2022

Practical Network Programming Automation

Lei Shi
University of Pennsylvania

Practical Network Programming Automation

Abstract

Network configurations are notoriously hard to write and maintain correctly. It requiresexpertise about the domain to write, frequent and laborious updates, and sometimes formal proof to ensure the absence of certain mistakes. The problem becomes more challenging with the popularity of software-defined network(SDN) in recent years, which aims to give users more flexible control over the network's dynamic behaviors. There has been research on automating the process of configuring the network. However, much of it requires users to learn a specific programming abstraction or interface. Since network operators are a group generally unfamiliar with programming, using these systems may go beyond their abilities. It is also hard to ensure these systems are scalable and accurate enough for real-world usecases. They mostly lack both design considerations to address scalability and accuracy, and also a systematic evaluation of the two metrics in practical scenarios. In this work, we propose a series of approaches to automate network programming. They are based on specifications that are easy and natural to obtain by network operators. We also apply novel program analysis techniques to speed up the process of finding a program that can accurately capture the intention of the specification. We have evaluated our systems on a broad range of benchmarks obtained from real-world data. They have shown ability to finish complex programming tasks within minutes and achieved very high accuracy.

Degree Type

Dissertation

Degree Name

Doctor of Philosophy (PhD)

Graduate Group

Computer and Information Science

First Advisor

Rajeev Alur

Second Advisor

Boon Thau Loo

Keywords

Network, Program Analysis, Program Synthesis

Subject Categories

Computer Sciences

PRACTICAL NETWORK PROGRAMMING AUTOMATION

Lei Shi

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2022

Supervisors of Dissertation

Rajeev Alur
Zisman Family Professor of Computer and Information Science

Boon Thau Loo
RCA Professor of Computer and Information Science

Graduate Group Chairperson

Mayur Naik, Professor of Computer and Information Science

Dissertation Committee

Steve Zdancewic (Chair), Schlein Family President's Distinguished Professor of Computer and Information Science

Mayur Naik, Professor of Computer and Information Science

Vincent Liu, Assistant Professor of Computer and Information Science

Ryan Beckett, Researcher, Microsoft

PRACTICAL NETWORK PROGRAMMING AUTOMATION

ACKNOWLEDGEMENT

I can not thank my parents enough for giving me inspiration throughout my life. It has been the greatest momentum for me to complete the study at Penn.

At the start of the program, I thought my sole purpose is to have more fun of exploring something new. I am fortunate enough to have Prof. Boon Thau Loo and Rajeev Alur as advisors, who unconditionally support every idea I propose, even when it is barely feasible and very unproductive. They helped me find many new angles to see the problems so that my ideas can become realizable. I was also able to avoid innumerable pitfalls thanks to their advice.

Also, my work could not have been successful without help from Caleb Stanford and Yuepeng Wang at the right time. They patiently shared their professional knowledge and experiences when I was exploring unfamiliar topics. It could have cost me many times the effort to figure out how those things work.

Prof. Mayur Naik is in my dissertation committee as well as the committee for WPE II exam. He also gave lecture in a very interesting course. I received suggestions from him in many occasions. Prof. Steve Zdancewic also gave very detailed comments and sometimes challenges to my dissertation. They help make this work closer to perfection.

Finally, as a man who spends most of his life in the apartment, I thank many of my friends for occasionally dragging me out from my cocoon for a trip, or just having some food together. They include Zihui Yuan, Qizhen Zhang, Haoxian Chen, Leshang Chen, and many others. I was able to maintain sanity and survive this long program because of you.

ABSTRACT

PRACTICAL NETWORK PROGRAMMING AUTOMATION

Lei Shi

Rajeev Alur

Boon Thau Loo

Network configurations are notoriously hard to write and maintain correctly. It requires expertise about the domain to write, frequent and laborious updates, and sometimes formal proof to ensure the absence of certain mistakes. The problem becomes more challenging with the popularity of software-defined network(SDN) in recent years, which aims to give users more flexible control over the network's dynamic behaviors.

There has been research on automating the process of configuring the network. However, much of it requires users to learn a specific programming abstraction or interface. Since network operators are a group generally unfamiliar with programming, using these systems may go beyond their abilities.

It is also hard to ensure these systems are scalable and accurate enough for real-world usecases. They mostly lack both design considerations to address scalability and accuracy, and also a systematic evaluation of the two metrics in practical scenarios.

In this work, we propose a series of approaches to automate network programming. They are based on specifications that are easy and natural to obtain by network operators. We also apply novel program analysis techniques to speed up the process of finding a program that can accurately capture the intention of the specification.

We have evaluated our systems on a broad range of benchmarks obtained from real-world data. They have shown ability to finish complex programming tasks within minutes and achieved very high accuracy.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

Introduction

A network is a dynamically evolving distributed system. Many different components need to be programmed or configured in real time to provide new services, accommodate new devices, or counter new security threats. Manually writing and updating such programs is a major burden in these systems' maintenance and is prone to mistakes. Network operators typically do not have the expertise required to do this level of programming. Therefore, automation of this process is of great potential.

In the short term, we could hope to provide an interactive interface for end users to resolve their own demands without having to request network experts' service. In the long term, these techniques could contribute to a self-driving network that can monitor, maintain and evolve itself without human intervention. The complete transition to self-driving networks requires non-trivial research on a number of fields including new demand monitoring, formalization of the update demands, synthesis of the updates and correct installation of the updates, many of which have just emerged in recent years. This work is an early attempt at the update demands formalization and synthesis tasks.

There are existing researches attempting to automatically generate network configurations such as ACL [1] and routing policies [2; 3; 4; 5]. However, two types of shortcomings make them not very usable in practice. First, they rely on extra user efforts to provide acceptable specifications. Many of them have a programming abstraction or interaction model specific to the tool, which require users with some programming skill spending some time to get used to. Others require users to provide thorough, formal specifications describing the network program's behavior. Both of the above are not commonly available. Second, they lack evaluation of the tools' scalability and accuracy when handling data volume or program size close to production environment. Most of them have only shown case studies over demonstrative examples.

In this work, we show the feasibility of practical network programming automation by synthesizing or repairing network programs from specifications that can be easily collected by network operators. We also apply novel program analysis techniques to ensure the results can be generated efficiently with high accuracy even on benchmarks derived from real-world data.

Broadly speaking, there are two main categories of network programs: domain specific rules and general purpose control programs.

Domain specific rules are mostly deployed in the data plane to decide routing of packets. They could also exist in the control plane as meta rules. Individual rules are frequently updated. Since they are usually writen in compact domain specific languages(DSL), it is possible to derive entire new rules by search-based synthesis. We propose that large amount of labelled traffic data can specify in enough detail the behavior of this type of programs. As an example, we demonstrate how programs in NetQRE, a DSL designed for session-level traffic monitoring, can be learned from labelled network traces and used as traffic classifiers. Mainly two kinds of optimizations are apply in this system: partial program evaluation to eliminate impossible search spaces early, and divide-and-conquer, to speed up candidate program verification over large training data.

Although control plane programs are less likely to be replaced frequently once the applications are installed, they still need to be updated from time to time to fix bugs or add new functions. Specifications for this task could be given in two parts: counter-examples demonstrating the bug or expected new behavior, and logical constraints describing properties that shouldn't be broken during the update. The former could be manually or automatically collected in a separate data plane verification procedure. The latter is less likely to change and could be requested along with the application. It is also possible to use common network-wide safety properties. Our current work can support repairing control plane programs based on examples. It utilizes a modular semantic encoding of the program solvable by an SMT solver to localize the bug location, and synthesizes a patch from the local specification derived during the localization phase.

We have implemented and evaluated the forementioned automatic programming systems for both network program categories.

1.1. Contributions

This thesis proposes two methods that are user-friendly, accurate and fast for the purpose of automatically generating and updating network programs.

To generate network domain specific rules, we show how application-layer classification rules can be synthesized from labelled network traces. Users can provide 1) negative traces without any target flow and 2) positive traces that contain the target traffic with up to a small portion of irrelevant flows for our system to learn a classification rule, which can be used later to tell if a network trace contains any target application's traffic. We devise partial program execution and merge search techniques to tackle the large program search space and large training set problems, respectively. We also explore a noise tolerance algorithm to make the learned program applicable even when then target application only make up a small portion of the traffic.

To update general purpose controller programs, we show how control plane applications in Java language, a popular choice for control plane implementation, can be automatically repaired by unit tests indicating the expected behaviors of the network. Users can provide a few input/output examples as the specification. Our system is able to localize and repair a single point of bug with up to 3 lines of changes so that the repaired program strictly complies with the specification. We devise a modular semantic encoding method so that the bug location in the program can be efficiently solved by an SMT solver. We also use the local specification derived in the procedure to speedup the synthesis of the patch.

In both scenarios, we implement a prototype of the system and evaluate it by real-world benchmarks. The results show the feasibility of the methods. They also demonstrate state-of-the-art performances in terms of both accuracy and time consumption.

These contributions are based on two published papers [6; 7] and one paper in submission about domain specific rule synthesis and application.

CHAPTER 2

Background and Related Work

2.1. Network Model

A network can be intuitively viewed as a directed graph, where vertices are hosts and switches, while edges are links between them. Switches decide which path packets should take. When a packet arrives at a switch, it typically looks up a series of tables for rules on how to update and forward the packet.

Under the framework of softwared defined network(SDN), the portion executing the table lookup and packet updating and forwarding at each switch is called the data plane, which is usually composed of simple logic. In traditional networks, the data plane configurations are manually composed and updated, which is laborious and time-consuming. SDN allows dynamic updates to the configurations managed by software. When the data plane encounters a packet of interest, such as one without a corresponding entry in some tables, it will report the packet to a logically centralized control plane through a pre-defined protocol (e.g. OpenFlow). The control plane then determines how to update the tables of that specific data plane. The control plane can be stateful and use complex logic written in general purpose languages to decide network-wide configurations.

Apart from basic routing tasks, a network may support more complex network functions such as firewall, load balancing, deep packet inspection, etc. A network function can also be viewed as a node in the forementioned graph model. But the specific implementation is orthogonal to SDN.

Prior work has proposed a number of domain specific languages to formally define the behavior of network components such as NetKAT[8], OF-DPA[9], Genesis[4] and Merlin[10].

2.2. Program Synthesis

Program synthesis is the task of automatically generating a program that satisfies a given specification. Popoular formalizations of this general task include syntax guided synthesis(SyGuS) and programming by example(PBE). SyGuS requires that both the specification and the program's grammar can be expressed as logical constraints based on a theory with efficient decision procedures of its satisfiability, in a hope that a good algorithm could solve a wide range of tasks encoded in this form [11]. Programming by example broadly includes any task with input/output examples as the specification for the program. It has been applied to a number of real-world tasks including automatic spreadsheet content transformation [12; 13], SQL queries generation[14], as well as tutorial tools[15].

In terms of program generation techniques, search-based synthesis is the most common form, where programs are enumerated by applying production rules in the grammar to the abstract syntax tree(AST). The procedure starts from a starting symbol or a program sketch with holes and gradually expands them into complete candidate programs. The candidate programs are typically checked against the specification in increasing order of complexity. Various techniques have been proposed to speed up this search, such as utilizing type system to do early pruning [16; 17], or predicting more possible production rule to apply with machine learning [18].

There are also specialized application of program synthesis in the network domain. For data plane configurations, EasyACL [1] aims at synthesis of access control lists(ACL) from natural language descriptions. Soumya, et al. [19] instead derive ACL implementations from network topology and input security policy specifications. NetGen [2], NetComplete [3] and Genesis [4] synthesize data plane routing configurations based on SMT solvers given policy specifications in regular expressions or customized policy languages. NetEgg [5] instead takes examples provided by user to generate routing configurations in an interactive way. There are also attempts at synthesizing the control plane programs. For example, Avenir [20] assumes that users can provide an abstract implementation of control plane operations and synthesizes translation rules from these abstract implementations to concrete configurations in a variety of data plane devices in a counter-example driven manner.

2.3. Automated Program Repairing

Program repairing can be viewed as a special kind of program synthesis where, along with the specification of the entire program, there is also an almost correct program given as a hint. The goal is to find the correct version of the program that can satisfy the specification.

Usually, there are two separate steps in program repairing: a localization step that finds the proper location of the bug, and a fixing step that synthesizes a patch at the bug location to make the program correct. There is research on both tracks.

On localization, a typical method uses heuristic-based scoring algorithms to evaluate most possible bug locations. It usually involves comparing statistics of execution traces from correct and incorrect input/output examples over the buggy program [21; 22; 23; 24; 25; 26; 27]. This method is quick in analyzing large programs, but usually low in accuracy, and it requires a number of test cases to work. Alternatively, prior work uses semantic encoding and theorem provers to strictly infer the bug location [28; 29; 30]. They can have higher accuracy in localizing the bug with few examples, but do not scale well, since they require solving some kind of NP-Hard problems with regard to the program's size during the process.

Patch synthesis during program repair has a major difference from general program synthesis. The specification is not give locally for the patch, but for the entire program instead. Although it is possible to run the entire repaired program for checking, it can become highly inefficient when the program's size is much larger than the patch. There are attempts to tackle this challenge by inferring the local specification at the bug location, for example by dynamic symbolic execution [21].

Program repair in networks has been highly domain specific so far. Prior work about autorepair [31; 32] relies on using Datalog to capture the operational semantics of the target language to be repaired. The repair techniques work for domain specific languages (e.g. Datalog or Ruby on Rails) with simple structure. Similarly, Hojjat, et al. [33] propose a framework based on horn clause repair problem to help network operators fix buggy configurations. No prior work has targeted more widely used general purpose languages in writing SDN programs such as Java or Python.

2.4. Machine Learning Alternatives

There are also methods that use machine learning models to entirely take up the rule of a network function, instead of synthesizing the corresponding program. The most notable efforts focuses on security applications such as intrusion detection.

There are applications of both supervised and unsupervised learning approaches. Supervised learning systems such as Kitsune [34] can learn from labelled traffic data a classification model that will raise an alert when similar traffic is observed next time. Such models are typically of higher accuracy. Unsupervised learning systems are useful for recognizing outliers and other types of "abnormal" flows [35; 36; 37]. That is to say, they can differentiate unknown types of traffic from the known.

There are state-of-the-art point solutions focusing on specific scenarios rather than general-purpose security threats. For example, PrivateEye focuses on detecting privacy breaches in the cloud[38]. RFDIDS solves intrusion detection challenges unique to power systems[39].

Generally speaking, machine learning methods enjoy the benefit that there are mature algorithms to efficiently learn from big data and achieve relatively high accuracies compared to most existing automatic network configuration synthesis systems. But they also fall short of interpretability compared to program synthesis. This shortcoming makes them hard to use in scenarios that require verification or involve frequent human interaction, such as many security applications [40].

CHAPTER 3

Overview

To understand how our tools can help with automation of programming in networks, let us imagine two cases where the operators may want to program the network.

In the first case, they may find direct need of new network monitoring rules, for example, to make policies for a new type of user behavior, or to raise an alert when a new type of attack occurs. Both demands essentially requires classification between the traffic of interest and the "background". They can demonstrate this expected behavior by collecting traffic traces going through the target switch at normal time and also traces of the interesting traffic. Usually, the operators will need to wait for an expert to analyze the latter and manually write a rule to match it, which takes both time and money. Our first tool, Sharingan, can instead take the two kind of traces as input and automatically generate a classification rule in NetQRE language that matches only the interesting trace. This rule can be directly installed to trigger a following action. It is also possible to interpret it into a natural language sentence by simple text replacement, so as to hint the operators on the nature of the traffic, as well as necessary actions to be taken. Even if they eventually decide to use a human expert, the interpretable learning result can still greatly boost the manual analysis.

In the second case, the operators may find the network showing undesired behaviors after deployment. By direct observation or output from a separate data plane verification tool, they may be able to get a counter example indicating an incorrect rule install by the control plane. In this case, they would want to fix the problem in the control plane based on the examples they've collected. But on the other hand, they do not want this fix to break existing safety properties. This is usually only possible when the developer of the control plane is present. Our tool, Orion, in its complete shape, will be able to automatically fix bugs in the control plane program given input/output examples as well as network-wide safety properties in first order logic, thus resolving the emergency in a reliable way. In the current version, input/output examples are supported.

CHAPTER 4

Classification Rule Synthesis

In this section, we introduce our solution to the first type of programming task. We designed and implemented Sharingan, a system that can help users automatically synthesize network traffic classification programs in NetQRE language based on labelled traffic data.

4.1. Design Goals

To achieve high usability of the tool, we guaranteed the following three advantages of Sharingan

Requires minimal feature engineering: NetQRE [41] is an expressive language that allows succinct description of a wide range of tasks ranging from detecting security attacks to enforcing application-layer network management policies. Sharingan can synthesize any network task on raw traffic expressible as a NetQRE program, without any additional feature engineering. This is an improvement over systems based on manually extracted feature vectors. Also, one outstanding feature of search-based program synthesis is that the only a priori knowledge it needs is information about the language itself. No task-specific heuristics are required.

Efficient implementation: The NetQRE program synthesized by Sharingan can be compiled, as has been shown in prior work [41], to efficient low-level implementations that can be integrated into routers and other network devices. On the other hand, traditional statistical classifiers are not directly usable or executable in network filtering systems.

Easy to decipher and edit: Sharingan generates NetQRE programs that can be read and edited. Since they are generic executable programs with high expressiveness, the patterns in the program reveal the stateful protocol structure that is used for the classification, which blackbox statistical models, packet-level regular expressions and feature vectors have difficulty describing. The programs are also amenable to calibration by a network operator, for example, to mix in local policies or debug.

Noise-tolerant: An application-level pattern may span through a number of packets and flows. But when they are mixed with other background traffic, users are unlikely to know a priori which flows belong to the same event or application. Sharingan is enhanced by a method based on an improved learning stage and a combination of traffic sample sizes during monitoring to achieve highly accurate recognition of the pattern even when it only occupies a fraction of the total traffic.

4.2. Overview

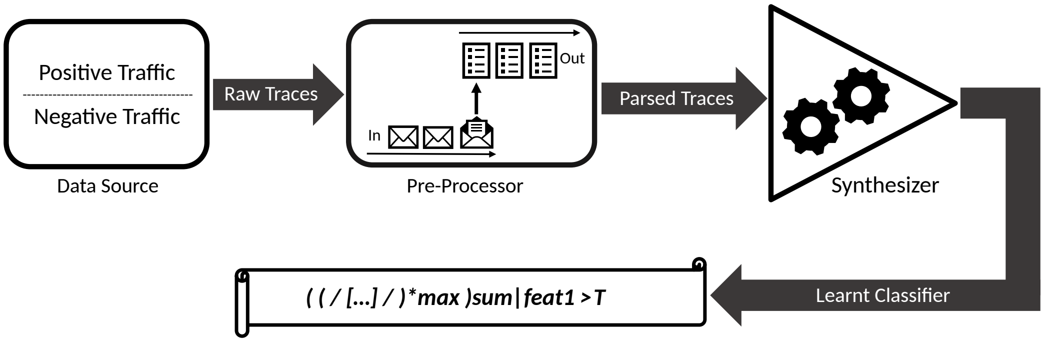


Figure 1: System Overview

As is shown in Figure 1, Sharingan's workflow is largely similar to a statistical supervised learning system, although the underlying mechanism is different. Sharingan takes labeled positive and negative network traces as input and outputs a classifier that can classify any new incoming trace. To preserve most of the information from input data and minimize the need for feature engineering, Sharingan considers three kinds of properties in a network trace: (1) all available packet-level header fields, (2) position information of each packet within the sequence, and (3) time information associated with each packet.

Specifically, Sharingan represents a network trace as a stream of feature vectors: $S = v_0, v_1, v_2, \ldots$. Each vector represents a packet. Vectors are listed in timestamp order. Contents of the vector are parsed field values of that packet. For example, we can define

$v[0] = ip.src$, $v[1] = tcp.sport$, $v[2] = ip.dst, \ldots$.

Depending on the information available, different sets of fields can be used to represent a packet. By default, we extract all header fields at the TCP/IP level, which are available in almost all packets. To make use of the timestamp information, we also append time interval since the previous packet in the same flow to a packet's feature vector. Further feature selection is not necessary for Sharingan.

The output classifier is a NetQRE program p that takes in a stream of feature vectors. Instead of giving a probability score that the data point is positive, it outputs an integer that quantifies the matching of the stream and the pattern. The program includes a learnt threshold T. Sharingan aims to ensure that p's outputs for positive and negative traces fall on different sides of the threshold T. Comparing p's output for a data point with T generates a label. It is possible to translate p and T into executable rules using a compilation step.

Given the above usage model, a network operator can use Sharingan to generate a NetQRE program trained to distinguish normal and suspected abnormal traffic generated from unsupervised learning systems. The synthesized programs themselves, as we will later show, form the basis for deciphering each unknown trace. Consequently, traces whose patterns look interesting can be subjected to a detailed manual analysis by the network operator. Moreover, the generated NetQRE programs can be further refined and compiled into filtering system's rules.

4.3. Background on NetQRE

NetQRE [41] is a high-level declarative language for querying network traffic. Streams of tokenized packets are matched against regular expressions and aggregated by multiple types of quantitative aggregators. The NetQRE language is defined by the BNF grammar in Figure 2.

As an example, if we want to find out if any single source is sending more than 100 TCP

```
<classifier>::= <program> > <value>
<program> ::= <group-by>
<group-by> ::= (<group-by>)<op>|<feats>
            | <qre>
<qre> ::= (<qre> <qre>)<op>
          | (<qre>)*<op>
          | <unit>
<unit> ::= /<re>/
<re> ::= <re> <re>
         | (<re>)*
         | <pred>
         | _
<pred> ::= <pred> && <pred>
           | <pred> || <pred>
           | [<feat> == <value>]
           | [<feat> >= <value>]
           | [<feat> <= <value>]
           | [<feat> -> <prefix>]
<feats> ::= <feat>
            | <feats>, <feat>
<feat>  ::= 0 | 1 | 2 | ......
<op>    ::= max | min | sum
```

Figure 2: NetQRE Grammar

packets, the following classifier based on a NetQRE program describes the desired classifier:

```
( ( / [ip.type = TCP] / )*sum )max|ip.src_ip > 100
```

At the top level, there are two parts of the classifier. A processing program on the left that maps a network trace to an output number, and a threshold against which this value is compared on the right. They together form the classifier. Inputs fall into different classes based on the results of the comparison.

Group-by expression (`<group-by>`) splits the trace into sub-flows based on the value of the specified field (source IP address in this example):

```
( ........... )max|ip.src_ip
```

Packets sharing the same value in the field will be assigned to the same sub-flow. Sub-flows are processed individually, and the outputs of which are aggregated according to the aggregation operator (`<op>`) (maximum in this example).

In each sub-flow, we want to count the number of TCP packets. This can be broken down into three operations: (1) specifying a pattern that a single packet is a TCP packet, (2) specifying that this pattern repeats arbitrary number of times, and (3) adding 1 to a counter each time this pattern is matched.

(1) is achieved by a *plain regular expression* involving *predicates*. A predicate describes properties of a packet that can match or mismatch one packet in the trace. Four types of properties frequently used in networks can be described:

1. It equals a value. For example: `[tcp.syn == 1]`
2. It is not less than a value. For example: `[ip.len >= 200]`
3. It is not greater a value. For example: `[tcp.seq <= 15]`
4. It matches a prefix. For example: `[ip.src_ip -> 192.168]`

Predicates combined by concatenation and Kleene-star form a plain regular expression, which matches a network trace considered as a string of packets.

A *unit expression* indicates that a plain regular expression should be viewed as atomic for quantitative aggregation (in this case a single TCP packet):

```
/ [ip.type = TCP] /
```

It either matches a substring of the trace and outputs the value 1, or does not match.

To achieve (2) and (3), we need a construct to both connect the regular patterns to match the entire flow and also aggregate outputs bottom up from units at the same time. We call it *quantitative regular expression* (`<qre>`). In this example, we use the iteration operator:

```
( / [ip.type = TCP] / )⋆sum
```

It matches exactly like the Kleene-star operator, and at the same time, for each repetition of the sub-pattern, the sub-expression's output is aggregated by the aggregation operator. In this case, the sum is taken, which acts as a counter for the number of TCP packets. The aggregation result for this expression will in turn be returned as an output for higher-level aggregations.

The language also supports the concatenation operator:

```
(<qre> <qre>)<op>
```

which works analogous to concatenation for regular matching. It aggregates the quantity by applying the `<op>` on the outputs of two sub-expressions that match the prefix and suffix.

The disjunction expression is not supported at `<qre>` level in order to avoid ambiguity, which requires any accepted sequence to have a unique matching path. Such restriction may lead to lack of efficient expression for certain tasks. But it's in exchange of a more efficient implementation.

In addition to this core language, there is a specialization for the synthesis purpose. We observe that comparing a field with values that do not appear in any of the given examples is expensive but will not produce any meaningful information. For example, if a feature has value space $\{1, 3, 12, 15\}$ in the dataset, then two predicates that assert the feature is greater than number 3 and 4 respectively will produce exactly the same truth values for all packets in the dataset. We can not possibly tell which one is better merely from the dataset. Therefore we use the relative position in the examples' value space instead of a specific value, for example, 50% instead of 3 in value space $\{1, 3, 12, 15\}$. In this manner, the search is more efficient than searching in the entire integer space.

4.4. Synthesis Algorithm

Given a set of positive and negative examples E_p and E_n, respectively, the goal of our synthesis algorithm is to derive a NetQRE program p_f and a threshold T that differentiates E_p apart from E_n. We start with notations to be used in this section:

Notation. p and q denote individual programs, and P and Q denote sets of programs. $p_1 \rightarrow p_2$ denotes it is possible to mutate p_1 following production rules in NetQRE's grammar to get p_2. The relation $\rightarrow$ is transitive. We assume the starting symbol is always `<program>`.

$p(x)$ denotes program p's output on input x, where x is a sequence of packets and $p(x)$ is a numerical value. If p is an incomplete program, i.e., if p contains some non-terminals, then $p(x) = \{q(x) \,|\, p \to q\}$ is a set of numerical values, containing x's output through all possible programs p can mutate into. Since we do not use any multiplicative operator for aggregation, values in this set can not be greater than the input sequence's length. Therefore the set is always of limited size. We define $p(x).max$ to be the maximum value in this set. Similarly, $p(x).min$ is the minimum value.

The synthesis goal can be formally defined as: $\forall e \in E_p, p_f(e) > T$ and $\forall e \in E_n, p_f(e) < T$.

4.4.1. Overview

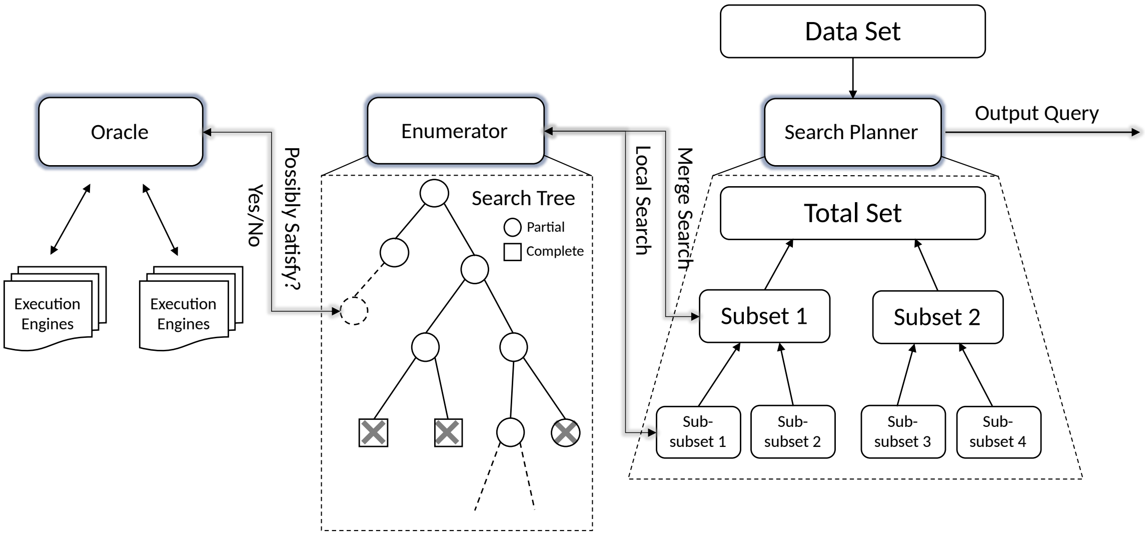


Figure 3: Synthesizer Overview

Our design needs to address two key challenges. First, NetQRE's rich grammar allows a large possible program space and many possible thresholds for search. Second, the need to check each possible program against a large data set collected from network monitoring tasks poses scalability challenge to the synthesis.

We propose two techniques for addressing these challenges: *partial execution* (Section 4.4.2) and *merge search* (Section 4.4.3). Figure 3 shows an overview of the synthesizer.

The top-level component is the *search planner*, that assigns search tasks over subsets of the entire training data to the enumerator in a divide-and-conquer manner. Each such task is a search-based synthesis instance, where the *enumerator* enumerates all possible programs starting from s_0, expanded using the productions in NetQRE grammar, until one that can distinguish the assigned subset of E_p and E_n is found, or until a complexity limit is reached.

The *enumerator* optimizes for the first challenge by querying the distributed *oracle* about each partial program's feasibility and doing pruning early. The *oracle* evaluates partial programs using *partial execution*. The *search planner* optimizes for the second challenge by merging search results from subsets of the large training data, so as to save unnecessary checking, which we call the *merge search* strategy.

We next explain each technique in detail in the rest of this section.

4.4.2. Partial Execution

A *partial program* is an incomplete program with non-terminals. Similar to prior work making overestimation on regular expressions and imperative languages for early pruning in the search process [42; 15; 43], we want to evaluate a partial NetQRE program for the feasibility of all possible completions of it, so as to decide early if any of them can serve as a proper classifier for E_p and E_n.

This process includes three main steps: (1) finding an equivalent completion $\hat{p}$ of a partial program p so that evaluating $\hat{p}$ on any input x is equivalent to evaluating the combination of all possible completions of p on x, (2) efficiently evaluating $\hat{p}(x)$, (3) deciding whether to discard p based on the evaluation result.

Equivalent Completion: Recall that we define $p(x)$ of a partial program p to be the union of all $q(x)$ such that $p \to q$. Since we mainly care about outputs of positive and negative examples on different sides of a threshold, the essential information is the upper and lower bounds for $p(x)$. Therefore, the criterion for finding an equivalent completion is the bounds of $\hat{p}(x)$ should include $p(x)$ for any input x.

Many non-terminals have a straightforward equivalent completion. We replace (1) any uncertain numerical value with the largest or smallest possible value depending on the context, (2) any unknown predicate with *unknown*, (3) any unknown regular expression with `_*` and (4) any unknown quantitative regular expression with `(/_ _*/)*sum`. The details of these four cases are briefly described below.

The syntax tree enumeration phase identifies proper numerical values by binary search. For example, to find 62.5%, the range $[50\%, 100\%]$ is first explored. If it works, this is refined to $[50\%, 75\%]$, and eventually 62.5%. If there is an incomplete predicate [*feat1* >= $[50\%, 100\%]$], it can be completed by taking the smallest possible value 50% in the unknown part and turned into $\hat{p}$: [*feat1* >= 50%]. If the operator is <=, we take the largest possible value instead.

We define *unknown* as a Boolean state that is possibly true and possibly false, which indicates the uncertain status of a partial predicate with incomplete parts other than numerical values. Wherever a *true* is required, *unknown* also works, since it already implies the possibility of matching. The calculation rule for *unknown* is:

```
T = true, F = false, U = unknown
T && U = U  T || U = T
F && U = F  F || U = U
U && U = U  U || U = U
```

The remaining two cases are both within the grammar of NetQRE. `_*` matches an arbitrary number of arbitrary packets, therefore containing the matching results of all possible regular expressions. `(/_ _*/)*sum` matches an arbitrary number of packets and outputs 0 (in case no packet is matched) or $[1, n]$ where n is the number of packets matched. Since a unit expression outputs a constant 1 and there is no multiplicative aggregator, this is exactly the range of all possible outputs of all possible expansions.

There are some non-terminals that cannot be completed in this way, such as flow split and aggregation operators. We put a complexity penalty over these non-terminals if they are not expanded, therefore encouraging expanding them earlier to allow partial execution. Since flow split is not frequently used, and aggregation operator does not have many expansion choices, they are not a major source of overhead.

Computing Ambiguity: Notice that although the finished program still largely follows NetQRE's grammar, its semantics is different, because this process introduces ambiguity into the program. That is, for a given input x, the completed program $\hat{p}$ can have different matching strategies and different outputs. In core NetQRE, such a case would have produced a *conflict* output. But in partial execution, our goal, and also the main challenge, is to properly estimate the set of all possible outputs for the possible completions.

We demonstrate this ambiguity problem by an example. Suppose there are two predicates A and B defined as:

```
A: [ip.type == TCP]
B: [ip.type == UDP]
```

We can write a NetQRE program based on them:

```
( ( /AA/ )*sum ( /B/ )*sum )max
```

which describes a trace with an even number of TCP packets followed by an arbitrary number of UDP packets. It counts the number of TCP packet pairs and the number of UDP packets, and outputs the larger number. Since A and B are mutually exclusive, the expression is not ambiguous.

Suppose there is a trace of packets $CCCCD$, where C's are some TCP packets and D is a UDP packet. Let us first consider how it is processed by the unambiguous NetQRE program. The execution can be illustrated by the flowchart in Figure 4. A trace first goes to the left cycle that consumes pairs of TCP packets, and then to the right for UDP packets. The order of packets matched is also shown in the figure as subscripts.

In order to compute the numerical output, the program needs to maintain aggregation states during the matching. For example, when it processes the third packet C_3, the execution is at the middle of the second loop of the left cycle. The accumulated sum of this iteration

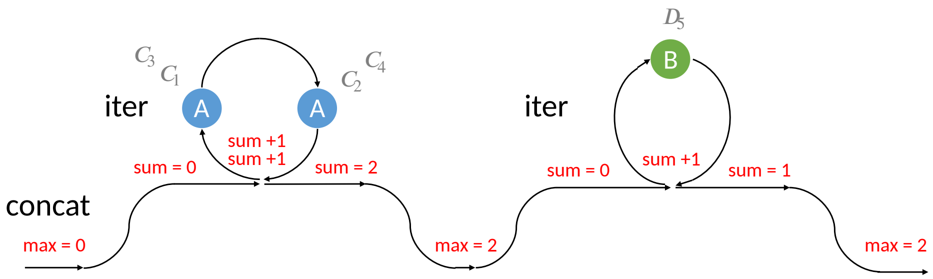


Figure 4: Illustration of an unambiguous program

cycle is currently 1, and maximum value of the outermost concatenation is currently the initial value 0. These are core states for computing ambiguous outputs. Eventually, the outermost aggregation result 2 is taken as the final output.

Now let us look at the synthesis steps. Suppose that during the search, we have explored part of this program and the predicate B is not yet known:

```
( ( /AA/ )*sum ( /<pred>/ )*sum )max
```

To evaluate this partial program, we complete it by replacing the missing predicate with $unknown$, denoted as _ below, which matches any packet:

```
( ( /AA/ )*sum ( /_/ )*sum )max
```

As a result, the program has become ambiguous. To evaluate it on the input trace $CCCCD$, there are three different correct matching strategies: matching the first iteration of two TCP packets with 0, 2, or 4 number of C packets respectively, and matching the rest of the trace with the iteration of wildcard. They produce three outputs: 5, 3, 2. The set $\{2, 3, 5\}$ is an optimal result. But in practice, since we will compare with a specific threshold, only the upper-bound and lower-bound of all possible outputs is needed. Therefore we want the output to be the interval $[2, 5]$.

A strawman method is simply to enumerate all possible matching strategies and take the union of all their outputs. The problem is that there can be exponentially many distinct matching strategies with respect to the length of the network trace, leading to unacceptable synthesis time.

We solve this problem by an approximation: we merge ”close” matching strategies. Two strategies are defined to be ”close” if at some step of their matching process (1) they have matched the same number of packets in the trace and (2) the last predicate they have matched is exactly the same. We explore all matching strategies simultaneously and do a merging whenever two strategies can be identified to be close.

We now describe how this merging works. Again, we use the program and trace above as an example. We inspect the two matching strategies that match the first iteration of two TCP packets against 0 and 2 packets of the trace respectively. After matching the third packet C against the wildcard _, they can be identified to be close. Their matching and aggregation states before this point are shown in Figures 5 and 6.

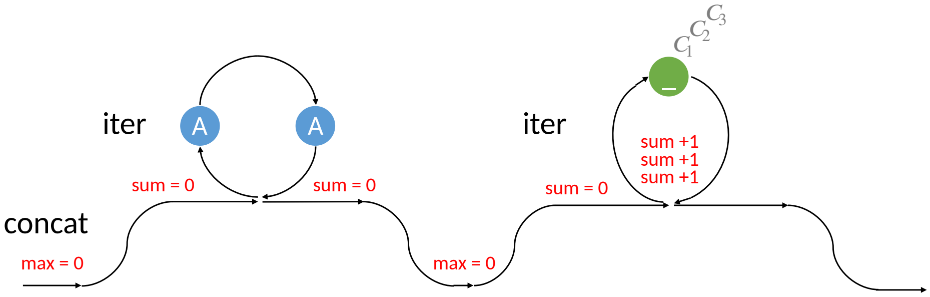


Figure 5: Illustration of the first 3 steps of strategy one

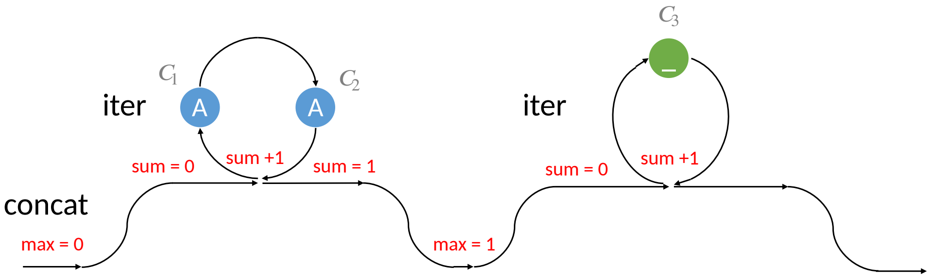


Figure 6: Illustration of the first 3 steps of strategy two

We observe that the corresponding executions have maintained different aggregation states. For strategy one, the current maximum for outermost concatenation is 0 and the current sum for the second iteration is 3. For strategy two, the current maximum for the concatenation is 1 while the current sum for the second iteration is 1. Similar to the way we handle final outputs, we merge these aggregation states by recording the range of all possible values from merged strategies. In this specific example, the two values will result in intervals $[0, 1]$ and $[1, 3]$ respectively. For the remaining part of the matching, the aggregations will be done on the intervals, which is illustrated in Figure 7. Eventually, the final aggregation result $[3, 5]$ is the estimated output for this merged strategy.

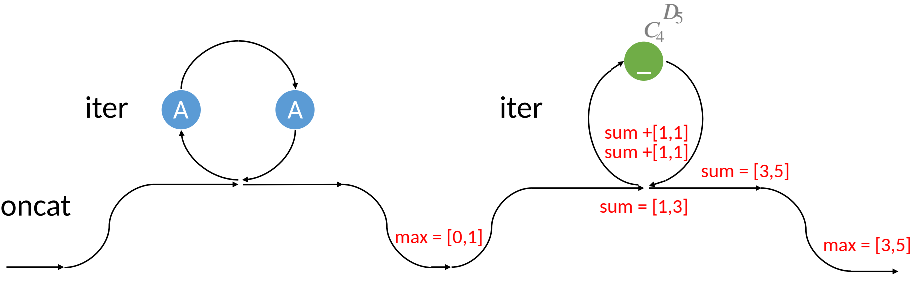


Figure 7: Illustration of the last 2 steps of merged strategy one & two

The regular matching result by this approximation is correct, since at any step, a matching strategy's remaining part is determined by matched packets' length and the current predicate. It can also be proven by the properties of interval arithmetic that the aggregation result strictly contains the true output range. Or more formally, $\hat{p}(x).min \leq p(x).min \leq p(x).max \leq \hat{p}(x).max$. Therefore $p(x)$ can be approximated by $\hat{p}(x)$.

Intuitively, the proposed evaluation scheme works well because we only care about the boundary of outputs, which are represented by intervals as the abstract data type. We implement the execution and approximation process by the Data Transducer model proposed by [44], which consumes a small constant memory and liner time to the input trace's length given a specific program.

Make Decision: To make a decision regarding a partial program p, let q be a complete program and assume there is only one pair of examples e_p and e_n. For q to accept e_p and e_n, there must be a threshold T such that $q(e_n).max < T < q(e_p).min$. Therefore, given a pair of examples e_p and e_n, a program q is correct if and only if $q(e_n).max < q(e_p).min$. When this holds, any value between $q(e_n).max$ and $q(e_p).min$ can be used as the threshold.

Lemma 1: There exists a correct program q such that $p \to q$ only if $\hat{p}(e_n).min < \hat{p}(e_p).max$

Lemma 2: If $\hat{p}(e_n).max < \hat{p}(e_p).min$ then any program q such that $p \rightarrow q$ is correct.

Proof Sketch of Lemma 1:

There exists correct $q \Rightarrow q(e_n).max \leq q(e_p).min \Rightarrow$

$\hat{p}(e_n).min \leq q(e_n).min \leq q(e_n).max \leq q(e_p).min \leq q(e_p).max \leq \hat{p}(e_p).max$

Proof Sketch of Lemma 2:

$\hat{p}(e_n).max \leq \hat{p}(e_p).min \Rightarrow$

For any q, $q(e_n).max \leq \hat{p}(e_n).max \leq \hat{p}(e_p).min \leq q(e_p).min \Rightarrow q$ is correct.

From Lemma 1, we can decide if p must be rejected. From Lemma 2, we can decide if p must be accepted. These criteria can be extended to more than 1 pair of examples. Figures 8 and 9 show two intuitive examples for explanations of the decision making process. (but do not necessarily represent properties of real data sets). Each vertical bar represents the output range of the corresponding data point produced by the program under investigation.

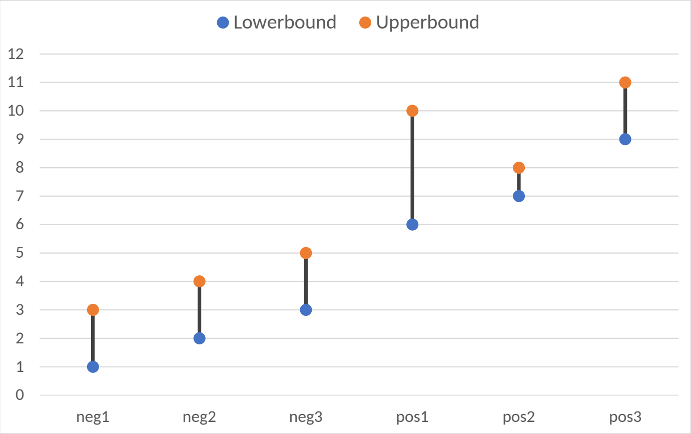


Figure 8: A correct program found. No negative output can ever be greater than any positive output. 5.5 can be used as a threshold

4.4.3. Merge Search

In the rest of this subsection, we describe three heuristics for scaling up synthesis to large data sets, namely *divide and conquer*, *simulated annealing*, and *parallel processing*. We call the combination of these the *merge search* technique.

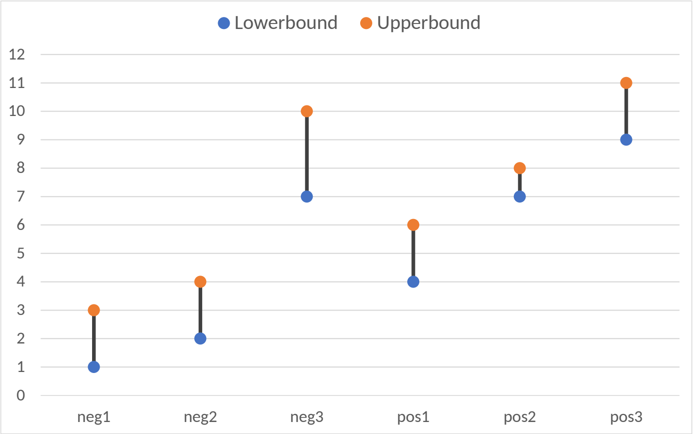


Figure 9: A bad program. pos 1 can never be greater than neg 3.

Divide and Conquer. Enumerating and verifying programs on large data sets is expensive. Our core strategy to improve performance is to learn patterns on small subsets and merge them into a global pattern with low overhead.

It is based on two observations: First, the pattern of the entire data set is usually shaped by a few extreme data points. Looking at these extreme data points locally is enough to figure out critical properties of the global pattern. Second, conflicts in local patterns are mostly describing different aspects of a same target rather than fundamental differences, thus can be resolved by simple merge operations such as disjunction, truncation or concatenation.

This divide and conquer strategy is captured in the following algorithm:

```
def d&c(dataset)
    if dataset.size > threshold
        subsetL,subsetR = split(dataset)
        candidateL = d&c(subsetL)
        candidateR = d&c(subsetR)
        return merge(dataset, candidateL, candidateR)
    else
        return synthesize(dataset, s0)
```

The "split" step corresponds to evenly splitting positive and negative examples. Then sub-patterns are synthesized on smaller subsets. If the numbers of positive and negative examples are significantly different, the lesser side will be duplicated to match the other side. The conquer, or "merge" step requires synthesizing the pattern again on the combined dataset. But sub-patterns are reused in two ways to speedup this search.

First, if we see a sub-pattern as an AST, then its low-level sub-trees up to certain depth threshold are added to the syntax as a new production option for the corresponding non-terminal at the sub-tree's root. They can then serve as shortcuts for likely building blocks. Second, the sub-patterns' skeletons left after removing these sub-trees are used as seeds for higher-level searches, which serve as shortcuts for likely overall structures. Both are given complexity rewards to encourage the reuse.

In practice, many search results can be directly reused from cached results generated from previous tasks on similar subsets. This optimization can further reduce the synthesis time.

Simulated Annealing When searching for local patterns at lower levels, we require the Enumerator to find not 1 but t candidate patterns for each subset. Such searches are fast for smaller data sets and can cover a wider range of possible patterns. As the search goes to higher levels for larger data sets, we discard the least accurate local patterns and also reduce t. The search will focus on refining the currently optimal global pattern. This idea is based on traditional simulated annealing algorithms and helps to improve the synthesizer's performance in many cases.

Parallelization. Most steps in the synthesis process are inherently parallelizable. They include (1) doing synthesis on different subsets of data, (2) exploring different programs in the enumeration, (3) verifying different programs found so far, (4) executing a program on different data points during the verification.

We focus less on optimizing (1) and (2) since they are not the performance bottlenecks. We instead focus on parallelizing (3) and (4) over multiple cores. In our implementation, using 5 machines with 32 cores each, we devote one thread each to run task (1) and (2) on one machine, 64 threads on the same machine to run task (3), and 512 threads distributed over the remaining four machines to run task (4). The distributed version is approximately two orders of magnitude faster than the single-threaded version for complex tasks. Given more computing power, a proportional speedup can be expected.

4.5. Noise Tolerance

In this section, we investigate how to apply the learned NetQRE to real-world network traffic, including the difficulties that may be faced and our proposed solutions.

4.5.1. Overview

NetQRE programs describe application-level patterns that may span through a number of packets and flows. While how the program is used depends on the specific scenario which we will discuss in the evaluation, in a typical network monitoring task, there can be two major challenges in using the NetQRE program.

First, unlike in the training data, where we expect the user to provide pure positive and negative traces as the specification, real network traffic to be classified can be noisy, and we need the classifier to recognize the target traffic even when it is mixed in irrelevant background traffic.

Second, while raising the classifier's sensitivity to positive traffic in an environment with rich negative noises, we need the false positive rate to stay low.

More precisely, we define a network trace to be *pure* with regard to an application if it consists of only flows generated by this application, possibly by both the server and the client sides. The trace is *noisy* if it also contains flows from other sources. These other flows are *noise.* Suppose the total number of the target application's flows is a, and the total number of noise flows is b, *noise ratio* is computed by $b/(a+b)$.

Different from the classic task where the aim is to label each individual flow or packet, we investigate the recognition problem: we want to tell if a continuous traffic contains any of the application's traffic. If yes, it should be labelled positive. Otherwise, it should be labeled negative. If the traffic contains a flows of the target application in total, we call a the *duration* of the application. In practice, we may want to split the traffic into a series of traces and apply the learned NetQRE program on each individual trace to get a label. But the eventual result that is desired and used to decide the accuracy is still the label for the entire traffic, which we will describe how to decide later.

We address the two challenges above respectively with improved training data and monitoring strategies. Eventually, we will reduce the recognition accuracy problem to the requirement of a minimal duration of the target traffic. Namely, we want to transform the problem into the following format: if the target application's duration is at least l under noise ratio α, we can guarantee accurate recognition of it, and we want to minimize l. In the following subsections, we introduce each strategy in detail and explain how a minimal duration requirement could be achieve at last.

4.5.2. Handling Noise

Notice that NetQRE is a quantitative query language based on aggregation through a contiguous sequence of packets. The more target flows this sequence contains, the higher we can expect the output quantity to become. Although `max` and `min` operators existing together could make this claim false, we observe that in almost all learned programs, their effects are equivalent in distinguishing positive and negative examples and only one is necessary. Therefore, we only use the `max` operator that fits better in the noise-tolerance task.

One challenge introduced by noise is the decrease of the density of the target application's flows in the traffic. Given a specific noise ratio, due to the above observation, we can increase the length of traffic we continuously match with the learned NetQRE program to increase the output quantity, and therefore raising the ratio of positive labels. We call this length a *sample size* for monitoring. Theoretically, decreasing the threshold of the NetQRE classifier can achieve the same effect. But since NetQRE programs only output integer values, positive and negative outputs will become indistinguishable beyond a certain noise ratio, where we have to decrease the threshold below 1.

Through our empirical study, it is observed that for each given program p and noise ratio

Sample Size	TP Rate
10	62.80%
30	92.17%
50	97%
70	100%
90	100%
100	100%

Noise Ratio	TP Rate
20%	100%
50%	100%
80%	100%
90%	84.51%
95%	66.20%
97%	50.70%

Table 1: True Positive Rates with different settings for a NetQRE classifier for Portscan. Table to the left has fixed noise ratio 80%. Table to the right has fixed sample size 70.

α, there is a *saturation sample size* l that guarantees the following property:

- if the noise ratio of the traffic is at most α, and the sample size is at least l, then 100% true positive rate can be guaranteed.

The trend can be demonstrated by Table 1. Therefore we can guarantee the recognition of the target traffic with noise by using a proper sample size.

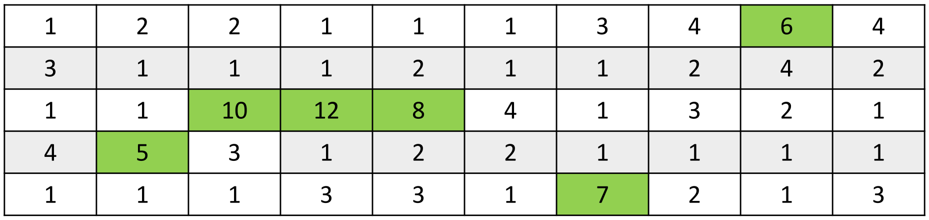
Another challenge is that the NetQRE program is learned from pure positive traffic, and may not consider the existence of noise. This could lead to a failure of regular pattern matching and the saturation sample size will not exist. The phenomenon is essentially an overfitting, and therefore can be resolved by adding some noisy positive traces into the training data. In practice, we copy 100 positive traces from the training data, mix in 20% of negative flows, and add them back to the positive training set. Through experiments, we decide that these are the minimal numbers necessary for the method to work well in our specific dataset. This can efficiently break the overfitting to the purity of the traffic without greatly increasing the learning difficulty.

4.5.3. Handling False Positives

While the above approach can guarantee the recognition of positive traces in a noisy network environment, the increase of sample sizes can also lead to more negative traffic being falsely recognized as positive. This can be addressed in two ways.

First, we can view the increase of false positive rate along with the sample size as an overfitting problem. It is related to the fact that we use only negative traces of a constant size for training. The synthesizer may be fooled to think it is only necessary to suppress the output quantity for negative traces of this specific size. This overfitting can also be broken by mixing in negative traces of varying sizes. In practice, we add 10 negative traces containing 10, 20, 50 and 100 flows respectively to the training set.

Second, we make use of the fact that true positive traces are continuously positive after satisfying the saturation sample sizes. On the other hand, negative traces are only occasionally mistaken as positive. We can empirically find a threshold of continuously observed positive results to determine a true positive event. For example, if there are at most q continuous false positive traces for sample size l given a NetQRE program p, we can decide $q + 1$ consecutive positive labels observed during the monitoring is a trigger for a true positive event. It is also possible to compute such a threshold by the observed false positive probability and a target false positive rate requirement. But since similar flows typically cluster in the network, the statistical way may not work well. This approach is demonstrated in Figure 10. Ideally, this approach can also guarantee 100% true negative rate as long as the positive events always last long enough.



1	2	2	1	1	1	3	4	6	4
3	1	1	1	2	1	1	2	4	2
1	1	10	12	8	4	1	3	2	1
4	5	3	1	2	2	1	1	1	1
1	1	1	3	3	1	7	2	1	3

Figure 10: A NetQRE program's outputs for a sequence of negative network traces. The threshold to label a trace positive is 5. The longest false positive sequence's length is 3.

4.5.4. Minimal Duration

The above approach is summarized in Figure 11, which illustrates the classification result space of a given NetQRE program p with regard to sample size l and consecutive positive label number q. The procedure will consider a positive event to be happening if and only if it falls in the upper-right quadrant. There are two reasons for a false negative: insufficient sample size and insufficient duration of the target traffic. Ideally, there is no false positive.

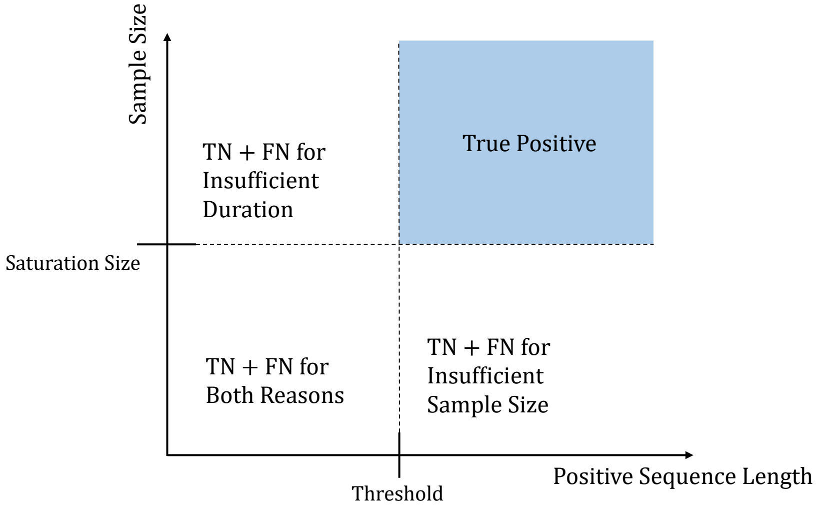


Figure 11: Decision space for a traffic type with regard to the sample size and observed positive sequence length.

Since we can always use all sample sizes simultaneously for monitoring, the only concern that may cause misclassification is the short duration of the target traffic.

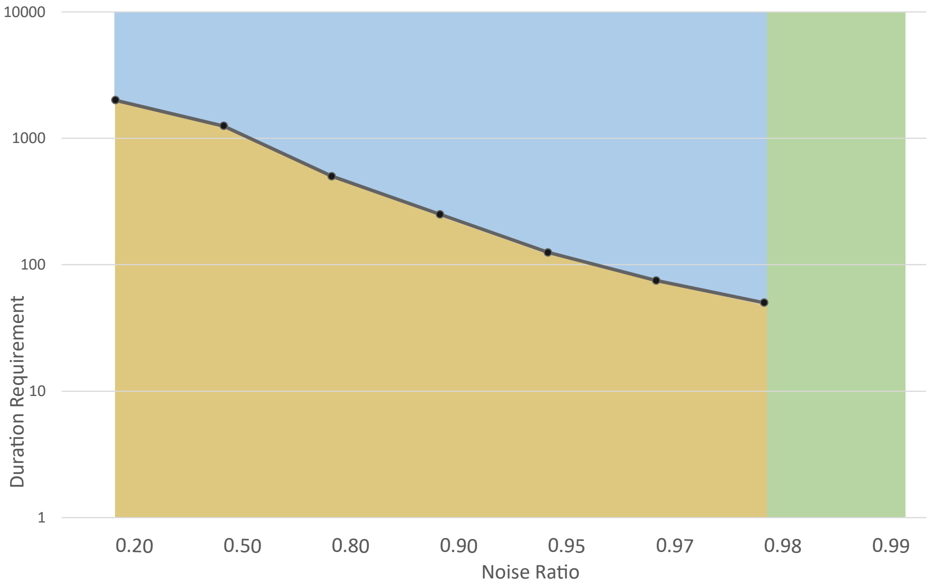


Figure 12: Property space for a traffic type with regard to the noise ratio and duration.

To understand the requirement on the duration, we use another illustration in Figure 12. We can call the combination of a certain program p and a certain sample size l a *setting*. Notice that the required consecutive positive sequence length q for false positive suppression is dependent only on the setting. This figure shows the setting's required duration at different noise ratios. Give a setting (p, l, q) and a noise ratio α, the required duration d can be computed by the following formula:

$$d = (1 - \alpha) * l * q$$

This corresponds to the curve's height at noise ratio α. The space can be divided into three sections: 1) the part above the curve is where we can guarantee accurate classification, 2) the part to the right of the curve's end is where the sample size is insufficient and may cause a false negative, 3) the part blow the curve is where the duration is insufficient and our approach will give 100% false negative. The right end of the curve is determined by experiments.

We put the curves of the same program with multiple different sample sizes in Figure 13. It can be observed that larger sample sizes can tolerate higher noise ratios, and smaller sample sizes requires shorter duration at lower noise ratios. Therefore, these settings can be used simultaneously to form a more powerful classifier, the accurate classification space of which can be derived by connecting the lower ends of all the curves in the figure, which is also illustrated in Figure 13. In this way, a higher noise ratio can be tolerated, and the required duration is short at all noise ratios. Notice that we do not assume any a priori knowledge on the real noise ratio, nor do we require it to be a constant.

4.6. Evaluation

We implemented Sharingan in 10K lines of C++ code. Our experiments are carried out in a cluster of five machines directly connected by Ethernet cables, each with 32 Intel(R) Xeon(R) E5-2450 CPUs. The frequency for each core is 2.10GHz. Arrangements of tasks are explained in the last part of Sec 4.4.3. We will evaluate the minimal feature engineering(4.6.1), accuracy(4.6.2), interpretability and editability(4.6.3), efficient implementation(4.6.4), noise tolerance(4.6.5), and synthesis algorithm efficiency(4.6.6) aspects of Sharingan in order.

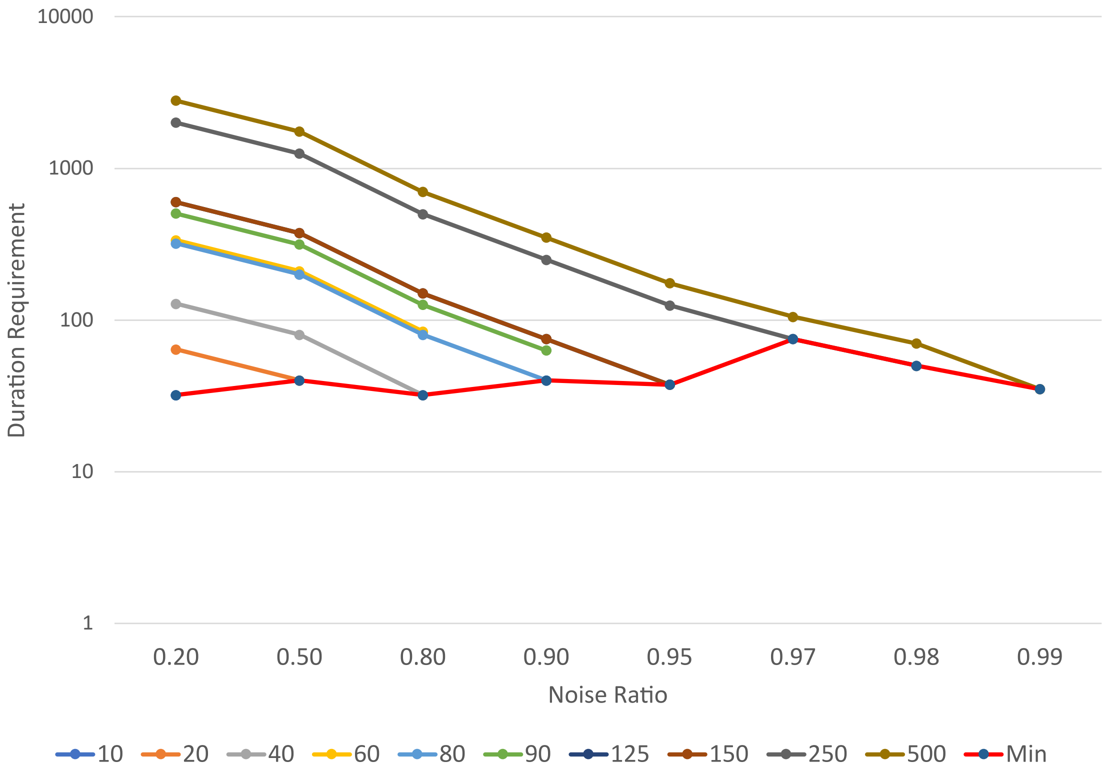


Figure 13: A set of classifier settings working together.

4.6.1. Data Preparation

We utilize eight types of attacks from the CICIDS2017 database[45; 46], a public repository of benign and attack traffic used for evaluating intrusion detection systems. They cover a wide range of attack traffic including botnets, Denial of service (DoS), port scanning, and password cracking.

The data is labelled per flow by an attack type or “Benign”. We learn each type of attack against benign traffic separately. To use as much data as possible, for each attack type, we use 1500 positive (attack) flows and 10000 negative (benign) flows for training, and another distinct data set of similar size for testing.

The main benefit of Sharingan in this step is the *minimal need* for feature engineering. We simply use all header fields of TCP and IP, and the inter-packet arrival time between adjacent packets in the same flow as features. In total, there are 19 features per packet and

$N \times 19$ features per trace of length N.

In contrast, other state-of-the-art systems rely on a carefully designed feature extraction step to work well. For example, the feature vectors included in CICIDS2017 database contain 84 features extracted by the CICFlowMeter [47; 48] tool for each flow, characterizing performance metrics of the entire flow such as duration, mean forward packet length, min activation time, etc. Kitsune [34] extracts bandwidth information over the past short periods as packet-level features. DECANTeR [49] uses HTTP-level properties such as constant header fields, language, amount of outgoing information, etc. as flow-level features.

4.6.2. Learning Accuracy

We next validate Sharingan's learning accuracy using the following evaluation methodology. For each individual attack type, we use the training data (attack and normal traffic) as input to Sharingan to learn a NetQRE program. The NetQRE program is then validated on the corresponding testing set for accuracy. The output of Sharingan includes a NetQRE program that maps a network trace to an integer output and a recommended range for the threshold. By modifying the threshold, true positive rate (TP) and false positive rate (FP) can be adjusted, as we will later explain in Section 4.6.3. We use AUC (Area under Curve) - ROC (Receiver Operating Characteristics) metric, which is a standard statistical measure of classification performance. In our evaluation, Sharingan computes the top five candidate programs instead of one, and the program with the highest test accuracy is picked as the final answer. In practice, all the top candidates are presented to the network operator to choose, based on domain knowledge.

Figure 14 contains results for eight types of attacks. Apart from AUC-ROC values, we also show the true positive rates when false positive rate is adjusted to 3 different levels: 0.001, 0.01, and 0.03. Given that noise is common in most network traffic, the last metric shown in Figure 14 is the highest achievable learning rate, which is defined as the ratio of training examples the learnt classifier can correctly classify.

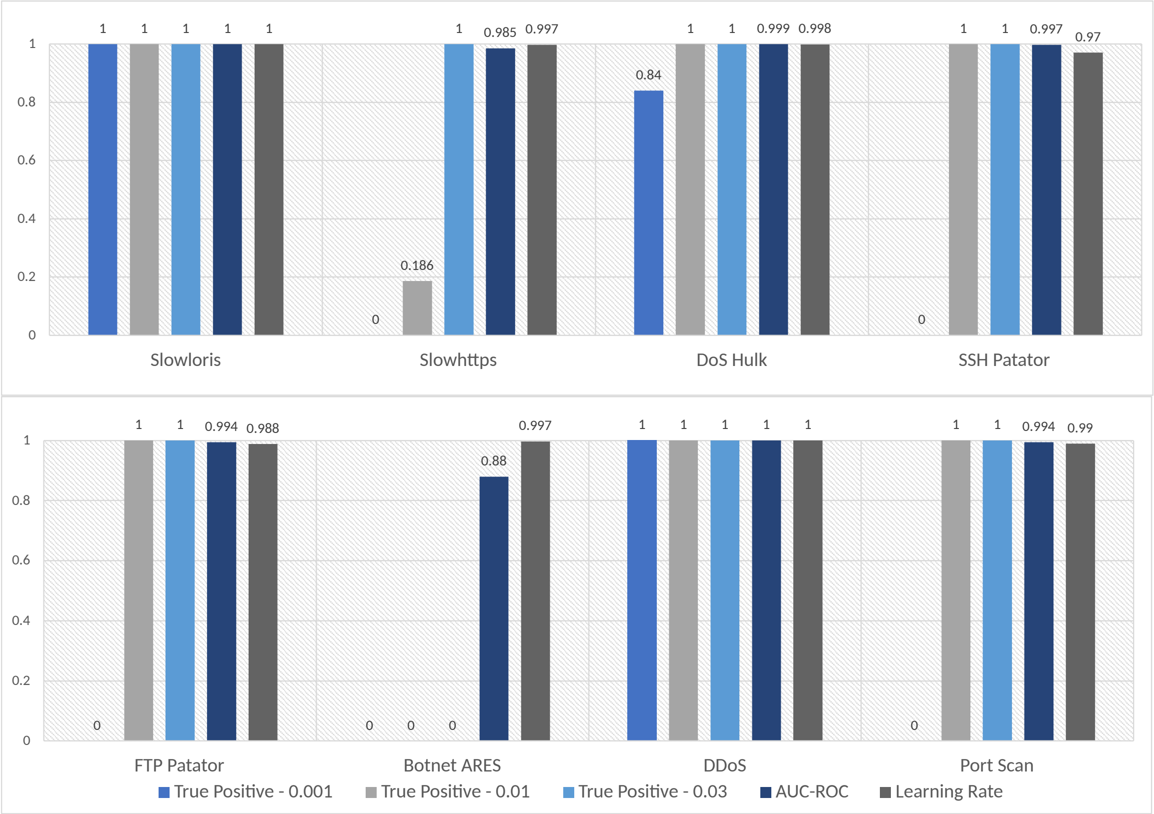


Figure 14: Sharingan's true positive rate under low false positive rate, AUC-ROC and learning rate for 8 attacks in CICIDS2017 (higher is better)

Overall, we observe that Sharingan performs well across a range of attacks with accuracy numbers on par with prior state-of-the-art systems such as Kitsune, which has an average AUC-ROC value of 0.924 on nine types of IoT-based attacks, and DECANTeR, which has an average detection rate of 97.7% and a false positive rate of 0.9% on HTTP-based malware. In six out of eight attacks, Sharingan achieves above 0.994 of AUC-ROC and 100% of true positive rate at 1% false positive rate. The major exception is Botnet ARES, which consists of a mix of malicious attack vectors. The complexity of a program that simultaneously describe these different patterns is beyond the searching ability of our current synthesis algorithm. Handling such multi-vector attacks is an avenue for our future work.

4.6.3. Post-processing and Interpretation

One of the benefits of Sharingan is that it generates an actual classification program that can be further adapted and tuned by a network operator. The program itself is also close to the stateful nature of session-layer protocols and attacks, and thus is readable and provides a basis for the operator to understand the attack cause. We briefly illustrate these capabilities in this section.

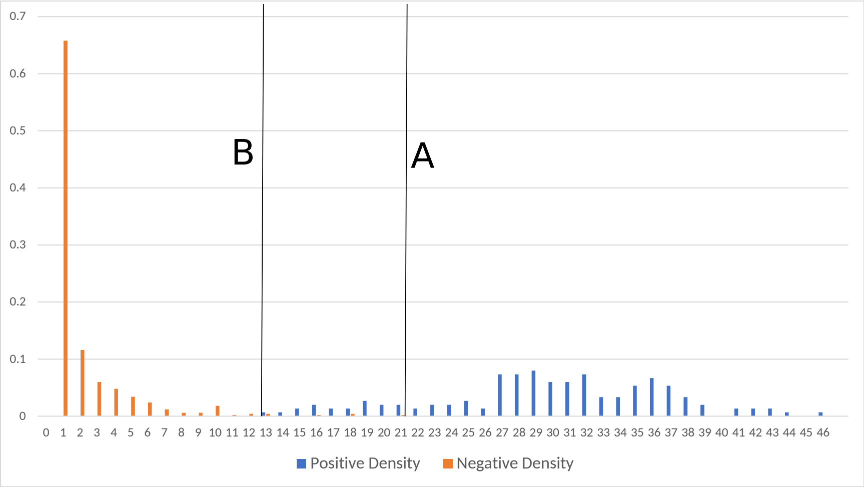


Figure 15: Output distribution of training set(DoS Hulk)

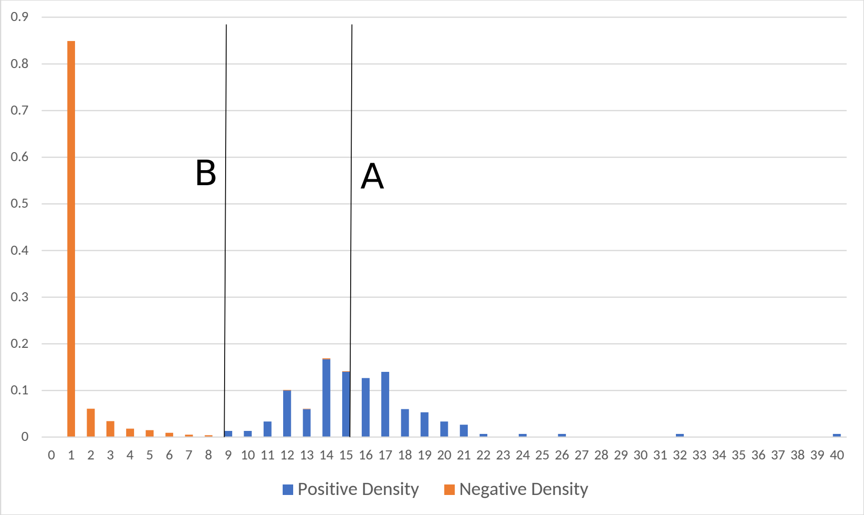


Figure 16: Output distribution of test set(DoS Hulk)

FP-TP Tradeoff Network operators need to occasionally tune a classifier's sensitivity to false positives and true positives. Sharingan generates a NetQRE program with a threshold T. This threshold can be adjusted to vary the false positive and true positive rate. Figures 15 and 16 show the output distribution from positive and negative examples in the DoS

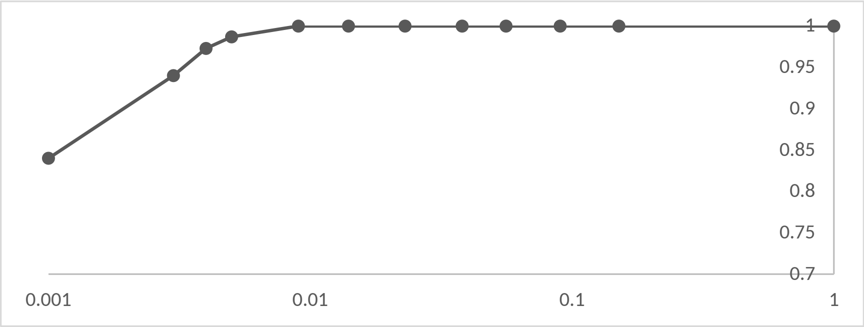


Figure 17: ROC Curve, logarithmic scale(DoS Hulk)

Hulk attack. A denotes the largest negative output and B denotes the smallest positive output. When $A > B$, there is some unavoidable error. We can slide the threshold T from B to A and obtain an ROC curve for the test data, as illustrated in Figure 17.

Interpretation We describe a learnt NetQRE program to demonstrate how a network operator can interpret the classifiers. The NetQRE program synthesized by Sharingan for DDoS task above is:

```
( ( /_* A _* B _*/ )*sum /_* C _*/ )sum > 4
Where
A = [ip.src_ip->[0%,50%]]       B = [tcp.rst==1]
C = [time_since_last_pkt<=50%]
```

DDoS is a flood attack from a botnet of machines to exhaust memory resources on the victim server. The detected pattern consists of packets that start with source IP in a certain range, followed by a packet with the reset bit set to 1, and then a packet with a short time interval from its predecessor. Finally, the program considers the flow a match if the patterns show up with a total count of over 4.

The range of source IP addresses specified in the pattern possibly contains botnet IP addresses. Attack flows are often reset when the load cannot be handled or the flows' states cannot be recognized, In either case, it is highly possible that the sender is sending a large number of flows without keeping their states. If the attack tool is adapted from general purpose tools like hping3, it will follow the convention and end these flows with a reset. which indicates the attack is successfully launched. Packets with short intervals further support a flood attack. Unique properties of DDoS attack are indeed captured by this program!

Our next use case is based on Hulk, an attack similar to Slowloris. Hulk issues multiple HTTPS requests, trying to keep them alive, adding more and more connections as time moves forward, and eventually overwhelming the webserver. Hulk requests have a high level of variety, adding difficulty to learning even with knowledge of the requests' contents. The synthesized NetQRE program to identify Hulk is as follows:

```
( /_* A _*/  ( /_* B _*/ )*sum )max > 13
Where
A = [tcp.seq>=50%]
B = [tcp.fin==1]
```

The program first identifies a large sequence number, which is an indication that someone is trying to keep the connection long. This is followed by a large number of normally finished TCP connections. Connecting the two, it is not hard to guess someone is launching a long and slow attack. This is exactly how Hulk works to cause a DoS.

A takeaway in this use case is that Sharingan is able to build accurate classifiers without reliance on application-layer data, which is often encrypted. In some cases, even if application-layer data is desirable, Sharingan is able to build effective classifier simply by relying on features based on TCP and IP fields. We can observe that, even if the knowledge at the same level of the application is not available, the specific tool or specific method used can still unintentionally leave patterns at lower level. Such patterns can be captured by Sharingan.

Refinement by Human Knowledge Finally, an advantage of generating a program for classification is that it enables the operator to augment the generated NetQRE program with domain knowledge before deployment. For example, in the DDoS case, if they know that the victim service is purely based on TCP, they can append `[ip.type = TCP]` to all predicates. Alternatively, if they know that the victim service is designed for 1000 requests per second, they can explicitly replace the arrival time interval with $1ms$. The modified program then is:

```
( ( /_* A _* B _*/ )*sum /_* C _*/ )sum > 4
Where
A = [ip.type = TCP]&&[ip.src_ip->[0%,50%]]
B = [ip.type = TCP]&&[tcp.rst==1]
C = [ip.type = TCP]&&[time_since_last_pkt<=1ms]
```

4.6.4. Deployment Scenarios

We now describe three ways for network operators to deploy the output of Sharingan: (1) taking action hinted by the interpretation; (2) directly executing the NetQRE program as a monitoring system; and (3) translating the NetQRE program to rules in other monitoring systems.

Revisiting the DDoS example in Section 4.6.3, in the first case, the operator may refine the source IP part to find out the accurate range of attacker machines and block them.

If the NetQRE program itself is to be used as a monitoring system, its runtime system can be directly deployed on any general purpose machine. Prior work [41] has shown that NetQRE generates performance that is comparable to optimized low-level implementations. Moreover, these programs can be easily compiled into other formats acceptable to existing monitoring systems.

4.6.5. Noise Tolerance

We next evaluate the efficacy of our noise tolerance algorithm for network monitoring tasks. The same intrusion detection benchmark is used for this evaluation. Intrusion detection is a typical scenario where noise tolerance is required. It is especially important to clearly tell apart each type of attack as well as the background normal traffic. Explaining each type of attack that is going on is also necessary for the operators to take corresponding countermeasures.

Considering that our algorithm reduces the detection accuracy problem to target traffic duration requirement, we specifically want to answer the following questions in this section:

- How long do we need each attack type to last to guarantee the detection under different noise ratios?

Attack Type	Noise Type
DDoS	Portscan
Portscan	DDoS
Slowloris	Hulk
Hulk	Slowloris
SlowHTTPS	Hulk
FTP Patator	SSH Patator

Table 2: List of attack types used as noise

- How will the performance be affected if another similar but different attack type is going on in parallel?
- How well can we estimate the noise tolerance parameters for deployment with limited training data?

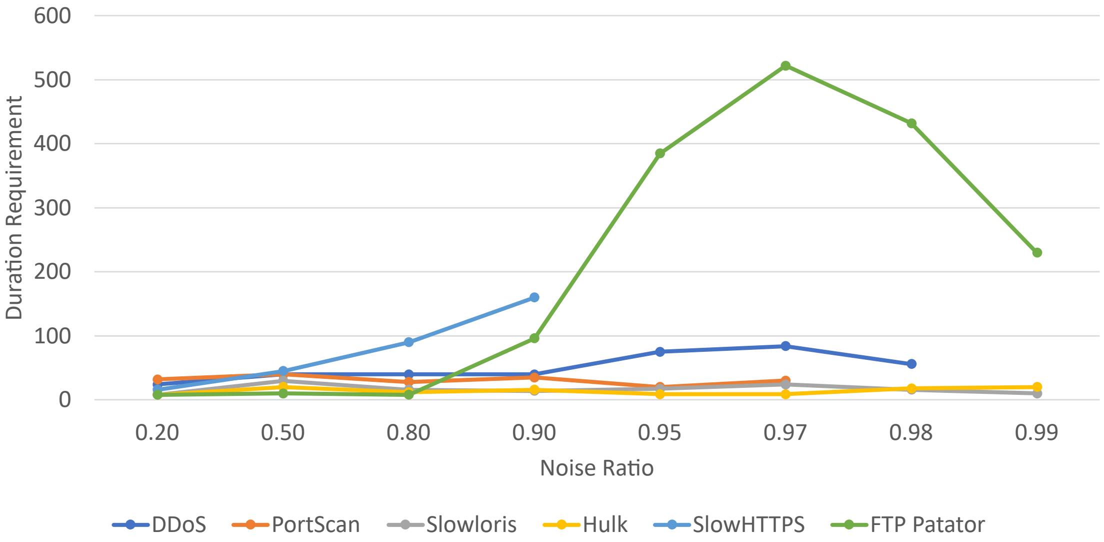


Figure 18: Minimal duration requirements for 6 successfully learned attack types

We next describe the methodology to conduct this evaluation. First, we prepare positive and negative training and testing data for 8 types of attacks in the same way as section 4.6.1. Next, we evaluate each attack type individually. We add extra data entries into the training data as is described in section 4.5 in order to reduce overfitting. A NetQRE program is synthesized for this attack type from this improved training set. After that, the positive training set is resampled to find the saturation sample sizes at a few representative noise ratios. The negative training set is also resampled to find the minimal positive trace se-

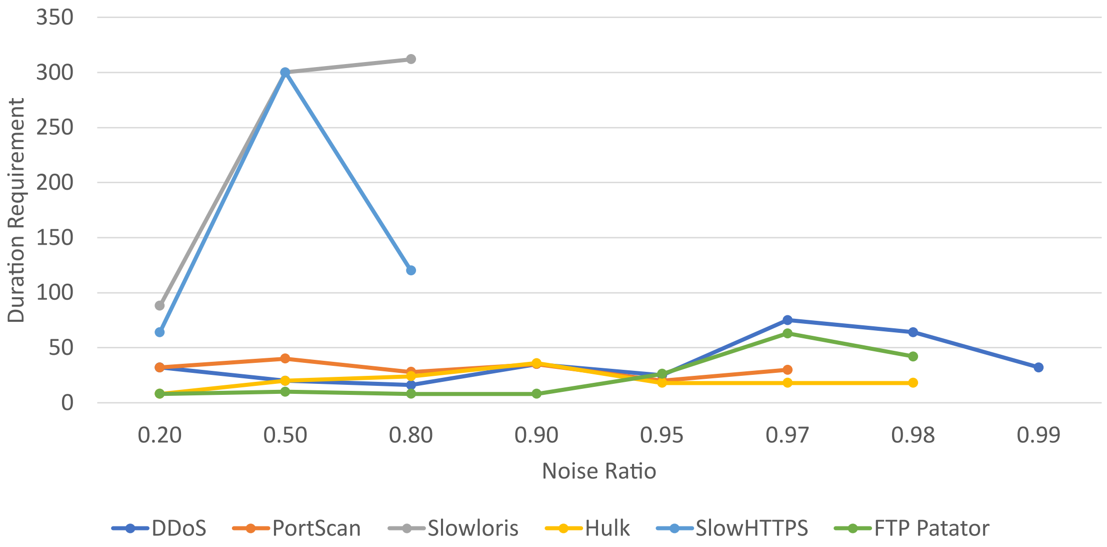


Figure 19: Minimal duration requirements for 6 successfully learned attack types with another close attack type mixed into the noise

quence length to trigger a positive alert. Finally, after determining the optimal values for these parameters, we use the formula in section 4.5 to compute the required duration.

We perform the procedure above on a data set assuming the noise is composed purely of normal traffic. We then mix another similar but different type of attack traffic into the noise and perform the procedure again to observe its effect on the duration requirement. The amount of the other type of attack traffic is the same as the target traffic (or at most 10%).

Finally, we apply the resulting classifier to a separate testing set of similar size and report the classification accuracy results.

Figure 18 reports the result on the original training set. The synthesizer successfully learned the required NetQRE program on the improved training data for 6 out of 8 attack types. Two attack types are unsuccessful in the learning phase. The Botnet ARES attack is too complicated to learn with the additional difficulty introduced by the new training data. SSH Patator is too indistinguishable from a normal SSH connection with noise mixed in. For other 6 traffic types, our approach can tolerate at least 90% of noise. For most of them the tolerance level is close to 99%, and the required duration of the attack to guarantee

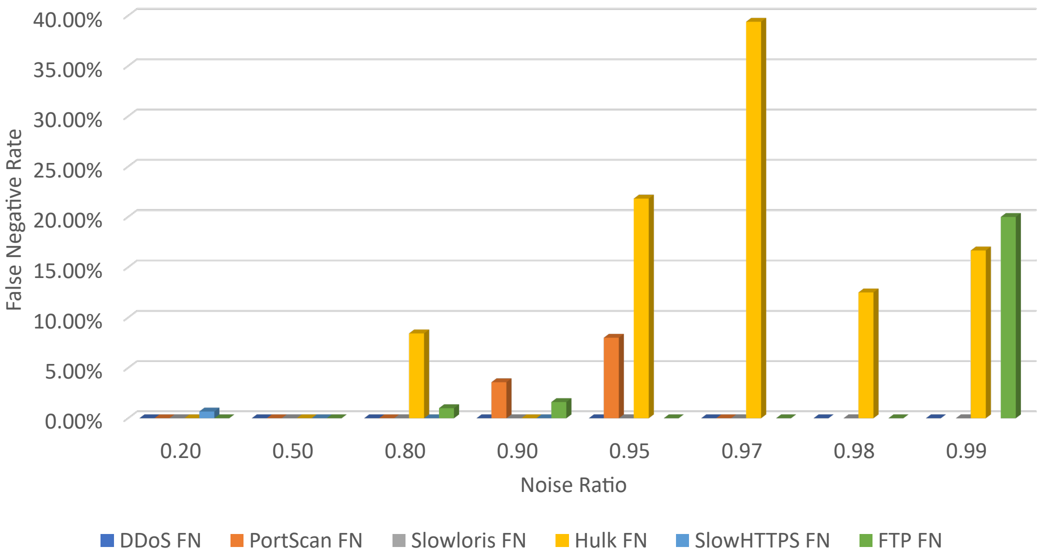


Figure 20: Validation false negative rate at different noise ratios for each attack type

correct classification of this data set mostly falls below 100 flows. Some attack types are less distinguishable from normal traffic when there are more noises. And we do not consider any sample size over 500 flows. Therefore, the noise level that can be tolerated for them are lower than other types.

Figure 19 reports the result after adding in another type of attack into the noise. The attack type used as noise corresponding to each target type is listed in Table 2. We can observe that the additional noise type only has some minor attack-dependent influence on the results. One major impact is that the noise tolerance ability drops for Slowloris and SlowHTTPS traffic, mostly due to false positives. This is possibly explained by the fact that the added noise attack type is too similar in traffic characteristics to the target type.

Figure 20 and 21 shows the validation results on the testing set. Both false positive and false negative rates for all attack types remain low by 90% of noise ratio. Above this point, false positive ratio starts to rise sharply. This is partly due to that the negative traces for the training and testing sets are collected from different time of a day and are probably not very similar. The learned NetQRE programs may not properly generate low outputs for negative traces in the testing set. Essentially, this is an overfitting phenomenon, and could

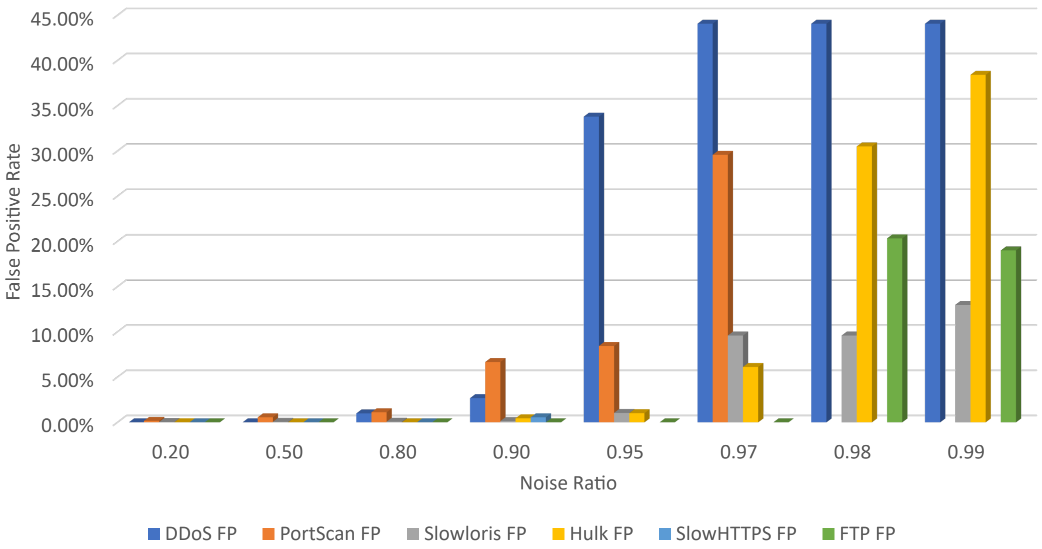


Figure 21: Validation false positive rate at different noise ratios for each attack type

be mitigated by including more traces collected from different environments for training.

In addition, there is a tradeoff between accuracy and required target traffic duration. If we assume the target traffic can be observed for a longer duration, we can classify with higher accuracy. The learning phase decides the minimal duration required to achieve the best accuracy for the training set. We observe that slightly increasing this number can significantly improve the accuracy results in the testing set, at the cost of a greater risk that the duration requirement is not satisfied. The specific parameters to use can be decided by the user based on the scenario.

4.6.6. Program Synthesis Performance

Synthesis time: In our final experiment, the performance of Sharingan is measured, in terms of time needed for program synthesis.

Figure 22 shows the program complexity (Y-axis) and synthesis (learning) time (in minutes). Not surprisingly, complex programs require more time to synthesize. We further observe that Sharingan is able to synthesize complex programs with at least 20-30 terms, mostly within minutes to an hour, which is practical for many real-world use cases and can be further reduced through parallelism over more machines. As a comparison, Kitsune reports training times between 8 minutes and 52 minutes on individual attacks [34], and DECANTeR reports training times between 5 hours and 10 hours on individual users' data [49].

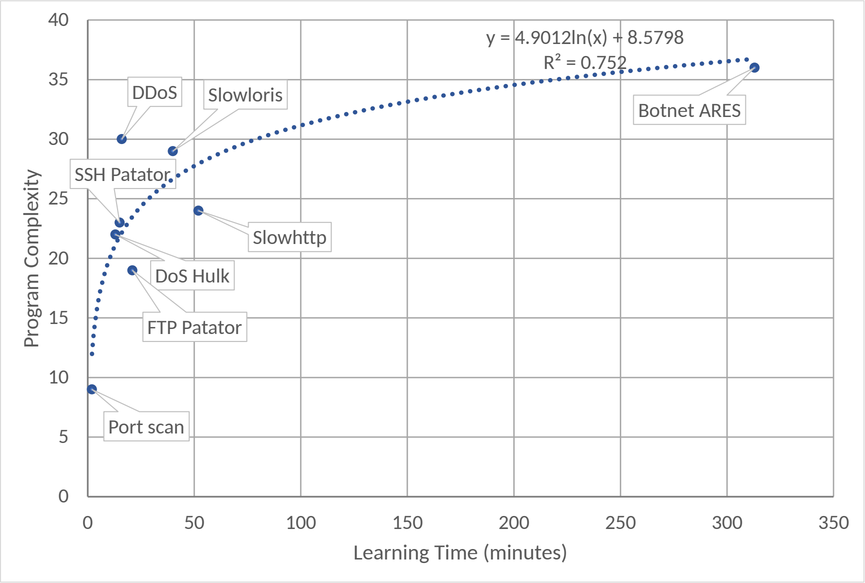


Figure 22: Time-complexity relation

Effectiveness of Optimizations. We explore the effectiveness of the individual optimization strategies described in Section 5.3.4. In Figure 23, we compare the synthesis time and the number of programs searched for a fully optimized Sharingan against results from disabling each optimization. SSH Patator is used as the demonstrating example since it is moderately complex.

We observe that disabling partial execution optimization makes both metrics significantly worse. Being able to prune early can indeed greatly reduce time wasted on unnecessary exploration and checking. By disabling merge search, although the number of programs searched decreases, the total synthesis time increases given the overhead of having to check each program against the entire data set. The synthesis cannot finish within reasonable time if both are disabled.

In summary, all optimization strategies are effective to speed up the synthesis process. A

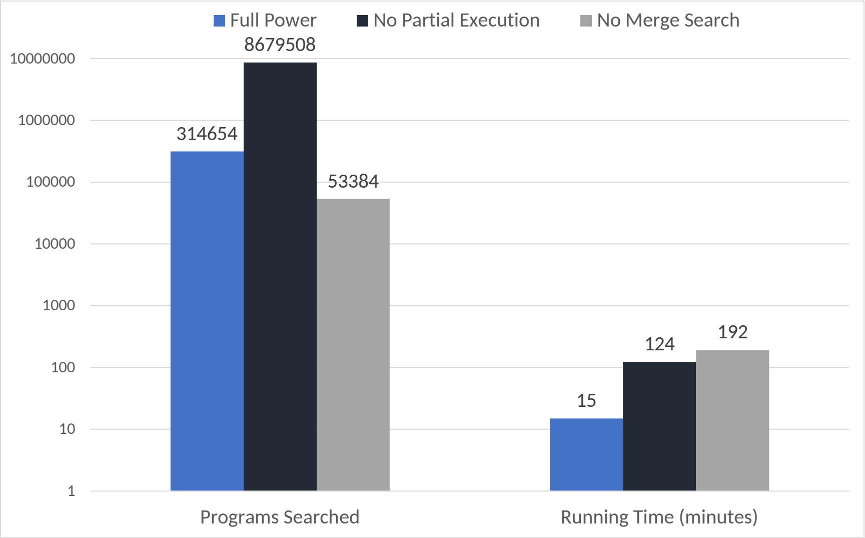


Figure 23: Impact of optimizations on synthesis performance

synthesis task that is otherwise impossible to finish within practical time can now be done in less than 15 minutes.

4.7. Limitations

The current version of Sharingan has two major limitations.

First, the training stage requires pure negative traces and almost pure positive traces of a single application type as the specification. Although pure negative traces are easy to collect without user involvement, positive traces need manual effort to extract. Since realistic network traffic usually contains multiple application types mixed up. Besides, as is shown by the evaluation, Sharingan also has difficulty learning from applications with multiple distinct operations. It is an interesting future work to explore ways the synthesizer could learn from mixed traffic types.

Second, the classification granularity is coarse-grained. If the traces provided for classification may contain negative flows or more than one type of applications, the learned classifiers are only able to decide if the entire trace contains the target application. But they are not able to tell which specific packets or flows belong to the application. This may hinder the follow-up actions. The problem can be mitigated by the interpretability of the learned classifiers, since more information needed to decide the action can be determined statically.

4.8. Takeaways

In this section, we demonstrate how network domain-specific rules can be automatically learned from labelled traffic. We present Sharingan, which develops syntax-guided synthesis techniques to generate NetQRE programs for classifying session-layer network traffic. Sharingan can be used for generating network monitoring queries or signatures for intrusion detection systems from labeled traces. Our results demonstrate three key value propositions for Sharingan, namely minimal feature engineering, efficient implementation, and interpretability as well as editability. While achieving accuracy comparable to state-of-the-art statistical and signature-based learning systems, Sharingan is significantly more usable and requires synthesis time practical for real-world tasks.

CHAPTER 5

Control Plane Program Repairing

In this section, we introduce our solution to the second type of programming task. We designed and implemented Orion system that can automatically repair control plane programs given input/output examples.

5.1. Design Goals

Accuracy Network is sensitive to even minor mistakes. Therefore we want our repairing procedure to prioritize high accuracy rather than discovering a large number of bugs through huge projects.

Scalability Although strict semantic encoding can provide higher accuracy results, they scale poorly with the program's size due to the difficulty of solving the constraints. We want to improve based on these methods so that the tool can scale better in addition to achieving high accuracy.

5.2. Overview

In this section, we give a high-level overview of our repair techniques and walk through the Orion tool using an example adapted from the Floodlight SDN controller [50].

Figure 24 shows a simplified code snippet about firewall rules in Floodlight. Specifically, the program consists of two classes – FirewallRule and MacAddress. The FirewallRule class describes rules enforced by the firewall, including information about source and destination mac addresses. The MacAddress class is an auxiliary data structure that stores the raw value of mac addresses[1] and provides functions useful to network applications.

The network program shown in Figure 24 is problematic because the `isSameAs` function compares two mac addresses using the `!=` operator rather than a negation of the `equals`

[1] A unique 48-bit number that identifies each network device.

```
@network
public class MacAddress {
  private long value;
  private MacAddress(long value) { this.value = value; }
  public static MacAddress NONE = new MacAddress(0);
  public static MacAddress of(long value) { return new MacAddress(
    value); }
  ...
}
public class FirewallRule {
  public MacAddress dl_dst;
  public boolean any_dl_dst;
  public FirewallRule() { dl_dst = MacAddress.NONE; any_dl_dst = true
    ; ... }
  public boolean isSameAs(FirewallRule r) {
    if (... || any_dl_dst != r.any_dl_dst
            || (any_dl_dst == false &&
                dl_dst != r.dl_dst)) {
        return false;
    }
    return true;
  }
  ...
}
```

Figure 24: Code snippet about a bug in Floodlight.

functions. The `!=` operator only compares two objects based on their memory addresses, whereas the intent of the developer is to check if two mac addresses have the same raw value. The bug is revealed by the unit test shown in Figure 25, then confirmed and fixed by the Floodlight developers [2]. In the remainder, let us illustrate how Orion localizes this bug based on unit tests `test(1, 2) = false` and `test(1, 1) = true` and automatically synthesizes a patch to fix it.

Domain-Specific Abstraction. Orion has incorporated abstractions for common network data structures. For example, the MacAddress Class marked with the @network annotation contains a 48-bit integer field and several handy functions for bit manipulations. We have pre-built the high-level specifications and function summaries for data structures like MacAddress based on domain knowledge. Orion is able to leverage the specifications

[2] https://github.com/floodlight/floodlight/commit/4d528e4bf5f02c59347bb9c0beb1b875ba2c821e

```java
public boolean test(long mac1, long mac2) {
  FirewallRule r1 = new FirewallRule();
  r1.dl_dst = MacAddress.of(mac1); r1.any_dl_dst = false;
  FirewallRule r2 = new FirewallRule();
  r2.dl_dst = MacAddress.of(mac2); r2.any_dl_dst = false;
  boolean output = r1.isSameAs(r2);
  return output;
}
```

Figure 25: Unit test that reveals the bug in FirewallRule.

and summaries for symbolic analysis without additional user input.

At a high level, Orion enters a loop that iteratively attempts to find the fault location and synthesize the patch. Since our repair technique works in a modular fashion, Orion first selects a function F in the program and tries to repair each possible fault location at a time. If Orion cannot synthesize a patch consistent with the provided unit tests for any potential fault location in F, it backtracks and selects the next function and repeats the same process until all possible functions are checked. We now describe the experience of running Orion on our illustrative example.

Iteration 1. Orion selects the constructor of FirewallRule as the target function. Fault localization determines that the fault is located at the `dl_dst = MacAddress.NONE` part of Line 12, because it is related to the equality checking in the unit test. However, it is not the fault location. Although Orion invokes its underlying synthesizer and tries to synthesize a patch for this location, the synthesis procedure cannot find a solution that passes the unit test by replacing the `dl_dst = MacAddress.NONE` statement.

Iteration 2. Orion selects the same function – constructor of FirewallRule, but the fault localization switches to a different statement `any_dl_dst = true` at Line 12. Similar to Iteration 1, the synthesizer cannot generate a correct patch by replacing this statement.

Iteration 3. Since none of the statements in the constructor is the fault location, Orion now selects a different function: `isSameAs`. The fault localization determines that `any_dl_dst`

$$
\begin{array}{rcl}
\textit{Program } \mathcal{P} & ::= & \mathcal{C}^{+} \\
\textit{Class } \mathcal{C} & ::= & \textbf{@network}?\ \textbf{class}\ C\ \{a^{+}\ F^{+}\} \\
\textit{Function } F & ::= & \textbf{function}\ f(x_1, \ldots, x_n)\ (L : s)^{+} \\
\textit{Statement } s & ::= & l := e \mid \textbf{jmp}\ (e)\ L \mid \textbf{ret}\ v \mid x := \textbf{new}\ C \\
 & \mid & x := C.f(v_1, \ldots, v_n) \mid x := y.f(v_1, \ldots, v_n) \\
\textit{LValue } l & ::= & x \mid x.a \mid x[v] \\
\textit{Immediate } v & ::= & x \mid c \\
\textit{Expression } e & ::= & l \mid c \mid op(e_1, \ldots, e_n)
\end{array}
$$

$$
x, y \in \textbf{Variable} \quad c \in \textbf{Constant} \quad L \in \textbf{LineID}
$$

$$
C \in \textbf{ClassName} \quad f, f_0 \in \textbf{FuncName} \quad a \in \textbf{FieldName}
$$

Figure 26: Syntax of network programs.

`= false` at Line 15 may be the fault location as it may affect the testing results. However, having tried to replace the statement with many other candidate statements, e.g., `r.any_dl_dst = false`, `any_dl_dst = true`, the synthesizer still fails to generate the correct patch.

Last iteration. Finally, after several attempts to localize the fault, Orion identifies the fault lies in `dl_dst != r.dl_dst` at Line 16, which is indeed the reported bug location. At this time, the synthesizer manages to generate a correct patch `!dl_dst.equals(r.dl_dst)`. Replacing the original condition at Line 16 with this patch results in a program that can pass all the provided test cases, so Orion has successfully repaired the original faulty program.

5.2.1. Preliminaries

In this section, we present the language of network programs and describe a program formalism that is used in the rest of this chapter. We also define the program repair problem that we want to solve.

Language of Network Programs

The language of network programs considered in this chapter is summarized in Figure 26. A network program consists of a set of classes, where each class has an optional annotation @network to denote that the class is a network-related data structure such as mac address and IPv4 address. The annotations are helpful for recognizing network data structures and facilitate domain-specific analysis during program repair.

Each class in the program consists of a list of fields and functions. Each function has a name, a parameter list, and a function body. We collectively refer to the function name and parameter list as the *function signature*. The function body is a list of statements, where each statement is labeled with its line number. Various kinds of statements are included in our language of network programs. Specifically, assign statement $l := e$ assigns expression e to left value l. Conditional jump statement **jmp** (e) L first evaluates predicate e. If the result is true, then the control flow jumps to line L; otherwise, it performs no operation. Note that our language does not have traditional if statements or loop statements, but those statements can be expressed using conditional jumps. [3] Return statement **ret** v exits the current function with return value v. New statement $x :=$ **new** C creates an object of class C and assigns the object address to variable x. Static call $x := C.f(v_1, \ldots, v_n)$ invokes the static function f in class C with arguments $v_1, \ldots, v_n$ and assigns the return value to variable x. Similarly, virtual call $x := y.f(v_1, \ldots, v_n)$ invokes the virtual function f on *receiver* object y with arguments $v_1, \ldots, v_n$ and assigns the return value to variable x. Different kinds of expressions are supported including constants, variable accesses, field accesses, array accesses, arithmetic operations, and logical operations. Since the semantics of network programs is similar to that of traditional programs written in object-oriented languages, we omit the formal description of semantics.

As is standard, we assume class names are implicitly appended to function names, so a function signature uniquely determines a function in the program. In addition, we assume each statement in the program is labeled with a globally unique line number, and line numbers are consecutive within a function. In the remainder of this chapter, we use several auxiliary functions and relations about the control flow structure of programs. Specifically, $\mathsf{FirstLine}(\mathcal{P}, F)$ returns the first line of function F in program $\mathcal{P}$. $\mathsf{IsPrevLine}(\mathcal{P}, L_1, L_2)$ is a relation representing that L_1 is a control-flow predecessor of L_2 in program $\mathcal{P}$: L_1 is a

[3] Our repair techniques only handle bounded loops. If there are unbounded loops in the network program, we need to perform loop unrolling.

predecessor of L_2 if (1) L_2 is the next line of L_1 and L_1 is not a return statement, or (2) L_1 is a jump statement and the target location of jump is L_2.

Problem Statement

In this chapter, we assume a unit test t is written in the form of a pair (I, O), where I is the input and O is the expected output. Given a network program $\mathcal{P}$ and a unit test $t = (I, O)$, we say $\mathcal{P}$ passes the test t if executing $\mathcal{P}$ on input I yields the expected output O, denoted by $[\![\mathcal{P}]\!]_I = O$. Otherwise, if $[\![\mathcal{P}]\!]_I \neq O$, we say $\mathcal{P}$ fails the test t. In general, given a network program $\mathcal{P}$ and a set of unit tests $\mathcal{E}$, program $\mathcal{P}$ is *faulty* modulo $\mathcal{E}$ if there exists a test $t \in \mathcal{E}$ such that $\mathcal{P}$ fails on t.

Now let us turn the attention to the meaning of fault locations and patches.

Fault location and patch. Let $\mathcal{P}$ be a program that is faulty modulo tests $\mathcal{E}$. Line L is called the *fault location* of $\mathcal{P}$, if there exists a statement s such that replacing line L of $\mathcal{P}$ with s yields a new program that can pass all tests in $\mathcal{E}$. Here, the statement s is called a *patch* to $\mathcal{P}$.

Having defined these concepts, we precisely describe our research problem next.

Problem statement. Given a network program $\mathcal{P}$ that is faulty modulo tests $\mathcal{E}$, our goal is to find a fault location L in $\mathcal{P}$ and generate the corresponding patch s, such that for any unit test $t \in \mathcal{E}$, the patched program $\mathcal{P}'$ can always pass the test t.

5.3. Modular Program Repair

In this section, we present our algorithm for automatically repairing network programs from a set of unit tests.

5.3.1. Algorithm Overview

The top-level repair algorithm is described in Algorithm 1. The Repair procedure takes as input a faulty network program $\mathcal{P}$ and unit tests $\mathcal{E}$ and produces as output a repaired program $\mathcal{P}'$ or $\perp$ to indicate repair failure.

Algorithm 1 Modular Program Repair

```
1:  procedure REPAIR($\mathcal{P}, \mathcal{E}$)
      Input: Program $\mathcal{P}$, examples $\mathcal{E}$
      Output: Repaired program $\mathcal{P}'$ or $\bot$ to indicate failure
2:    $\mathcal{P} \leftarrow \mathsf{Abstraction}(\mathcal{P})$;
3:    $\mathcal{V} \leftarrow \{L \mapsto \mathit{false} \mid L \in \mathsf{Lines}(\mathcal{P})\}$; $\mathcal{P}' \leftarrow \bot$;
4:    while $\mathcal{P}' = \bot$ do
5:        $F \leftarrow \mathsf{SelectFunction}(\mathcal{P}, \mathcal{V})$;
6:        if $F = \bot$ then return $\bot$;
7:        $\mathcal{V}, \mathcal{P}' \leftarrow$ REPAIRFUNCTION($\mathcal{P}, F, \mathcal{E}, \mathcal{V}$);
8:    return $\mathcal{P}'$;
9:  procedure REPAIRFUNCTION($\mathcal{P}, F, \mathcal{E}, \mathcal{V}$)
      Input: Program $\mathcal{P}$, function $F$, examples $\mathcal{E}$, visited map $\mathcal{V}$
      Output: Updated visited map $\mathcal{V}$, repaired program $\mathcal{P}'$
10:   $\mathcal{P}' \leftarrow \bot$;
11:   while $\mathcal{P}' = \bot$ do
12:       $L \leftarrow$ LOCALIZEFAULT($\mathcal{P}, F, \mathcal{E}, \mathcal{V}$);
13:       if $L \neq \bot$ then
14:           $\mathcal{V} \leftarrow \mathcal{V}[L \mapsto \mathit{true}]$;
15:       else
16:           $\mathcal{V} \leftarrow \mathcal{V}[L' \mapsto \mathit{true} \mid \mathsf{TransInFunc}(L', \mathcal{P}, F)]$;
17:       if $L = \bot$ or $\mathsf{IsCallStmt}(\mathcal{P}, L)$ then return $\mathcal{V}, \bot$;
18:       $\mathcal{P}' \leftarrow$ SYNTHESIZEPATCH($\mathcal{P}, \mathcal{E}, F, L$);
19:   return $\mathcal{V}, \mathcal{P}'$;
```

At a high level, the Repair procedure maintains a visited map $\mathcal{V}$ from line numbers to boolean values, representing whether each line of $\mathcal{P}$ is checked or not. Specifically, $\mathcal{V}[L] = \mathit{true}$ indicates line L is checked; otherwise, $\mathcal{V}[L] = \mathit{false}$ means L is not checked yet. As shown in Algorithm 1, the Repair procedure first applies the domain-specific abstraction to program $\mathcal{P}$ (Line 2) and initializes the visited map $\mathcal{V}$ by setting every line in $\mathcal{P}$ as not checked (Line 3). Next, it tries to iteratively repair $\mathcal{P}$ in a modular way until it finds a program $\mathcal{P}'$ that is not faulty modulo tests $\mathcal{E}$ (Lines 4 – 8). In particular, the Repair procedure invokes SelectFunction to choose a function F as the target of repair (Line 5). If none of the functions in $\mathcal{P}$ can be repaired, it returns $\bot$ to indicate that the repair procedure failed (Line 6). Otherwise, it invokes the RepairFunction procedure (Line 7) to enter the localization-synthesis loop inside the target function F.

Focusing on one function at a time and repairing programs in a modular fashion is beneficial for two reasons. First, given a faulty program $\mathcal{P}$ and a target function F, we only need to check functions in the call stack from the main function of $\mathcal{P}$ to F. It can significantly reduce the number of functions to analyze and the number of locations to check, which enables faster program analysis and repair. Second, we can summarize the behavior of non-target functions and reuse summaries to achieve better scalability. Specifically, given a target function F, all functions that are not in the call stack of F can be replaced with their corresponding summaries. It further decreases the number of locations to track, because we do not need to inline non-target functions that are invoked in the transitive callers of F.

In addition to the program $\mathcal{P}$ and tests $\mathcal{E}$, the RepairFunction procedure takes as input a target function F and the current visited map $\mathcal{V}$. It produces as output the updated version of the visited map $\mathcal{V}$, as well as a repaired program $\mathcal{P}'$ or $\perp$ to indicate that the function F cannot be repaired. As shown in Lines 11 – 18 of Algorithm 1, RepairFunction alternatively invokes sub-procedures LocalizeFault and SynthesizePatch to repair the target function. In particular, the goal of LocalizeFault is to identify a fault location in function F. If LocalizeFault manages to find a fault location L in F, then line L is marked as visited (Line 14). Otherwise, if LocalizeFault returns $\perp$, it means function F and all functions transitively invoked in F are correct or not repairable. In this case, all lines in F and its transitive callees are marked as checked (Line 16). Furthermore, if the identified fault location L corresponds to a statement that invokes F', it means the fault location is inside F'. Thus, RepairFunction directly returns $\perp$ (Line 17) and SelectFunction will choose F' as the target function in the next iteration. On the other hand, the goal of the sub-procedure SynthesizePatch is generating a patch for function F given the fault location L. If SynthesizePatch successfully synthesizes a patch and produces a non-faulty program $\mathcal{P}'$, then the entire procedure succeeds with repaired program $\mathcal{P}'$. Otherwise, RepairFunction backtracks with a new program location and repeat the same process.

In the rest of this section, we explain domain-specific abstraction, fault localization, and patch synthesis in more details.

5.3.2. Domain-Specific Abstraction

A domain-specific abstraction is essentially a function's summary. For those repeatedly used network classes (identified by the @network annotation), we can pre-define some more succinct abstractions based on domain knowledge to make the analysis easier. The abstraction $\mathcal{A}[F]$ of a function F is an over-approximation of F that is precise enough to characterize the behavior of F.

The abstraction is useful due to three observations. First, source code for a programs may only be partially available due to the use of high-level interface and native implementation. For example, when comparing the equality between two network addresses, the getClass function is frequently used, but its implementation depends on the runtime and is not available. To make the analysis easier, we can instead use the following abstraction for such comparison:

$$\mathcal{A}[\text{equals}] : \lambda x.\ \lambda y.\ (x.\mathit{dtype} = y.\mathit{dtype} \wedge x.\mathit{value} = y.\mathit{value}),$$

where $x.\mathit{dtype}$ denotes the dynamic type of the object x.

Second, when network functions have an overlap in the set of protocols they need to process, which happens quite frequently, they will also share the same data structure corresponding to that protocol, such as headers for TCP/IP protocols. A plain encoding of operations on these data types in the source code is not succinct enough, since the encoding method must be general. We can replace them with native implementations in the encoder's language that include much smaller formulae.

Third, network programs have complex operations that are challenging for symbolic reasoning. For instance, bit manipulations are heavily used in network data structures to process addresses and control flags. While bit manipulations can improve the performance of network programs, they present significant challenges for symbolic analysis due to the encoding in the theory of bitvectors. We can give an abstraction that is equivalent in correctness but simpler in the behavior. For example, a checking function applies a number of bit-shift operations to an address and compares the results to some constant control flags. The bit-shift operation is an interaction between integer and bitvector, which is very slow to reason about. We can shift the constant control flags in the opposite direction(which are still hard-coded constants) and skip the bit-shift operations on the address to simplify the encoding.

5.3.3. Fault Localization

Next, we present our fault localization technique that aims to find the fault location in a given target function. Before delving into the details of the algorithm, we will explain the methodology of fault localization at a high level.

Methodology

At a high level, our fault localization technique uses a symbolic approach by reducing the fault localization problem into a constraint solving problem. In particular, we introduce a boolean variable for each line L, denoted by $\mathcal{B}[L]$, and encode the fault localization problem as an SMT formula, such that the value of the variable $\mathcal{B}[L]$ indicates whether line L is correct or not.

Checking faulty programs. To understand how to encode the fault localization problem, let us first explain how to encode the consistency check given a program $\mathcal{P}$ and a test case $t = (I, O)$. Specifically, the encoded SMT formula $\Phi(t)$ consists of three components:

1. *Semantic constraints.* For each line $L_i : s_i$, we generate a formula $\Phi_i(S, S')$ to describe the semantics of the statement s_i. Specifically, given a state S that holds before statement s_i, $\Phi_i(S, S')$ is valid if S' is the state after executing s_i. There are two parts of the constraint: the memory contents that are changed, and the memory contents that are preserved. For example, in case of an assignment statement, the

constraint will claim that 1) the evaluation result of the right value in state S equals to the left value in state S', and 2) all values except for the left value are the same in S and S'.

2. *Control flow integrity constraints.* In order to ensure all traces satisfying the constraint faithfully follow the control flow structure of a given program $\mathcal{P}$, we generate another set of formulae Φ_f. Specifically, we require that any line of code that is executed must have exactly one predecessor and one successor that are executed, and the branch condition in the code must be respected when picking the successor. This guarantees that there is exactly one valid execution trace corresponding to one test case.

3. *Consistency between program and test.* For the provided test case $t = (I, O)$, we also generate formula $\Phi_{in}(S_0, I)$ and $\Phi_{out}(S_n, O)$ to ensure the program behavior is consistent with the test. In particular, $\Phi_{in}(S_0, I)$ binds input I to the initial state S_0 and $\Phi_{out}(S_n, O)$ describes the connection between output O and final state S_n.

The satisfiability of formula $\Phi(t)$ indicates the result of consistency check. If $\Phi(t)$ is satisfiable, then program $\mathcal{P}$ can pass the test t, because there exists a valid trace according to the control flow and every pair of adjacent states in this trace is consistent with the semantics of the corresponding statement. Otherwise, if formula $\Phi(t)$ is unsatisfiable, then $\mathcal{P}$ fails the test t.

Now to check whether program $\mathcal{P}$ is faulty modulo a set of unit tests $\mathcal{E}$, we can conjoin the formula $\Phi(t_j)$ for each unit test $t_j \in \mathcal{E}$ and obtain the conjunction

$$\Phi = \bigwedge_{t_j \in \mathcal{E}} \Phi(t_j)$$

Here, the satisfiability of formula Φ indicates whether $\mathcal{P}$ is faulty modulo tests $\mathcal{E}$.

Methodology of fault localization. Let $\mathcal{P}$ be a faulty program modulo $\mathcal{E}$, we know the corresponding formula Φ for consistency check is unsatisfiable. Suppose the fault location is line L_i, one key insight is that replacing the semantic constraint $\Phi_i(S, S')$ with *true* yields a satisfiable formula. This is because *true* does not enforce any constraint between the pre-state S and post-state S', so a previously invalid trace caused by the bug at L becomes valid now.

Based on this insight, we develop a methodology to find the fault location using symbolic reasoning. Specifically, given a consistency check formula Φ, we can obtain a fault localization formula Φ' by replacing the semantic constraint $\Phi_i(S, S')$ with $\mathcal{B}[L_i] \rightarrow \Phi_i(S, S')$ for every line $L_i, i \in [1, n]$. Here, variable $\mathcal{B}[L_i]$ decides whether or not it turns the semantic constraint of L_i into *true*. Thus, $\mathcal{B}[L_i] =$ *false* indicates L_i is a fault location.

One hiccup here is that formula Φ' is always satisfiable and a model of Φ' can simply assign $\mathcal{B}[L_i] =$ *false* for all L_i. It means all lines in the program are fault locations, which is not useful for fault localization. To address this issue, we can add a cardinality constraint stating there are exactly K variables in map $\mathcal{B}$ that can be assigned to *false*, which forces the constraint solver to find exactly K fault locations in program $\mathcal{P}$.

Algorithm

Algorithm 2 Fault Localization

```
1: procedure LocalizeFault($\mathcal{P}, F, \mathcal{E}, \mathcal{V}$)
   Input: Program $\mathcal{P}$, function $F$, examples $\mathcal{E}$, visited map $\mathcal{V}$
   Output: Buggy line $L$ or $\perp$ to indicate failure
2:     $\mathcal{B} \leftarrow \{\}$; $\pi \leftarrow \{\}$; $M \leftarrow \{\}$;
3:     for $L \in dom(\mathcal{V})$ do
4:         if $\mathcal{V}[L] = true$ then
5:             $\mathcal{B}[L] \leftarrow true$;
6:     $\mathcal{S} \leftarrow \{F' \mapsto \mathsf{Summary}(F') \mid F' \in \mathsf{GetCallees}(\mathcal{P}, F)\}$;
7:     $\Phi \leftarrow$ Encode($\mathcal{B}, \mathcal{S}, \pi, M, F$);
8:     $\Phi \leftarrow \Phi \land \mathsf{ExampleConsistency}(\mathcal{P}, \mathcal{E})$;
9:     if $\mathsf{UNSAT}(\Phi)$ then return $\perp$;
10:    return $\mathsf{Filter}(\mathsf{IsFalse}, \mathsf{GetModel}(\Phi))$;
```

Having explained the high-level methodology, let us look at the detailed fault localization algorithm for network programs. As shown in Algorithm 2, the LocalizeFault procedure takes as input a program $\mathcal{P}$, a target function F, a set of tests $\mathcal{E}$, and a visited map $\mathcal{V}$, and produces as output the fault location in F that causes the behavior of $\mathcal{P}$ is not consistent with $\mathcal{E}$.

In the beginning, the LocalizeFault procedure initializes the boolean map $\mathcal{B}$ from visited map $\mathcal{V}$ (Lines 2 – 3). If line L is marked as visited by $\mathcal{V}$, then $\mathcal{B}[L]$ is initialized to *true* because L is not a fault location. Otherwise, $\mathcal{B}[L]$ does not have a determined value. The initialization also creates two empty maps π and M. Here, π is a mapping related to the encoding control flow integrity. It maps a line number L to a boolean variable $\pi[L]$, where $\pi[L] = true$ represents line L occurs in the trace. M maps a line L to an uninterpreted function $M[L]$, representing the memory after executing line L. In particular, we use an uninterpreted function to represent the memory, which takes an address x as input and produces as output the value stored at address x. Furthermore, since we need to maintain

$$\frac{}{M, L \vdash c \twoheadrightarrow c}\ (\text{Const}) \qquad \frac{M, L \vdash l \hookrightarrow \delta_l \quad l \in \{x, x.a, x[v]\}}{M, L \vdash l \twoheadrightarrow M[L](\delta_l)}\ (\text{LValue})$$

$$\frac{M, L \vdash e_i \twoheadrightarrow \psi_{e_i} \quad i = 1, \ldots, n}{M, L \vdash \mathit{op}(e_1, \ldots, e_n) \twoheadrightarrow \mathit{op}(\psi_{e_1}, \ldots, \psi_{e_n})}\ (\text{Op})$$

Figure 27: Inference rules for encoding expressions.

$$\frac{}{M, L \vdash x \hookrightarrow \text{Addr}(x)}\ (\text{Var})$$

$$\frac{\begin{array}{c} i = \text{offset}(a) \\ M, L \vdash x \hookrightarrow \delta_x \end{array}}{M, L \vdash x.a \hookrightarrow \delta_x + i}\ (\text{Field}) \qquad \frac{\begin{array}{c} M, L \vdash v \twoheadrightarrow i \\ M, L \vdash x \hookrightarrow \delta_x \end{array}}{M, L \vdash x[v] \hookrightarrow \delta_x + i}\ (\text{Array})$$

Figure 28: Inference rules for encoding the address of left-values.

multiple versions of the memory based on the execution status of each line in $\mathcal{P}$, we introduce a map M from line numbers to their corresponding uninterpreted functions.

Next, LocalizeFault computes function summaries for all callees of target function F and follows the methodology in Section 5.3.3 to localize fault based on symbolic reasoning. Specifically, it invokes the Encode procedure to generate semantic constraints and control flow integrity constraints (Line 7) and then invokes ExampleConsistency to generate consistency check for provided test $\mathcal{E}$ (Line 8). If the generated formula Φ is unsatisfiable, fault localization fails for target function F (Line 9). Otherwise, LocalizeFault returns the line L where the corresponding variable $\mathcal{B}[L] = \mathit{false}$ based on the model of Φ (Line 10).

Since the encoding for binding test cases to initial and final states is straightforward, we omit the discussion. In the remainder, we describe how to generate semantic constraint and control flow integrity constraint in more detail.

Expressions. Since the semantic constraint of a line involves encoding expressions, we first present the symbolic encoding of expressions in our network programs. The inference rules of generating constraints for expressions are summarized in Figure 27. A judgment of the form

$$M, L \vdash e \twoheadrightarrow \phi$$

denotes that the encoding of expression e is ϕ given memory map M and line number L. For example, the Const rule states that constant c is encoded as c. To encode a left-value l, including variable, field, and array accesses, we first need to obtain the address δ_l of l. Then according to the LValue rule, we look up the memory map M based on the current line number L and address δ_l to get its value $M[L](\delta_l)$. For an expression $op(e_1, \ldots, e_n)$, we can recursively encode sub-expression e_i as ψ_{e_i} and generate the composed encoding $op(\psi_{e_1}, \ldots, \psi_{e_n})$ (rule Op).

Similarly, inference rules of encoding addresses are summarized in Figure 28, where judgements of the form

$$M, L \vdash e \hookrightarrow \phi$$

denote the address of expression e is ϕ. Specifically, the address of variable x is simply obtained by the address operator (rule Var). The address of field access $x.a$ is $\delta_x + i$ where δ_x is the address of x and i is the offset of field a. Similarly, the address of array access $a[v]$ is $\delta_x + i$ where δ_x is the address of x and i is the symbolic encoding of immediate number v.

$$\frac{\begin{array}{c} M, L \vdash e \twoheadrightarrow \psi_e \quad M, L \vdash l \hookrightarrow \delta_l \\ L' = \text{PrevLine}(L, \pi, F) \quad \Phi_m \equiv \mathcal{B}[L] \to M[L](\delta_l) = \psi_e \\ \Phi_c \equiv \bigwedge_{z \neq \delta_l} M[L](z) = M[L'](z) \quad \Phi_f \equiv \pi[L+1] \wedge \pi[L'] \end{array}}{\mathcal{B}, \mathcal{S}, \pi, M, F \vdash L : l := e \rightsquigarrow \pi[L] \to \Phi_m \wedge \Phi_c \wedge \Phi_f} \quad (\text{Assign})$$

$$\frac{\begin{array}{c} M, L \vdash e \twoheadrightarrow \psi_e \quad L' = \text{PrevLine}(L, \pi, F) \\ \Phi_c \equiv \bigwedge_{z} M[L](z) = M[L'](z) \quad \Phi_f \equiv (\pi[L''] \vee \pi[L+1]) \wedge \pi[L'] \\ \Phi_m \equiv ((\mathcal{B}[L] \to \psi_e) \leftrightarrow \pi[L'']) \wedge ((\mathcal{B}[L] \to \neg\psi_e) \leftrightarrow \pi[L+1]) \end{array}}{\mathcal{B}, \mathcal{S}, \pi, M, F \vdash L : \textbf{jmp}\ (e)\ L'' \rightsquigarrow \pi[L] \to \Phi_m \wedge \Phi_c \wedge \Phi_f} \quad (\text{Jump})$$

$$\frac{\begin{array}{c} M, L \vdash x \twoheadrightarrow \psi_x \quad L' = \text{PrevLine}(L, \pi, F) \\ \Phi_f \equiv \pi[L+1] \wedge \pi[L'] \quad \Phi_c \equiv \bigwedge_{z} M[L](z) = M[L'](z) \wedge \mathit{DType}[\psi_x] = C \end{array}}{\mathcal{B}, \mathcal{S}, \pi, M, F \vdash L : x := \textbf{new}\ C \rightsquigarrow \pi[L] \to \Phi_c \wedge \Phi_f} \quad (\text{New})$$

$$\frac{\begin{array}{c} M, L \vdash v \twoheadrightarrow \psi_v \quad L' = \text{PrevLine}(L, \pi, F) \\ \Phi_m \equiv \mathcal{B}[L] \to M[L](\delta_{ret}) = \psi_v \quad \Phi_c \equiv \bigwedge_{z \neq \delta_{ret}} M[L](z) = M[L'](z) \quad \Phi_f \equiv \pi[L'] \end{array}}{\mathcal{B}, \mathcal{S}, \pi, M, F \vdash L : \textbf{ret}\ v \rightsquigarrow \pi[L] \to \Phi_m \wedge \Phi_c \wedge \Phi_f} \quad (\text{Return})$$

$$\frac{\begin{array}{c} M, L \vdash v_i \twoheadrightarrow \psi_i \quad i = 1, \ldots, n \quad L' = \text{PrevLine}(L, \pi, F) \\ \Phi_1 = \mathcal{S}[C.f][\psi_1/\mathit{arg}_1, \ldots, \psi_n/\mathit{arg}_n, x/\mathit{ret}, M[L']/M_{in}, M[L]/M_{out}] \\ \Phi_m \equiv \mathcal{B}[L] \to \Phi_1 \quad \Phi_f \equiv \pi[L+1] \wedge \pi[L'] \end{array}}{\mathcal{B}, \mathcal{S}, \pi, M, F \vdash L : x := C.f(v_1, \ldots, v_n) \rightsquigarrow \pi[L] \to \Phi_m \wedge \Phi_f} \quad (\text{SCall})$$

$$\frac{\begin{array}{c} M, L \vdash v_i \twoheadrightarrow \psi_i \quad i = 1, \ldots, n \quad M, L \vdash y \twoheadrightarrow \psi_y \quad L' = \text{PrevLine}(L, \pi, F) \\ \Phi_{S_j} \equiv \mathcal{S}[C_j.f][\psi_y/\mathit{this}, \psi_1/\mathit{arg}_1, \ldots, \psi_n/\mathit{arg}_n, x/\mathit{ret}, M[L']/M_{in}, M[L]/M_{out}] \\ \Phi_1 \equiv \bigwedge_{j=1,\ldots,m} \mathit{DType}[\psi_y] = C_j \to \Phi_{S_j} \quad C_j <: \text{Class}(f) \quad j = 1, \ldots, m \\ \Phi_m \equiv \mathcal{B}[L] \to \Phi_1 \quad \Phi_f \equiv \pi[L+1] \wedge \pi[L'] \end{array}}{\mathcal{B}, \mathcal{S}, \pi, M, F \vdash L : x := y.f(v_1, \ldots, v_n) \rightsquigarrow \pi[L] \to \Phi_m \wedge \Phi_f} \quad (\text{VCall})$$

$$\frac{\mathcal{B}, \mathcal{S}, \pi, M, F \vdash L_i : s_i \rightsquigarrow \Phi_i \quad i = 1, \ldots, n}{\mathcal{B}, \mathcal{S}, \pi, M, F \vdash L_1 : s_1; \ldots; L_n : s_n \rightsquigarrow \Phi_1 \wedge \ldots \wedge \Phi_n} \quad (\text{Compose})$$

Figure 29: Inference rules for generating semantic constraints and control flow integrity constraints. Given a function f with n arguments, the formal parameters of f are denoted by $\mathit{arg}_1, \ldots, \mathit{arg}_n$. The return variable is denoted by *ret*, and the reference variable is denoted by *this*.

Statements. Having explained the constraint generation for expressions, now let us illustrate how to generate constraints for statements. As shown in Figure 29, our inference rules for statement-level constraints take judgments of the form

$$\mathcal{B}, \mathcal{S}, \pi, M, F \vdash L : s \rightsquigarrow \Phi$$

meaning that the statement s at line L is encoded as formula Φ under line indicator map $\mathcal{B}$, summary map $\mathcal{S}$, trace selector map π, memory map M, and target function F. For ease of illustration, we divide the final constraints into three parts: Φ_m denotes the semantic constraint about the "maybe incorrect" operations in the statement, which typically involves updating the memory. Φ_c represents the semantic constraint about the "always correct" operations in the statement, which usually describes what memory values should remain unchanged by the execution of the statement. Φ_f is the control flow integrity constraint that characterizes a valid trace based on the control flow structure of function F. The output of the PrevLine function is a symbolic value dependent on the assignment of the trace selector map π.

Assign statement. Given an assign statement $l := e$ at line L, the Assign rule generates a formula $\pi[L] \rightarrow \Phi_m \wedge \Phi_c \wedge \Phi_f$, which means if line L is selected in the trace, then all three kinds of constraints should hold. Specifically, it first computes the address of left-hand side l as δ_l and computes the expression encoding of right-hand side e as ψ_e. The generated "maybe incorrect" constraint adds a guard $\mathcal{B}[L]$ to memory update $M[F][a]$, saying if the line is not the fault location, then the value at address δ_l after executing line L is ψ_e. However, if line L is the fault location, then no constraint is added effectively. The "always correct" constraint states all values except the one at δ_l should be preserved by the assign statement. The control flow integrity constraint Φ_f says that both previous line L' and next line $L + 1$ must be selected in the trace.

Jump statement. Similar to assign statements, the Jump rule also emits three constraints if the statement at line L is a conditional jump statement **jmp** (e) L''. Since the jump condition e might be faulty, the rule adds a guard $\mathcal{B}[L]$ to the encoded expression ψ_e. Φ_m says if line L is not the fault location, then the jump destination L'' is selected in the trace if condition evaluates to *true* or the next line L is selected in the trace if condition e evaluates to *false*. Furthermore, the control flow integrity constraint Φ_f requires that (1) the previous line L' must occur in the trace, and (2) either the next line $L + 1$ or the jump destination L'' must occur in the trace. In addition, since a jump statement does not write to the memory, Φ_c describes all values in the previous memory $M[L']$ are preserved in current memory $M[L]$.

New statement. Since we do not consider memory allocation as a source of bugs, the New rule does not generate the "maybe incorrect" constraint Φ_m. Instead, given a statement $x :=$ **new** C, it generates Φ_c stating the dynamic type of x is C and all values in the member are preserved. In addition, the previous line L' and next line $L + 1$ must occur in the trace.

Return statement. Given a return statement **ret** v at line L, the Return rule first evaluates immediate number v to ψ_v, and then write the value to memory $M[L]$ at location δ_{ret}, where *ret* is an implicit variable for storing return values and δ_{ret} is its address. Since the return value could be faulty, the rule adds a guard $\mathcal{B}[L]$ in constraint Φ_m. By contrast, all other values are considered correct, so it preserves all but the return value after execution in constraint Φ_c. In addition, the control flow integrity constraint Φ_f only requires the previous line to occur in the trace.

Static call. Since our fault localization algorithm is modular and summaries are computed for all functions in the program, we can directly utilize the function summary for invocations. In particular, given a static function call $x := C.f(v_1, \ldots, v_n)$, the SCall rule evaluates the actual parameter v_i to ψ_i and substitutes the formal parameter arg_i in summary $\mathcal{S}[C.f]$ with ψ_i. Furthermore, it also substitutes the return variable *ret* with variable x and substitutes the formal input memory M_{in} and output memory M_{out} with the actual memories $M[L']$

and $M[L]$, respectively.

Virtual call. The VCall rule for virtual function calls is similar to SCall. The only difference is that it needs to dispatch function summaries based on the receiver object. Recall that every time the program creates a new object, the New rule stores its dynamic type in the *DType* map. Thus, given a virtual call $x := y.f(v_1, \dots, v_n)$, SCall can obtain the dynamic type of receiver object y by evaluating y to ψ_y and looking up the map *DType*. According to the dynamic type $\mathit{DType}[\psi_y]$, SCall selects the appropriate function summary to use.

Statement composition. Finally, the Compose rule is quite straightforward. Specifically, the constraints generated for multiple statements are obtained inductively by conjoining the constraints for each individual statement.

Since we encode each statement and each branch option between statements rather than entire execution paths, the total encoded formula's size is linear to the program's size.

5.3.4. Patch Synthesis

The last step of our repair algorithm is to generate a patch to fix the faulty program. The high-level idea is to reduce the patch generation problem to an expression synthesis problem. Specifically, given a faulty function F in program $\mathcal{P}$ and the fault location L, we generate a sketch by replacing the line L with a hole and complete the sketch based on the given tests. As shown in Algorithm 3, our patch synthesis algorithm consists of three steps: (1) introducing a hole at the fault location of program $\mathcal{P}$ to obtain a sketch Ω, (2) generating a context-free grammar $\mathcal{G}$ to capture the search space for the expression to fill in the hole, and (3) completing the sketch Ω by finding a correct expression accepted by $\mathcal{G}$. In what follows, we describe each of the steps in more details.

Algorithm 3 Patch Synthesis

```
procedure SynthesizePatch($\mathcal{P}, \mathcal{E}, F, L$)
  **Input:** Program $\mathcal{P}$, examples $\mathcal{E}$, faulty function $F$, faulty line $L$
  **Output:** Repaired program $\mathcal{P}'$ or $\perp$ to indicate failure
    $\Omega \leftarrow$ IntroduceHole($\mathcal{P}, F, L$);
    $\mathcal{G} \leftarrow$ GenerateGrammar($\Omega$);
    return CompleteSketch($\Omega, \mathcal{G}, \mathcal{E}$);
```

Hole introduction. To generate a sketch from the original program and target function F, we replace the *maximal* expressions at the fault location in F with a hole. The maximal expressions to be considered are determined by the kind of the faulty statement. In particular, we introduce holes for the right-hand-side expressions of assignments, conditional expressions of jump statements, return values of return statements, and functions and arguments for function invocations. Replacing these expressions with holes turns the original program into a sketch and reduces the repair problem into a problem that aims to find a correct expression to instantiate the hole.

Search space generation. After the hole is generated in the sketch, we still need to determine the search space for candidate expressions that can potentially result in a correct patch. The key challenge for defining the search space is to ensure the search space indeed includes the correct patch expression. To address this challenge, we define a context-free grammar as shown in Figure 30, which includes all expressions that are constructed using constants, variables, field accesses, function invocations, unary and binary operators. While it is possible to obtain a fixed grammar that contains a comprehensive set of constructs that are useful for many network programs, the search space of such grammar may become unnecessarily large for a particular program. To solve this problem, we parameterize all constants, variables, fields, functions, and operators over the sketch and only instantiate constructs that are in scope. For example, given a particular sketch with a hole, we only

$$\begin{array}{rcl} \mathit{Expr} & ::= & \mathit{UnaryOp}(\mathit{Expr}) \;\mid\; \mathit{BinaryOp}(\mathit{Expr}, \mathit{Expr}) \\ & \mid & v.f(\mathit{Expr}, \ldots, \mathit{Expr}) \;\mid\; v.\mathit{Expr} \;\mid\; v \;\mid\; c \end{array}$$

$$v \in \textbf{Variables} \quad c \in \textbf{Constants} \quad f \in \textbf{Functions}$$

Figure 30: Grammar for completions of sketch holes.

populate the variable set with all local and global variables that are in scope of the hole. As another example, if the hole corresponds to the conditional expression of a if statement, we only add logical operators to the grammar.

Consider again the isSameAs function from our motivating example in Figure 24.

```
1  public class FirewallRule {
2    public MacAddress dl_dst; public boolean any_dl_dst;
3    public boolean isSameAs(FirewallRule r) {
4      if (... || any_dl_dst != r.any_dl_dst
5              || (any_dl_dst == false &&
6                  dl_dst != r.dl_dst))
7          return false;
8      return true;
9    }
10 }
```

Suppose the fault localization procedure finds the fault location is Line 6, we obtain the sketch

```
public boolean isSameAs(FirewallRule r) {
  if (... || any_dl_dst != r.any_dl_dst
          || (any_dl_dst == false && ?? ))
      return false;
  return true;
}
```

where ?? denotes a hole in the generated sketch. The search space of expressions for filling in the hole ?? can be described by the following context-free grammar with start symbol *Expr*

$$\begin{array}{rcl} \mathit{Expr} & ::= & \mathit{true} \;\mid\; \mathit{false} \;\mid\; \mathit{any\_dl\_dst} \;\mid\; r.\mathit{any\_dl\_dst} \\ & \mid & \mathit{any\_dl\_dst} == r.\mathit{any\_dl\_dst} \;\mid\; \mathit{any\_dl\_dst.equals}(r.\mathit{any\_dl\_dst}) \\ & \mid & !\mathit{Expr} \;\mid\; \mathit{Expr}\&\&\mathit{Expr} \;\mid\; \mathit{Expr}||\mathit{Expr} \end{array}$$

Sketch completion. Given a program sketch Ω with a context-free grammar $\mathcal{G}$ for its hole, the goal of sketch completion is to find an expression accepted by $\mathcal{G}$ such that the program obtained by replacing the hole with this expression can pass the given tests $\mathcal{E}$. To solve the sketch completion problem, we use a top-down synthesis approach and perform depth-first search in the space of expressions generated by the grammar $\mathcal{G}$ to find the correct expression. The algorithm is summarized in Algorithm 4.

Algorithm 4 Sketch Completion

```
procedure $\textsc{CompleteSketch}(\Omega, \mathcal{G}, \mathcal{E})$
Input: Sketch $\Omega$, grammar $\mathcal{G}$, examples $\mathcal{E}$
Hyperparameter: Maximum number of expansions $K$
Output: Completed sketch $\Omega'$ or $\bot$ to indicate failure
    if $\mathsf{ExpansionNum}(\Omega, \mathcal{G}) > K$ then return $\bot$;
    if $\mathsf{IsCompleteProgram}(\Omega)$ then
        if $\mathsf{Verify}(\Omega, \mathcal{E})$ then return $\Omega$;
        else return $\bot$;
    $N \leftarrow \mathsf{GetANonTerminal}(\Omega)$;
    for each production $\alpha ::= \beta \in \mathsf{Productions}(\mathcal{G})$ do
        if $\alpha = N$ then
            $\Omega' \leftarrow \mathsf{Expand}(\Omega, N, \beta)$;
            $\Omega^* \leftarrow \textsc{CompleteSketch}(\Omega', \mathcal{G}, \mathcal{E})$;
            if $\Omega^* \neq \bot$ then return $\Omega^*$;
    return $\bot$;
```

At a high level, the CompleteSketch procedure in Algorithm 4 starts with a sketch obtained by replacing the hole with the root non-terminal of grammar $\mathcal{G}$. The procedure progressively expands the non-terminal based on productions in $\mathcal{G}$ until it finds a sketch without any non-terminal (i.e., a complete program) that can pass the tests $\mathcal{E}$. Since there could be recursive production rules that make infinite number of candidate programs accepted by the grammar, the sketch completion procedure may not terminate. To resolve this issue, we count the number of expansions used to obtain a sketch or complete program and put a limit on the maximum number of expansions. In this way, the CompleteSketch procedure is guaranteed to terminate in finite time.

As shown in Algorithm 4, the CompleteSketch procedure first checks the current number of expansions and immediately returns $\bot$ if the number exceeds the predefined hyper-parameter K (Line 2). Next, it checks the termination condition of the recursion (Line 3), i.e. whether Ω is a complete program. If Ω is complete or does not contain any non-terminal, CompleteSketch executes the tests $\mathcal{E}$ on Ω. If Ω indeed passes the tests $\mathcal{E}$, sketch completion succeeds with Ω (Line 4); otherwise, the procedure returns $\bot$ indicating failure (Line 5). If sketch Ω has at least one non-terminal symbol, CompleteSketch obtains the next non-terminal symbol N to expand (Line 6). Then it enters a loop (Lines 7 – 11) and enumerates all productions in $\mathcal{G}$ where the left-hand side of the production is N. In particular, for each production $N ::= \beta$, CompleteSketch obtains a new sketch Ω' from Ω by replacing the non-terminal N with β (Line 9) and recursively invokes itself with the new sketch Ω' (Line 10). If a correct completion Ω^* is found by the recursive call, then the caller CompleteSketch also returns Ω^* (Line 11); otherwise, it moves on to the next production. This process is repeated until all productions in $\mathcal{G}$ are checked. If no correct completion exists, CompleteSketch returns $\bot$ to indicate failure (Line 12).

[Soundness of sketch completion] Given a sketch Ω, a grammar $\mathcal{G}$ for expressions to fill in the hole in Ω, and a set of unit tests $\mathcal{E}$, let Ω' be the return value of CompleteSketch$(\Omega, \mathcal{G}, \mathcal{E})$. If $\Omega' \neq \bot$, then Ω' can pass all unit tests in $\mathcal{E}$.

[Completeness of sketch completion] Given a sketch Ω, a grammar $\mathcal{G}$, a set of unit tests $\mathcal{E}$, and a hyper-parameter K, if CompleteSketch$(\Omega, \mathcal{G}, \mathcal{E}) = \bot$, then there does not exist an expression e accepted by $\mathcal{G}$ such that (1) the number of expansions from the root symbol of $\mathcal{G}$ to e is no more than K, and (2) substituting the hole in Ω with e results in a program that passes all unit tests in $\mathcal{E}$.

We will skip the proof of these theorems in this document.

5.3.5. Implementation

We have implemented the proposed repair technique in a tool called Orion. Orion leverages the Soot static analysis framework [51] to convert Java programs into Jimple code, which provides a succinct yet expressive set of instructions for analysis. In addition, Orion utilizes the Rosette tool [52] to perform symbolic reasoning for fault localization and patch synthesis. While our implementation closely follows the algorithm presented in Section 5.3, we also conduct several optimizations that are important to improve the performance of Orion.

Validating patches with local specifications. Observe that the patch synthesis procedure could potentially validate a large number of candidate patches, it is crucial to optimize the validation procedure of Orion for better performance. Our key idea is to validate the correctness of a candidate patch based on the pre- and post-condition of fault locations, rather than executing the tests from the beginning of the program. Specifically, for each provided unit test, Orion symbolically executes the network program and infers the pre- and post-states of the faulty line in the process of fault localization. Then in the patch synthesis phase, Orion can execute each candidate patch from the inferred pre-state and check if the execution result is consistent with the inferred post-state. This validation procedure enables fast checking of each candidate patch, because it avoids repeated symbolic execution of correct statements and only executes those patched statements.

Memories for different types. Since the conversion between bitvectors and integers imposes significant overhead on running time, Orion divides the memory into two parts, one for integers and the other for bitvectors. In this design, Orion automatically selects the memory chunk based on the variable types. In particular, it only stores integer values in the integer memory and likewise bitvectors in the bitvector memory. This optimization significantly improves the performance of symbolic reasoning, because there is no need to convert between different data types.

Stack and heap. In order to reduce the number of memory operations, ORION also divides the memory into stack and heap. As is standard, stack only stores static data and its layout is deterministic. For example, the locations of function arguments and return values are fixed and statically available, so they are stored on stack. Therefore, stacks are implemented using fixed-size vectors, and thus can be efficiently accessed for read and write operations. On the other hand, heap stores dynamic data that are usually not known at compile time, such as allocated objects. Since the heap size cannot be determined beforehand, ORION uses an uninterpreted function $f(x)$ to represent heaps, where x is the address and $f(x)$ is the value stored at x.

String values. Since reasoning over string values is a challenging task and not always necessary for repairing network programs, we simplified the representation of strings with integer values. Specifically, ORION maps each string literal to a unique integer and represent all string operations (e.g. concatenation) with uninterpreted functions. While many existing techniques [53; 54] can improve the precision of string analysis, we find our current approximation is sufficient for repairing network programs in our experimental evaluation.

Bounded program analysis. In order to improve the repair time, ORION only performs bounded program analysis for fault localization and patch synthesis. Namely, we unroll loops and inline functions up to K times, where K is a predefined hyper-parameter. In this way, function summaries can be easily and efficiently computed using symbolic execution. While it is possible to incorporate invariant inference and recursive function summarization techniques, we do not implement them in the current version of ORION and leave it as future work.

5.4. Evaluation

To evaluate the proposed techniques, we perform experiments that are designed to answer the following research questions:

RQ1 Is Orion effective to repair realistic network programs?

RQ2 How efficient are the fault localization and repair techniques in Orion?

RQ3 How helpful are modular analysis and domain-specific abstraction for repairing network programs?

RQ4 How does Orion perform compared to other repair tools for Java programs?

Benchmark collection To obtain realistic benchmarks, we crawl the commit history of Floodlight [50], an open-source SDN controller that supports the OpenFlow protocol, and identify commits based on the following criteria:

1. The commit message contains keywords about repairing bugs, e.g., "bug", "error", "fix";

2. The commit changes no more than three lines of code.

These criteria are important because they are able to distinguish commits caused by bug repairs from those generated for non-repair scenarios, such as code refactoring, adding functionalities, etc. Following these criteria, we have collected 10 commits from the Floodlight repository and adapted them into our benchmarks. Specifically, given a commit in the repository, we take the code before the commit as the faulty network program and the version after the commit as the ground-truth repaired program. The code is post-processed and the parts irrelevant to the bug of interest are removed. We also identify corresponding unit tests and modify them to directly reveal the bug as appropriate. Each benchmark in our evaluation consists of a faulty network program and its corresponding unit tests.

ID	Module	LOC	# Funcs	# Tests	Succ	Exp	Loc Time (s)	Synth Time (s)	Total Time (s)
1	DHCP	212	17	2	Yes	Yes	40	117	157
2	Load Balancer	336	28	2	No	No	-	-	-
3	Firewall	262	13	2	Yes	Yes	893	197	1090
4	DHCP	431	32	2	Yes	Yes	95	39	134
5	Utility	809	65	2	No	No	-	-	-
6	Routing	605	44	3	Yes	Yes	271	179	450
7	Utility	454	45	2	Yes	Yes	39	46	85
8	Learning Switch	738	34	2	Yes	No	571	595	1166
9	Database	442	17	2	Yes	No	310	2139	2449
10	Link Discovery	671	46	2	Yes	No	268	158	426

Table 3: Experimental results of Orion.

Experimental setup All experiments are conducted on a computer with 4-core 2.80GHz CPU and 16GB of physical memory, running the Arch Linux Operating system. We use Racket v7.7 as the compiler and runtime system of Orion and set a time limit of 1 hour for each benchmark.

5.4.1. Main Results

Our main experimental results are summarized in Table 3. The column labeled "Module" in the table describes the network module to which the benchmark belong. The next two columns labeled "LOC" and "# Funcs" show the number of lines of source code (in Jimple) and the number of functions, respectively. The "# Tests" column presents the number of unit tests used for fault localization and patch synthesis. Next, the "Succ" and "Exp" columns show whether Orion can successfully repair the program and if the generated patch is exactly the same as the ground-truth. Since Orion returns the first fix that can pass all provided test cases, the repaired programs are not necessarily the same as those expected in the ground-truth. In this case, the table will show a "Yes" in the "Succ" column and a "No" in the "Exp" column. Finally, the last three columns in Table 3 denote the fault localization time, patch synthesis time and the total running time of Orion.

As shown in Table 3, there is a range of 13 to 65 functions in each benchmark and the average number of functions is 34 across all benchmarks. Each benchmark has 212 – 809 lines of Jimple code, with the average being 496. Orion succeeds in repairing 8 out of

10 benchmarks. Furthermore, for 5 benchmarks that can be successfully repaired, Orion is able to generate exactly the same fix as ground-truth. By a manual examination, it is possible to construct adversarial inputs other than the unit tests that the rest 3 fixes can not properly handle.

The table also shows that only 2 to 3 examples are required to specify a successful fix. This can be explained by that the unit tests are specifically designed to reveal the bugs and that our strict encoding method is suitable for reasoning about the connection between input/output examples and the buggy statement. Given that our benchmarks cover programs from a variety of modules of Floodlight, such as DHCP Server, Firewall, etc, we believe that Orion is effective to repair realistic network programs (RQ1).

To understand why Orion is not able to repair benchmark 2 and 5, we manually inspect the corresponding network programs and the execution logs. We found Orion failed to localize the fault of benchmark 2 due to its incomplete support for the hash map data structure. Ideally, the hash map should be modeled as an unbounded data structure with dynamic allocation, which is beyond the capability of our current symbolic analysis. For Benchmark 5, Orion is able to localize the fault but not able to synthesize the correct patch. The expected patch for benchmark 5 requires replacing an invocation of a function with side effects with another function, which is out of the ability of Orion's patch synthesizer.

Regarding the efficiency, Orion can repair 8 benchmarks in an average of 744 seconds with only 2 to 3 test cases. The fault localization time ranges from 39 seconds to 893 seconds, with 50% of the benchmarks within five minutes. The patch synthesis time ranges from 39 seconds to 2139 seconds, with 60% of the benchmarks within five minutes. In summary, the evaluation results show that Orion only takes minutes to localize bugs in a faulty program and synthesize a correct patch based on two to three unit tests (RQ2).

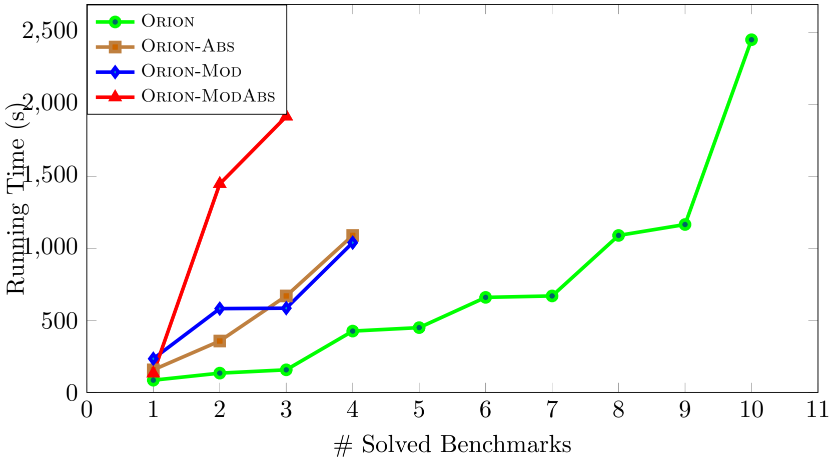


Figure 31: Comparing Orion against three variants.

5.4.2. Ablation Study

To explore the impact of modular analysis and domain-specific abstraction on the proposed repair technique, we develop three variants of Orion:

- Orion-NoMod is a variant of Orion without modular analysis. Specifically, given a faulty network program P, Orion-NoMod inlines the functions in P but still uses abstractions for network data structures for fault localization and patch synthesis. It does not compute or reuse function summaries for symbolic reasoning.

- Orion-NoAbs is a variant of Orion without domain-specific abstraction. In particular, Orion-NoAbs does not use abstractions for any function in network data structures. Instead, it uses the original concrete implementation of those functions for symbolic reasoning. If the implementation is written in a different language, we manually translate the implementation to Java.

- Orion-NoModAbs is a variant of Orion without modular analysis or domain-specific abstraction. Essentially, Orion-NoModAbs simply inlines all functions in the faulty program, including those functions in the network data structures, and performs symbolic analysis for fault localization and patch synthesis.

Tool	Expected	Succeed but unexpected	Failed	Total
Orion	5	3	2	10
Jaid	2	6	2	10

Table 4: Comparison between Orion and Jaid.

To understand the impact of modular analysis and domain-specific abstraction, we run all variants on the 10 collected benchmarks. For each variant, we measure the total running time (including time for fault localization and time for patch synthesis) on each benchmark, and order the results by running time in increasing order. The results for all variants are depicted in Figure 31. All lines stop at the last benchmark that the corresponding variant can solve within 1 hour time limit.

As shown in Figure 31, Orion-NoAbs can only solve 4 out of 10 benchmarks in the evaluation, with the average running time being 569 seconds. Similarly, Orion-NoMod is only able to solve 4 out of 10 benchmarks within the 1 hour time limit, and the average running time is 610 seconds. Among all different variants, Orion-NoModAbs solves the least number of benchmarks: 3 out of 10. For the ones that it can solve, the average running time is 1165 seconds. This experiment shows that modular analysis and domain-specific abstraction are both great boost to Orion's efficiency to repair network programs (RQ3).

5.4.3. Comparison with the Baseline

To understand how Orion performs compared to other Java program repair tools, we compare Orion against a state-of-the-art tool called Jaid [55] on our benchmarks. Specifically, Jaid takes as input a faulty Java program, a set of unit tests, and a function signature for fault localization and patch synthesis. Note that Jaid solves a simpler repair problem than Orion, because it requires the user to specify a function that is potentially incorrect in the program, whereas Orion does not need input other than the faulty program and unit tests. In order to run Jaid on our benchmarks, we adjust the format of our benchmarks into Jaid's format and provide the faulty function (known from the ground truth) as input for Jaid.

The comparison results are summarized in Table 4. As we can see from the table, JAID is able to generate valid fixes with respect to the test cases for 8 out of 10 benchmarks. But after a manual inspection, we find that only 2 of them are the same as those shown in the ground-truth. For the remaining two benchmarks, JAID fails to generate a valid patch. In particular, JAID exceeds the time limit for one benchmark and runs out of memory for the other.

It is not feasible to reasonably compare the running time between JAID and ORION, because JAID is not designed to stop after finding the first valid fix. Instead, it will generate a large number of candidate patches and output a ranked list of valid ones among them, which takes excessively long to eventually finish.

ORION outperforms JAID in terms of repairing accuracy. In particular, ORION is able to repair the same number of benchmarks as JAID and find the expected fix among five of them, whereas JAID is only able to find the expected fix on two.

In addition, ORION outputs the first valid patch it finds, while JAID may produce hundreds or thousands of candidate patches, which requires extra ranking heuristics. This difference can be explained by the amount of semantic information used by each tool. Specifically, JAID monitors a selected set of states chosen by dynamic semantic analysis, and localizes potential bug locations by a heuristic-based ranking algorithm over the values of states collected through the execution of test cases. Like similar preceding systems, this method relies on matching the ranking algorithm's heuristics with specific tasks, as well as a number of test cases to generate enough state information to use. ORION strictly encodes the semantic information of the entire program and infers the bug location as well as the specification for the patch from this encoding. Therefore, ORION is less likely to overfit to specific test cases or algorithm heuristics.

In summary, ORION is more effective in automatically fixing bugs in network programs compared to state-of-the-art repairing tools for Java programs, especially with respect to repairing accuracy and avoiding overfitting (RQ4).

5.5. Limitations

We discuss several limitations of Orion that we plan to improve in future work. First, Orion repairs the faulty network program with the first correct patch that can pass all provided unit tests, which may not be the fix intended by the user. The problem can be addressed by introducing a user interaction that resumes the synthesis procedure after finding the first correct patch, in case it is not intended by the user or a more formal specification.

Second, patches that require complicated changes, e.g., those involving control flow structures, more than one bug locations, or more than 3 lines of changes, are beyond Orion's ability to repair. They make up 44% of our collection of bug-fixing commits. Specifying these major changes of the program in enough detail may require more than input/output examples. Searching for the patch is also an excessively time-consuming procedure. One possible solution is to ask for more detailed specifications such as network-wide invariants from users. Another approach is to introduce some a priori knowledge about the program, such as searching over a domain-specific language for edits.

Third, in order to force symbolic execution to terminate in finite time, Orion currently unrolls all loops in the network program, which may result in missing a potential bug. Loop invariant inference techniques can be leveraged to overcome this challenge and still guarantee termination.

5.6. Takeaways

In this section, we demonstrate how general-purpose control programs in networks can be automatically updated. We have proposed an automated repair technique for network controller programs with unit tests as specifications. Our technique internally performs symbolic reasoning for bug localization and patch synthesis, optimized by network domain-specific abstractions and modular analysis to reduce encoding size. we have implemented a tool called Orionand evaluated it on 10 benchmarks adapted from the Floodlight framework. The experimental results demonstrate that Orion is effective for repairing realistic network programs with moderate change sizes.

CHAPTER 6

Future Work

There are a number of new interesting topics to be explored along the line of our existing work. We envision the following problems can be explored to increase the performance and completeness of the existing systems.

6.1. Learning from mixed traffic types

The traffic classification rule synthesis system currently relies on users to prepare part of the training data. Realistic network traffic typically contains a mixture of multiple types of target traffic. One application can also have multiple functionalities that look distinct from the perspective of network traffic. It would be interesting to explore new methods to train the synthesizer in these scenarios. One possibility is to turn it into an unsupervised learning system. The learned rules can play the role as the boundary of different clusters. Alternatively, we can try to discover multiple weak rules that are unified by another statistical model such as a decision tree. They together play the role of a classifier.

6.2. Network-wide Invariants as Specification

Correctness of the network is often specified by a list of properties the entire network's configuration must guarantee. Specifications in unit tests or counter-examples we currently support with the automatic debugging system are indirect and unsound. The functionality can be enhanced by exploring ways the existing debugging method can directly interact with the invariants specifying the control plane program's behaviors. Given this ability, the debugger will achieve the following two new functions: (1) debugging a control plane program to comply with a newly added property, (2) debugging a control plane program by given examples with guarantee that existing properties won't be broken by the change.

Notice that our debugger wants the SMT solver to give a valid assignment of selector variables as well as the program's intermediate states. Since input/output is essentially a part of intermediate states associated with the rest by the semantics, directly using a logical constraint with quantifiers over input/output would require quantifiers on every intermediate state of the program, so that a valid assignment of the selectors is feasible. This formula is unlikely to be solvable efficiently.

One possible idea is to do the debugging in a counter-example driven manner. A verification procedure is repeatedly invoked to generate inputs whose corresponding outputs from the control plane program violate the invariants. Then the debugger is invoked to generated a program that satisfy the invariants only for the counter-example inputs, so that the quantifiers can be eliminated. The process repeats until the verification passes and we'll get a fixed program.

6.3. Abstraction Refinement for Bug Localization

There is also great potential to improve the scalability of the debugging procedure. Ideally, we want the debugger to only analyze one function in the control plane program at a time, so that the SMT solver will only need to solve problems of small sizes regardless of the total size of the program. That is to say, the analysis is modular. This imposes two requirements on the debugger: (1) the module(function) under analysis must have a specification of constant size, which can already be supported with our local specification inference technique, and (2) all sub-modules(sub-functions) used by this module must have an interface of constant size. The second requirement is impossible to achieve without losing accuracy. A good abstract representation of the sub-function could possibly give us a balance between performance and accuracy. As the debugging goes deeper into sub-functions, we can gradually concretize these abstract sub-functions, which is close to the concept of abstraction refinement.

BIBLIOGRAPHY

[1] Xiao Liu, Brett Holden, and Dinghao Wu. Automated synthesis of access control lists. In *2017 International Conference on Software Security and Assurance (ICSSA)*, pages 104–109. IEEE, 2017.

[2] Shambwaditya Saha, Santhosh Prabhu, and P Madhusudan. Netgen: Synthesizing data-plane configurations for network policies. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, pages 1–6, 2015.

[3] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 579–594, 2018.

[4] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Genesis: Synthesizing forwarding tables in multi-tenant networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 572–585, 2017.

[5] Yifei Yuan, Rajeev Alur, and Boon Thau Loo. Netegg: Programming network policies by examples. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, pages 1–7, 2014.

[6] Lei Shi, Yahui Li, Boon Thau Loo, and Rajeev Alur. Network traffic classification by program synthesis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 430–448. Springer, 2021.

[7] Lei Shi, Yuepeng Wang, Rajeev Alur, and Boon Thau Loo. Automatic repair for network programs. In *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part II*, volume 13244, page 353. Springer Nature, 2022.

[8] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. *Acm sigplan notices*, 49(1):113–126, 2014.

[9] Broadcom-Switch. Broadcom-switch/of-dpa: Openflow data plane abstraction, 2022. [Online; accessed 20-April-2022].

[10] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A language for provisioning network resources. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 213–226, 2014.

[11] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak,

and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8. IEEE, 2013.

[12] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

[13] Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In *ACM SIGPLAN Notices*, volume 50, pages 107–126. ACM, 2015.

[14] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. In *ACM SIGPLAN Notices*, volume 52, pages 452–466. ACM, 2017.

[15] Sunbeom So and Hakjoo Oh. Synthesizing imperative programs from examples guided by static analysis. In *International Static Analysis Symposium*, pages 364–381. Springer, 2017.

[16] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices*, 50(6):619–630, 2015.

[17] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.

[18] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 436–449. ACM, 2018.

[19] Soumya Maity, Padmalochan Bera, and SK Ghosh. Policy based acl configuration synthesis in enterprise networks: A formal approach. In *2012 International Symposium on Electronic System Design (ISED)*, pages 314–318. IEEE, 2012.

[20] Eric Hayden Campbell, William T Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. Avenir: Managing data plane diversity with control plane synthesis. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 133–153, 2021.

[21] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: syntax- and semantic-guided repair synthesis via programming by examples. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the Joint Meeting on Foundations of Software Engineering, (ESEC/FSE)*, pages 593–604. ACM, 2017.

[22] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 88–99. IEEE Computer Society, 2009.

[23] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89–98, 2007.

[24] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for java. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 528–550. Springer, 2005.

[25] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE)*, pages 30–39. IEEE Computer Society, 2003.

[26] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric A. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 595–604. IEEE Computer Society, 2002.

[27] James A. Jones, Mary Jean Harrold, and John T. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 467–477. ACM, 2002.

[28] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 437–446. ACM, 2011.

[29] Manu Jose and Rupak Majumdar. Bug-assist: Assisting fault localization in ANSI-C programs. In *Proceedings of International Conference on Computer Aided Verification (CAV)*, volume 6806 of *LNCS*, pages 504–509. Springer, 2011.

[30] Andreas Griesmayer, Roderick Bloem, and Byron Cook. Repair of boolean programs with an application to c. In *International Conference on Computer Aided Verification*, pages 358–371. Springer, 2006.

[31] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Automated network repair with meta provenance. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, pages 26:1–26:7. ACM, 2015.

[32] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Automated bug removal for software-defined networks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 719–733. USENIX Association, 2017.

[33] Hossein Hojjat, Philipp Rümmer, Jedidiah McClurg, Pavol Cerný, and Nate Foster. Optimizing horn solvers for network repair. In Ruzica Piskac and Muralidhar Talupur,

editors, *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD)*, pages 73–80. IEEE, 2016.

[34] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: an ensemble of autoencoders for online network intrusion detection. *arXiv preprint arXiv:1802.09089*, 2018.

[35] Preeti Mishra, Vijay Varadharajan, Uday Tupakula, and Emmanuel S Pilli. A detailed investigation and analysis of using machine learning techniques for intrusion detection. *IEEE Communications Surveys & Tutorials*, 21(1):686–728, 2018.

[36] Jun Zhang, Xiao Chen, Yang Xiang, Wanlei Zhou, and Jie Wu. Robust network traffic classification. *IEEE/ACM Transactions on Networking (TON)*, 23(4):1257–1270, 2015.

[37] Guowu Xie, Marios Iliofotou, Ram Keralapura, Michalis Faloutsos, and Antonio Nucci. Subflow: Towards practical flow-level traffic classification. In *2012 Proceedings IEEE INFOCOM*, pages 2541–2545. IEEE, 2012.

[38] Behnaz Arzani, Selim Ciraci, Stefan Saroiu, Alec Wolman, Jack Stokes, Geoff Outhred, and Lechao Diwu. Privateeye: Scalable and privacy-preserving compromise detection in the cloud. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 797–815, 2020.

[39] Tohid Shekari, Christian Bayens, Morris Cohen, Lukas Graber, and Raheem Beyah. Rfdids: Radio frequency-based distributed intrusion detection system for the power grid. In *NDSS*, 2019.

[40] Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *2010 IEEE symposium on security and privacy*, pages 305–316. IEEE, 2010.

[41] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. Quantitative network monitoring with netqre. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 99–112, 2017.

[42] Mina Lee, Sunbeom So, and Hakjoo Oh. Synthesizing regular expressions from examples for introductory automata assignments. In *ACM SIGPLAN Notices*, volume 52, pages 70–80. ACM, 2016.

[43] Sunbeom So and Hakjoo Oh. Synthesizing pattern programs from examples. In *IJCAI*, pages 1618–1624, 2018.

[44] Rajeev Alur, Konstantinos Mamouras, and Caleb Stanford. Modular quantitative monitoring. *Proceedings of the ACM on Programming Languages*, 3(POPL):50, 2019.

[45] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *ICISSP*, pages 108–116, 2018.

[46] Canadian Institute for Cybersecurity. Ids 2017 | datasets | research | canadian institute for cybersecurity | unb, 2020. [Online; accessed 15-October-2019].

[47] Gerard Draper-Gil, Arash Habibi Lashkari, Mohammad Saiful Islam Mamun, and Ali A Ghorbani. Characterization of encrypted and vpn traffic using time-related. In *Proceedings of the 2nd international conference on information systems security and privacy (ICISSP)*, pages 407–414, 2016.

[48] Arash Habibi Lashkari, Gerard Draper-Gil, Mohammad Saiful Islam Mamun, and Ali A Ghorbani. Characterization of tor traffic using time based features. In *ICISSP*, pages 253–262, 2017.

[49] Riccardo Bortolameotti, Thijs van Ede, Marco Caselli, Maarten H Everts, Pieter Hartel, Rick Hofstede, Willem Jonker, and Andreas Peter. Decanter: Detection of anomalous outbound http traffic by passive application fingerprinting. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 373–386, 2017.

[50] Floodlight. `https://github.com/floodlight/floodlight`, 2022. [Online; accessed 20-April-2022].

[51] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, volume 15, 2011.

[52] Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 530–541. ACM, 2014.

[53] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark W. Barrett, and Morgan Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, volume 8559 of *LNCS*, pages 646–662. Springer, 2014.

[54] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Murphy Berzish, Julian Dolby, and Xiangyu Zhang. Z3str2: an efficient solver for strings, regular expressions, and length constraints. *Formal Methods Syst. Des.*, 50(2-3):249–288, 2017.

[55] Liushan Chen, Yu Pei, and Carlo A. Furia. Contract-based program repair without the contracts. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 637–647. IEEE Computer Society, 2017.