon, and Scott Shenker. Accountable internet protocol (aip). In *Proc. SIGCOMM*, 2008. 112

[62] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, November 1987. 9, 12, 28, 29, 37, 42, 43, 73, 74, 97

[63] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime

Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *Proc. USENIX Security Symposium*, 2017. 2

[64] George Argyros, Ioannis Stais, Suman Jana, Angelos D. Keromytis, and Aggelos Kiayias. Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In *Proc. CCS*, 2016. 8, 15, 20, 74, 164

[65] M. Bailey, D. Dittrich, E. Kenneally, and D. Maughan. The menlo report. *IEEE Security Privacy*, 10(2):71–75, 2012. 156

[66] Borja Balle and Mehryar Mohri. Learning weighted automata. In Andreas Maletti, editor, *Algebraic Informatics*, pages 1–21, Cham, 2015. Springer International Publishing. 166

[67] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. Snooze: Toward a stateful network protocol fuzzer. In *Proc. ISC*, 2006. 15, 21, 22

[68] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets. In *Proc. SIGCOMM*, 2005. 15, 22

[69] Kevin Bock, George Hughey, Xiao Qiang, and Dave Levin. Geneva: Evolving censorship evasion strategies. In *Proc. CCS*, 2019. 8, 15, 19, 24, 76, 78, 79, 81, 82, 91, 102, 163, 166

[70] M. Boucadair, C. Jacquenet, R. Parker, D. Lopez, and C. Pignataro J. Guichard. Service function chaining (sfc) use cases. `https://tools.ietf.org/html/draft-boucadair-service-chaining-framework-00`, 2013. 166

[71] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proc. CCS*, 2007. 8, 15, 17, 18

[72] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, 2008. 23

[73] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V. Krishnamurthy, and Lisa M. Marvel. Off-path tcp exploits: Global rate limit considered dangerous. In *Proc. USENIX Security Symposium*, 2016. 18

[74] Z. Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. Sensitive information tracking in commodity iot. In *Proc. USENIX Security Symposium*, 2018. 25

[75] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. Soteria: Automated iot safety and security analysis. In *Proc. ATC*, 2018. 24, 25

[76] S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *Proc. IEEE Symposium on Security and Privacy*, 2015. 23

[77] Weiteng Chen and Zhiyun Qian. Off-path TCP exploit: How wireless routers can jeopardize your secrets. In *Proc. USENIX Security Symposium*, 2018. 15, 18

[78] Y. Chen, C. M. Poskitt, and J. Sun. Learning from mutants: Using code mutation to learn and monitor invariants of a cyber-physical system. In *Proc. IEEE Symposium on Security and Privacy*, 2018. 25

[79] Chia Yuan Cho, Domagoj Babi ć, Eui Chul Richard Shin, and Dawn Song. Inference and analysis of formal models of botnet command and control protocols. In *Proc. CCS*, 2010. 20

[80] Chia Yuan Cho, Domagoj Babić, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proc. USENIX Security Symposium*, 2011. 15, 17, 20

[81] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw.*

Eng., 4(3):178–187, May 1978. 44, 46, 47

[82] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001. 88, 96

[83] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In *Proc. IEEE Symposium on Security and Privacy*, 2009. 18

[84] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proc. USENIX Security Symposium*, 2003. 15

[85] Jakub Czyz, Michael Kallitsis, Manaf Gharaibeh, Christos Papadopoulos, Michael Bailey, and Manish Karir. Taming the 800 pound gorilla: The rise and decline of ntp ddos attacks. In *Proc. IMC*, 2014. 6, 118, 141, 142, 144, 159

[86] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. A derivation system for security protocols and its logical formalization. In *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.*, pages 109–125, 2003. 22

[87] Anupam Datta, Ante Derek, John Mitchell, and Dusko Pavlovic. Secure protocol composition. volume 83, pages 11–23, 10 2003. 22

[88] Anupam Datta, Ante Derek, John C Mitchell, and Arnab Roy. Protocol composition logic (pcl). *Electronic Notes in Theoretical Computer Science*, 172:311–358, 2007. 22

[89] Anupam Datta, Ante Derek, John C. Mitchell, Vitaly Shmatikov, and Mathieu Turuani. Probabilistic polynomial-time semantics for a protocol security logic. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Automata, Languages and Programming*, pages 16–29, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. 22

[90] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In

TACAS, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. 88, 97

[91] Wenbo Ding and Hongxin Hu. On the safety of iot device physical interaction control. In *Proc. CCS*, 2018. 24, 25

[92] Mihai Dobrescu and Katerina Argyraki. Software dataplane verification. In *Proc. NSDI*, 2014. 15, 16

[93] Samuel Drews and Loris D'Antoni. Learning symbolic automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 173–189. Springer, 2017. 74

[94] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proc. ATC*, 2019. 63, 98, 136, 156

[95] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. A search engine backed by internet-wide scanning. In *Proc. CCS*, 2015. 1, 113, 119, 123, 136, 156, 157

[96] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Zmap: Fast internet-wide scanning and its security applications. In *Proc. USENIX Security Symposium*, 2013. 1, 113, 156

[97] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. Buzz: Testing context-dependent policies in stateful networks. In *Proc. NSDI*, 2016. 3, 12, 17, 24, 26, 28, 31, 32, 36, 72, 73

[98] E. Fernandes, J. Jung, and A. Prakash. Security analysis of emerging smart home applications. In *Proc. IEEE Symposium on Security and Privacy*, 2016. 24, 25

[99] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. Flowfence: Practical data protection for emerging iot application frameworks. In *Proc. USENIX Security Symposium*, 2016. 24, 25

[100] Daniel E Finkel. Direct optimization algorithm user guide. 2003. 167

[101] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. Learning fragments of the tcp network protocol. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 78–93. Springer, 2014. 20

[102] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. Combining model learning and model checking to analyze tcp implementations. In *International Conference on Computer Aided Verification*, pages 454–471. Springer, 2016. 20, 74

[103] Philippe Flajolet, Danièle Gardy, and Loÿs Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Appl. Math.*, 39(3):207–229, November 1992. 135

[104] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *Proc. NSDI*, 2015. 73

[105] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Trans. Softw. Eng.*, 17(6):591–603, June 1991. 44, 46

[106] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *SecureComm*, volume 164 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 330–347. Springer, 2015. 8, 15, 20, 76

[107] Erol Gelenbe, Gökçe Görbil, Dimitrios Tzovaras, Steffen Liebergeld, David Garcia, Madalina Baltatu, and George Lyberopoulos. Nemesys: Enhanced network security for

seamless service provisioning in the smart mobile ecosystem. In *Information Sciences and Systems 2013*, pages 369–378. Springer, 2013. 8, 21

[108] Patrice Godefroid et al. Dart: Directed automated random testing. In *Proc. PLDI*, 2005. 23

[109] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Proc. NDSS*, 2008. 23

[110] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014. 9, 21

[111] Serge Gorbunov and Arnold Rosenbloom. Autofuzz: Automated network protocol fuzzing framework. *IJCSNS*, 10(8):239, 2010. 15, 76

[112] Olga Grinchtein. *Learning of timed systems*. PhD thesis, Acta Universitatis Upsaliensis, 2008. 166

[113] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '02, pages 357–370, London, UK, UK, 2002. Springer-Verlag. 74

[114] Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security Symposium*, 2001. 111

[115] Changhua He, Mukund Sundararajan, Anupam Datta, Ante Derek, and John C. Mitchell. A modular correctness proof of ieee 802.11i and tls. In *Proc. CCS*, 2005. 22

[116] T. A. Henzinger. The theory of hybrid automata. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pages 278–292, 1996. 166

[117] Grant Ho, Derek Leung, Pratyush Mishra, Ashkan Hosseini, Dawn Song, and David Wag-

ner. Smart locks: Lessons for securing commodity internet of things devices. In *Proc. AsiaCCS*, 2016. 24, 25

[118] Syed Rafiul Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. Lteinspector: A systematic approach for adversarial testing of 4g LTE. In *Proc. NDSS*, 2018. 20

[119] Internet Systems Consortium. Using the response rate limiting feature. `https://kb.isc.org/docs/aa-00994`, 9 2018. 148

[120] Kevin G Jamieson et al. Query complexity of derivative-free optimization. In *Proc NIPS*, pages 2672–2680, 2012. 167

[121] S. Jero, H. Lee, and C. Nita-Rotaru. Leveraging state information for automated attack discovery in transport protocol implementations. In *Proc. DSN*, 2015. 15, 21

[122] Samuel Jero, Md. Endadul Hoque, David R. Choffnes, Alan Mislove, and Cristina Nita-Rotaru. Automated attack discovery in TCP congestion control using a model-guided approach. In *Proc. NDSS*, 2018. 20

[123] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, October 2009. 20

[124] Laurent Joncheray. A simple active attack against tcp. In *Proc. USENIX Security Symposium*, 1995. 15, 18

[125] D. Joseph and I. Stoica. Modeling middleboxes. *Netwrk. Mag. of Global Internetwkg.*, 2008. 26

[126] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proc. NSDI*, 2012. 38, 73, 94

[127] Sheharbano Khattak, Mobin Javed, Philip D. Anderson, and Vern Paxson. Towards illuminating a censorship monitor's model to facilitate evasion. In *Proc. USENIX Workshop on FOCI*, 2013. 15, 18

[128] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey.

Veriflow: Verifying network-wide invariants in real time. In *Proc. NSDI*, 2013. 73

[129] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000. 16, 29, 31, 63, 64, 68, 69, 72

[130] Nupur Kothari, Ratul Mahajan, Todd Millstein, Ramesh Govindan, and Madanlal Musuvathi. Finding protocol manipulation attacks. In *Proc. SIGCOMM*, 2011. 15, 16, 22

[131] Lukas Krämer, Johannes Krupp, Daisuke Makita, Tomomi Nishizoe, Takashi Koide, Katsunari Yoshioka, and Christian Rossow. Amppot: Monitoring and defending against amplification ddos attacks. In *Proc. RAID*, 2015. 159

[132] Tammo Krueger, Hugo Gascon, Nicole Krämer, and Konrad Rieck. Learning stateful models for network honeypots. In *Proc. ACM Workshop on Security and Artificial Intelligence*, 2012. 20

[133] Johannes Krupp, Mohammad Karami, Christian Rossow, Damon McCoy, and Michael Backes. Linking amplification ddos attacks to booter services. In *Proc. RAID*, 2017. 159

[134] Marc Kührer, Thomas Hupperich, Christian Rossow, and Thorsten Holz. Exit from Hell? Reducing the Impact of Amplification DDoS Attacks. In *Proc. USENIX Security Symposium*, 2014. 159

[135] Sanjeev Kumar. Smurf-based distributed denial of service (ddos) attack amplification in internet. In *Proc. ICIMP*, 2007. 124

[136] Fangfan Li, Abbas Razaghpanah, Arash Molavi Kakhki, Arian Akhavan Niaki, David Choffnes, Phillipa Gill, and Alan Mislove. Lib•erate, (n): A library for exposing (traffic-classification) rules and avoiding them efficiently. In *Proc. IMC*, 2017. 15, 18, 19, 78

[137] Frank Li, Zakir Durumeric, Jakub Czyz, Mohammad Karami, Michael Bailey, Damon McCoy, Stefan Savage, and Vern Paxson. You've got vulnerability: Exploring effective vulnerability notifications. In *Proc. USENIX Security Symposium*, 2016. 157

[138] Li Haifeng, Wang Shaolei, Zhang Bin, Shuai Bo, and Tang Chaojing. Network protocol security testing based on fuzz. In *Proc. ICCSNT*, 2015. 19

[139] Chieh-Jan Mike Liang, Börje F. Karlsson, Nicholas D. Lane, Feng Zhao, Junbei Zhang, Zheyi Pan, Zhao Li, and Yong Yu. Sift: Building an internet of safe things. In *Proc. IPSN*, 2015. 24, 25

[140] Zinan Lin, Soo-Jin Moon, Carolina M. Zarate, Ritika Mulagalapalli, Sekar Kulandaivel, Giulia Fanti, and Vyas Sekar. Towards oblivious network analysis using generative adversarial networks. In *Proc. Workshop on Hot Topics in Networks*, 2019. 15, 21, 168

[141] Christian Makaya and Douglas Freimuth. Automated virtual network functions onboarding. In *Proc. IEEE SDN-NFV Conference*, 2016. 3

[142] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019. 14, 17, 23

[143] Stephen McLaughlin. A trusted safety verifier for process controller code. 2014. 24, 25

[144] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990. 23

[145] Mehryar Mohri. Weighted automata algorithms. In *Handbook of weighted automata*, pages 213–254. Springer, 2009. 166

[146] Soo-Jin Moon, Jeffrey Helt, Yifei Yuan, Yves Bieri, Sujata Banerjee, Vyas Sekar, Wenfei Wu, Mihalis Yannakakis, and Ying Zhang. Alembic: Automated model inference for stateful network functions. In *Proc. NSDI*, 2019. 11, 26, 73, 84, 88

[147] Soo-Jin Moon, Yucheng Yin, Rahul Anand Sharma, Yifei Yuan, Jonathan M. Spring, and Vyas Sekar. Accurately measuring global risk of amplification attacks using ampmap. *To Appear in USENIX Security Symposium*, 2021. 12, 113

[148] David Moore, Colleen Shannon, Douglas J. Brown, Geoffrey M. Voelker, and Ste-

fan Savage. Inferring internet denial-of-service activity. *ACM Trans. Comput. Syst.*, 24(2):115–139, May 2006. 5

[149] S. Munir and J. A. Stankovic. Depsys: Dependency aware integration of cyber-physical systems for smart homes. In *Proc. ICCPS*, 2014. 24, 25

[150] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proc. NSDI*, 2004. 22

[151] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V. Krishnamurthy, Edward J. M. Colbert, and Patrick McDaniel. Iotsan: Fortifying the safety of iot systems. In *Proc. CoNEXT*, 2018. 24, 25

[152] Aurojit Panda, Ori Lahav, Katerina J. Argyraki, Mooly Sagiv, and Scott Shenker. Verifying reachability in networks with mutable datapaths. In *Proc. NSDI*, 2017. 3, 12, 17, 26, 28, 31, 32, 37, 72, 73

[153] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Proc. USENIX Security Symposium*, 1998. 15, 16, 78, 79

[154] Vern Paxson. An analysis of using reflectors for distributed denial-of-service attacks. *SIGCOMM CCR*, 31(3):38–47, July 2001. 112, 116, 117, 158

[155] Luis Pedrosa, Ari Fogel, Nupur Kothari, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. Analyzing Protocol Implementations for Interoperability. In *Proc. NSDI*, 2015. 8, 15, 16, 22

[156] Luis Pedrosa, Rishabh Iyer, Arseniy Zaostrovnykh, Jonas Fietz, and Katerina Argyraki. Automated synthesis of adversarial workloads for network functions. In *Proc. SIGCOMM*, 2018. 15, 16

[157] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. *J. Autom. Lang. Comb.*, 7(2):225–246, November 2001. 74

[158] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Auto-

mated domain-independent detection of algorithmic complexity vulnerabilities. In *Proc. CCS*, 2017. 23

[159] pfsense. Inbound Load Balancing. `https://doc.pfsense.org/index.php/Inbound_Load_Balancing`. 72

[160] Thomas H Ptacek and Timothy N Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, SECURE NETWORKS INC CALGARY ALBERTA, 1998. 15, 78, 79, 163

[161] Z. Qian and Z. M. Mao. Off-path tcp sequence number inference attack - how firewall middleboxes reduce security. In *Proc. IEEE Symposium on Security and Privacy*, 2012. 15, 18

[162] Harald Raffelt, Bernhard Steffen, and Therese Berg. Learnlib: A library for automata learning and experimentation. In *Proc. ACM FMICS 2005*. 46, 62, 97

[163] Luis Miguel Rios and Nikolaos V Sahinidis. Derivative-free optimization: a review of algorithms and comparison of software implementations. *Journal of Global Optimization*, 56(3):1247–1293, 2013. 167

[164] E. Ronen and A. Shamir. Extended functionality attacks on iot devices: The case of smart lights. In *Proc. IEEE European Symposium on Security and Privacy 2020*, 2016. 25

[165] Christian Rossow. Amplification Hell: Revisiting Network Protocols for DDoS Abuse. In *Proc. NDSS*, 2014. 15, 112, 113, 114, 115, 117, 118, 119, 141, 142, 151, 158

[166] Thijs Rozekrans and Javy de Koning. Defending against DNS reflection amplification attacks. `https://tinyurl.com/bvw3d85`. 114

[167] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. Tcp congestion control with a misbehaving receiver. *SIGCOMM Comput. Commun. Rev.*, 29(5):71–78, October 1999. 18

[168] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. Mtfuzz: Fuzzing

with a multi-task neural network. *CoRR*, abs/2005.12392, 2020. 24

[169] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: efficient fuzzing with neural program smoothing. In *Proc. IEEE Symposium on Security and Privacy*, 2019. 18, 23, 24

[170] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proc. SIGCOMM*, 2012. 1, 2

[171] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D. Keromytis, and Suman Jana. Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations. In *Proc. IEEE Symposium on Security and Privacy*, 2017. 8, 15, 20, 74, 164

[172] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Scalable symbolic execution for modern networks. In *Proc. SIGCOMM*, 2016. 3, 17, 26, 31, 32, 73

[173] Brendan Tschaen, Ying Zhang, Theo Benson, Sujata Benerjee, JK Lee, and Joon-Myung Kang. SFC-Checker: Checking the Correct Forwarding Behavior of Service Function Chaining. In *IEEE SDN-NFV Conference*, 2016. 31

[174] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. 20, 77

[175] Roland van Rijswijk-Deij, Anna Sperotto, and Aiko Pras. Dnssec and its potential for ddos attacks: A comprehensive measurement study. In *Proc. IMC*, 2014. 6, 8, 19, 118, 145, 159

[176] Randal Vaughn and Gadi Evron. Dns amplification attacks preliminary release. 2006. 114, 120, 141, 147

[177] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. In *Proc. ICST*, 2010. 74

[178] Yipeng Wang, Xiaochun Yun, M Zubair Shafiq, Liyan Wang, Alex X Liu, Zhibin Zhang, Danfeng Yao, Yongzheng Zhang, and Li Guo. A semantics aware approach to automated reverse engineering unknown protocols. In *Proc. International Conference on Network Protocols*, 2012. 21

[179] Zhongjie Wang, Yue Cao, Zhiyun Qian, Chengyu Song, and Srikanth V. Krishnamurthy. Your state is not mine: a closer look at evading stateful internet censorship. In *Proc. IMC*, 2017. 15, 18, 19, 30, 76, 78, 79, 81, 82, 91, 102, 163

[180] Ralf Weber. Better than Best Practices for DNS Amplification Attacks. `https://archive.nanog.org/sites/default/files/mon_general_weber_defeat_23.pdf`. 114

[181] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proc. CCS*, 2013. 23, 24

[182] Wenfei Wu, Ying Zhang, and Sujata Banerjee. Automatic synthesis of nf models by program analysis. In *Proc. Workshop on Hot Topics in Networks*, 2016. 8, 15, 16

[183] Zhaoyan Xu, Antonio Nappa, Robert Baykov, Guangliang Yang, Juan Caballero, and Guofei Gu. Autoprobe: Towards automatic active malicious server probing using dynamic binary analysis. In *Proc. CCS*, 2014. 17, 18

[184] Lihua Yuan, Jianning Mai, Zhendong Su, Hao Chen, Chen-Nee Chuah, and Prasant Mohapatra. FIREMAN: A toolkit for firewall modeling and analysis. In *Proc. IEEE European Symposium on Security and Privacy 2020*, 2006. 76, 110

[185] Yifei Yuan, Soo-Jin Moon, Sahil Uppal, Limin Jia, and Vyas Sekar. Netsmc: A custom symbolic model checker for stateful network verification. In *Proc. NSDI*, 2020. 26, 73

[186] Michal Zalewski. American fuzzy lop, 2014. 18, 23, 24

[187] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. Verifying software network functions with no verification

expertise. In *Proc. SOSP*, 2019. 15, 16, 17, 168

[188] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina J. Argyraki, and George Candea. A formally verified NAT. In *Proc. SIGCOMM*, 2017. 15, 16, 17, 168

[189] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. Automated verification of customizable middlebox properties with gravel. In *Proc. NSDI*, 2020. 15, 16, 17

[190] M. Zhang, C. Chen, B. Kao, Y. Qamsane, Y. Shao, Y. Lin, E. Shi, S. Mohan, K. Barton, J. Moyne, and Z. M. Mao. Towards automated safety vetting of plc code in real-world plants. In *Proc. IEEE Symposium on Security and Privacy*, 2019. 24, 25