# Synthesis of Stateful Systems from Execution Traces

Margarida Ferreira

Thesis Proposal

May 2025

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

**Thesis Committee**

Ruben Martins, Co-chair
Inês Lynce, Co-chair
João F. Ferreira
Justine Sherry
Fraser Brown
Nate Foster

*Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy*

# Abstract

Execution traces are a valuable source of information in modern computing systems. They are continuously collected and used for system debugging, monitoring, and optimization. They capture behavior across diverse scenarios, from routine operations to edge cases. This thesis investigates how execution traces can serve as specifications for program synthesis, enabling reverse engineering and analysis of complex systems and automation of traditionally manual tasks without explicit user input.

Program synthesis usually starts from high-level specifications expressed in input-output examples or natural language descriptions of desired behavior. Though more accessible than writing computer code directly, these specifications, in practice, require expertise and multiple iterations to articulate the desired behavior in a way that allows the synthesizer to produce the correct program. For many applications, execution traces provide a passive and accessible alternative: they are continuously collected, require no additional effort from users, and inherently capture real-world system behavior.

To keep the synthesis process passive and not reliant on user input, we must guarantee that the synthesized programs exhibit the desired behavior. We conduct a controlled search over the space of feasible programs, generating explanations for why one candidate program is selected over another. However, traces may contain errors or inaccuracies. When recording activity executed by networked components, data may get reordered or lost, introducing errors into collected traces. If traces are recorded from human agents executing tasks, they may include specificities or variations of the task that we do not wish to capture in synthesis. We aim to extract and generalize behavior exhibited across multiple traces while filtering out outliers. Another challenge comes from the volume of data. It is essential to ensure scalability as well as the correctness of the output program since traces can be too long or too many. A crucial part of working with traces is filtering and, potentially, sampling them to keep the problem tractable.

This proposal presents three synthesis frameworks, Abagnale, Syren, and Sluice, that illustrate the challenges of this problem on multiple applications and how we overcome them.

- Abagnale reverse-engineers the behavior of Congestion Control Algorithms (CCAs) from network traces. Network traces contain no information about the implementation of the CCA, displaying only the effects of their executions in the network. Thus, Abagnale must simulate each candidate solution in the same network conditions to assess if they exhibit the same behavior. To capture all different behaviors, we work with traces showing hundreds of executions, making trace filtering and parallelization paramount to Abagnale's viability.
- Syren allows users to generate arbitrary programs from partial traces that contain some of the function calls made by the program. Syren uses optimizing rewrites to introduce control flow in the program. These optimizing rewrites track the data used in the functions visible in the trace, which is then used to generate function calls *not* visible in the trace

using an example-based syntax-guided synthesizer.

- HyGLAD synthesizes regex-based anomaly filters that flag deviations from a system's expected behavior from execution logs. In this case, our goal is not to reverse-engineer the system itself but to synthesize a model of its execution.

All three synthesis frameworks were evaluated using real-world traces from their respective domains. Finally, as future work, we propose to develop a fourth synthesis approach to automate data-aware business processes modeled as data Petri nets. We will use logs collected from human-executed processes as traces and synthesize implementations that model the task logic, filtering out inconsistencies and errors unavoidable in human-generated logs.

# Contents

# 1

# Introduction

Execution traces are ubiquitous in computer science. Modern systems routinely log their activity for debugging, monitoring, billing, and optimization purposes, generating massive amounts of trace data. Examples of such traces include sequences of messages exchanged between networked servers, logs of calls made to an Application Programming Interface (API), or traces of system calls made by a workload. These logs reflect how systems behave under diverse scenarios, capturing not only the normal everyday behaviors of the systems but also the corner cases and subtleties of their implementations. In this thesis, we propose to explore how we can use trace data to synthesize implementations of the underlying systems with the ultimate goal of reverse engineering systems in the absence of source code, facilitating the optimization and analysis of complex systems by synthesizing simpler implementations, and automating processes currently reliant on manual effort.

The synthesis of programs from high-level specifications has been a focus of research for almost as long as there have been programming languages [92, 129]. First, researchers focused on synthesizing programs from logical formulas [57], which require significant expertise to write. With the goal of reaching a broader audience, another more accessible kind of specification arose: input-output examples [60]. Although easily readable to non-programmers, providing examples to a synthesizer requires specific knowledge and often multiple attempts or interactions with the user [46, 114] to ensure the synthesizer produces a program with the desired behavior. In the last few years, spurred by advancements in large language models, natural language has emerged as the main type of specification for generating programs automatically [70]. These systems make synthesis and programming more accessible to non-experts than ever by enabling users to describe desired program behavior using plain language. Nonetheless, these systems still require users to articulate their intent explicitly, often requiring more than one round of interaction [80] to clarify the desired behavior.

Logical formulas, input-output examples, and natural language specifications are all *active* synthesis specifications: they require users to articulate their intent explicitly. In this thesis, we instead tackle the challenge of synthesizing from a *passive* specification. We explore ways to synthesize using trace data collected during normal system operation without requiring additional input or effort from the user. From the user's standpoint, sets of traces are an ideal specification for program behavior. Non-programmers may access them from logs of tasks performed through a graphical interface since systems record underlying operations performed and use them to

automate their tasks. Traces are often collected continuously and automatically, making them readily available. They describe program behavior (albeit partially) so systems managers can also use them to generate models or simplifications of their systems' implementations, making their analysis and optimization easier. In a different scenario, a user may use the traces of the execution of a system whose implementation they cannot access to reverse-engineer and analyze it.

Despite their appeal, traces as specifications present significant challenges for program synthesis. First, traces are often *incomplete*. They represent only a subset of the system's behavior. Synthesizing a program that generalizes beyond the provided traces without introducing unintended behavior requires careful reasoning. Second, traces can be *noisy*. They may reflect implementation artifacts irrelevant to the intended functionality or errors, unpreventable from in the collection procedure. Third, traces do not show information about each execution of the desired system; instead, they cover *multiple successive executions* of the underlying program. This makes it hard to isolate the behavior of a single execution and to uncover the state maintained across multiple executions. Finally, execution traces can be *arbitrarily large and numerous*, which can easily render synthesis intractable, since we must reason about all them.

> **Thesis statement**
> We synthesize provably correct stateful programs from execution traces, even when the traces are incomplete, noisy, and lack information about the program's internal state. The synthesized programs can help users analyze their systems, automate manual tasks, or reverse-engineer systems whose source code is not available.

There is a line of work akin to synthesis from traces, Programming-By-Demonstration (PBD) [34, 78, 122]. PBD systems allow users to record their actions and generate programs that automate these actions. However, these systems aim only to uncover how to implement *each recorded action*, i.e., find the underlying functions to call and not to generalize beyond them. In contrast, we aim to generate programs that generalize beyond the recorded actions and include parts of the task that cannot be observed in the traces.

In this proposal, we present two past and one ongoing synthesis projects, each tackling different challenges of synthesis from traces. The first, Abagnale [44, 49], uses program synthesis to reverse-engineer the behavior of deployed CCAs from collected packet traces to facilitate their analysis. Second, Syren [47, 48] synthesizes arbitrary programs from logs containing partial information about their execution. The third project presented, HyGLAD [5], synthesizes a regex-based anomaly filter from structured logs showing a system's desired behavior. Finally, we propose to develop a fourth and final synthesis system that combines the strengths of the past work.

**Abagnale** Abagnale is a program synthesis pipeline that aims to reverse-engineer deployed CCAs' behavior from collected packet traces. It helps users automate the reverse-engineering task by discovering simpler implementations of CCAs that behave in the same way as the collected set of traces in the same conditions. The network traces used as a specification contain no information about the CCA or what signals/operations it uses. They show only the effects that the CCA has on the network. Furthermore, due to the nature of the collection of these traces, they may contain errors or noise, adding complexity to the synthesis process. All these challenges force Abagnale to rely on domain-specific knowledge to guide the synthesis process, so although the techniques used in Abagnale are general, in practice they are tailored to the problem of synthesizing CCAs,

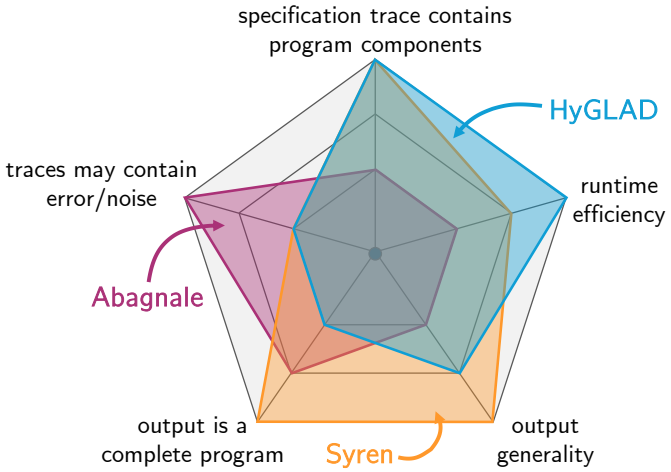



Figure 1.1: Visual comparison of the different properties in each synthesis system presented in this thesis.

and Abagnale cannot be easily applied to different domains. Abagnale is described in depth in Chapter 3 → p.11.

**Syren** Syren is a general synthesis approach that generates arbitrary programs from logs containing some of the functions called in their execution. It aims to generalize observed behavior to create a complete program that matches the partial traces and generalizes beyond them. Unlike Abagnale, the specification traces of Syren contain parts of the desired program. The calls to some functions, called *visible functions*, are visible in the traces, along with their input and output. Between the visible function calls, the program must maintain some hidden state and execute additional *hidden functions*. Syren's specification also differs from Abagnale's in that its program logs are collected at the source, and they do not contain errors or noise. We present Syren in detail in Chapter 4 → p.34.

**HyGLAD (ongoing work)** HyGLAD synthesizes a graph- and regex-based anomaly filter from structured logs that show the desired behavior of the system. It aims to identify and filter out anomalies in new incoming logs, helping users understand and analyze the normal behavior of the system, as well as identify and react to anomalous behavior. The specification is similar to Syren's, a set of traces consisting of logs from a system's executions, but the output differs significantly. HyGLAD's ultimate goal is to know if a new event was generated by normal execution of the same system as the specification, not to synthesize or reverse-engineer the *program* that generated the traces. We present HyGLAD in detail in Chapter 5 → p.59.

**DaPDiS (proposed work)** In the future, we want to build a new synthesis system that combines the generality of Syren with the robustness to errors and noise of Abagnale, leveraging the noise-identifying abilities of HyGLAD. Syren works with a set of logs that all perform the same task, but it is assumed that these logs have no errors. We want to extend this functionality to sets of logs that may contain errors or anomalies in the execution of the task. To test the new system, we will use a benchmark of logs of the manual execution of business processes. In business process logs, there are errors, noise, and variations in the way the task is performed. Humans

execute the tasks, and the logs are collected by monitoring the user's actions. So, to work with these logs, the proposed synthesizer will need to be able to generate a program that performs the same task as the logs, even if they contain errors. We delve into the proposed approach in Chapter 6 → p.72.

### Thesis structure

The rest of this thesis is organized as follows. In Chapter 2 → p.5 we present general background in program synthesis, necessary to understand the synthesis systems presented in this document. The following chapters introduce further background specific to the respective system. In Chapter 3 → p.11, we present Abagnale, a program synthesis pipeline that reverse-engineers the behavior of deployed CCAs from collected packet traces. In Chapter 4 → p.34, we present Syren, a general program synthesis approach that generates arbitrary programs from logs containing partial information about their execution. In Chapter 5 → p.59, we present HyGLAD, a system still under development, that synthesizes a hypergraph-guided regex-based anomaly filter from structured logs showing a system's desired behavior. In Chapter 6 → p.72, we propose to develop DaPDiS, a program synthesis system that extends Syren's functionality to work with noisy traces, such as the logs of business processes. Finally, in Chapter 7 → p.75 we propose a timeline for the development of the remaining PhD work, and in Chapter 8 → p.77 we conclude the thesis proposal.

# 2

# Background

## *An Overview of Program Synthesis*

For as long as there have been programming languages, computer scientists and engineers have been interested in the problem of automatically generating programs. It makes sense: one of the main advantages of computing is that it can automate repetitive tasks. So, once writing programs became such a commonplace activity, it only made sense to want to automate it as well. Thus was born program synthesis, the subfield of computer science research that studies how to automatically generate programs from high-level specifications that establish semantic and syntactic requirements for the generated code. Figure 2.1 shows the typical interaction with a program synthesizer.

The first program synthesis works, published in the late 1960s and early 1970s, proposed generating programs from formal specifications, such as logical formulas or mathematical equations [92, 129]. Today, that might seem nonsensical: why would anyone prefer to write a logical specification instead of a program? But we are picturing the programming languages we have nowadays, which have been through decades of development to make them human-friendly and as robust to errors as possible. Generating a program from a logical formula was a popular idea fifty years ago when computer programs were harder to write, read, and debug. Furthermore, computers were not a widespread commodity—they were used mainly by scientists or mathematicians, who *already* had the expertise required to write a logical specification.

As programming languages and paradigms evolved, so did program synthesizers. Once computers became a common appliance in people's homes, the focus shifted to generating programs from high-level specifications that are closer to what non-programmers and non-mathematicians know: input-output examples [58, 79, 111], or demonstrations [123, 126]. This shift was driven by the desire to make programming more accessible to a wider audience, including non-experts and domain specialists who may not have formal programming training.

The advent of deep learning and, more recently, Large Language Models (LLMs), has opened new doors in the field of program synthesis. LLMs can generate programs from natural language descriptions of what the program should do. Some tools, like Copilot [29], can even infer intention from previous lines of code, offering a very advanced auto-complete system that is very helpful for programmers. These new approaches can handle more complex tasks and generate more sophisticated programs faster than ever before. However, there are other challenges to overcome.

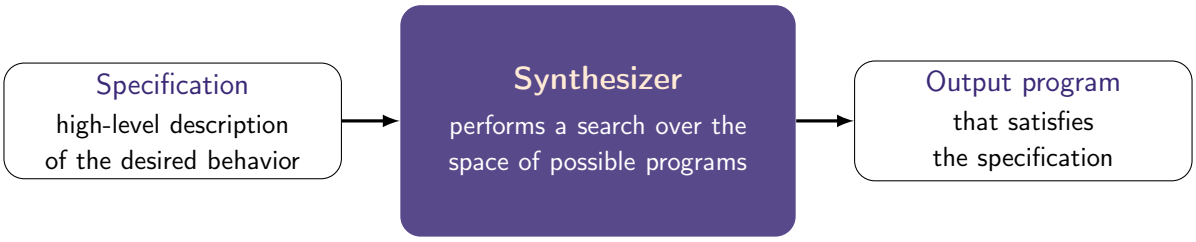



Figure 2.1: Program synthesis: the synthesizer takes a high-level specification and generates a program that satisfies it.

Deep learning synthesizers require large amounts of training data to produce correct results, whereas formal methods produce programs that are correct by construction based on just a few examples. Without a guarantee that the synthesized programs are correct, users must have some expertise in the programming language in question to be able to verify the correctness of the synthesized output. Finally, while their effectiveness has been demonstrated for extensive benchmarks of common programming tasks, it is hard to verify whether they can generalize to more niche programming tasks [113].

In this thesis, our goal is to synthesize correct programs from execution traces, without any additional natural language input, so we focus on example-based synthesis techniques.

## 2.1 Synthesis from Examples: PBE *vs.* PBD

In this section, we will take a deeper look at synthesis from input-output examples. The literature makes a distinction between Programming-By-Example (PBE), and Programming-By-Demonstration (PBD). In PBE, the user describes the program they want to synthesize by providing examples of desired input-output pairs, whereas in PBD, they also provide a trace of how the output was computed.

**Example 2.1: PBE *vs.* PBD**[a]

[a] This example is adapted from [124]

For example, if a user wants to synthesize the factorial function, they may provide the following set of examples to a PBE synthesizer:

$$\begin{aligned}\text{factorial}(3) &= 6\\ \text{factorial}(4) &= 24\\ \text{factorial}(5) &= 120\\ \text{factorial}(6) &= 720\end{aligned}$$

By contrast, with PBD, one may provide a more detailed trace of the computation:

$$\text{factorial}(6) = 6 \times (5 \times (4 \times (3 \times (2 \times 1)))) = 720$$

At first glance, it may seem that the main challenge that PBE and PBD synthesizers need to overcome is to *find a program that produces the same output for the given inputs*. But, as may

be clear in Example 2.1, PBE is a highly underspecified problem, since there are many possible functions that can return the provided output for each of the inputs, or produce a particular trace. So at the end of the day, PBE synthesizers are not very useful unless they can also answer the question *How do we know if the program we find is the one the user was actually looking for?*

For this reason, PBE synthesizers will generally need a large number of examples to constrain the search space as much as possible. Besides, PBE synthesizers rank the programs in the space, so that the first program they find is also likely to be what the user is looking for. This is not guaranteed to yield exactly the right program in every situation, but the more efficiently we can search the large but highly constrained search space, the more likely we are to find a program that does what the user was looking for. To be able to deal with fewer examples and still ensure the output program is as close to the user's need as possible, synthesizers often also rely on a disambiguation loop [46, 114], where the synthesizer collects more information from the user.

In PBD, the user demonstrates the desired behavior of a program through interactions with a system. It is often used in applications such as robotic programming, where users can demonstrate tasks through physical interactions. These interactions are recorded and the systems produce a trace with information on the specific steps that were executed. This makes it easier to infer the intended computation, so PBE systems need fewer examples. Nevertheless, the problem still remains with too few examples, PBD synthesizers struggle to know which parts of the program they should generalize in order to generate a program that not only replicates the input traces, but also exhibits the desired behavior in other contexts.

## 2.2 Domain Specific Languages

Program synthesis, in general, has the goal of synthesizing a program in some language that can be executed (Python, Rust, Java). But, in practice, to reason about the search space as efficiently as possible, example-based synthesizers target a more narrow, formally-defined language, the Domain-Specific Language (DSL). The choice of a DSL affects the synthesizer's efficiency (smaller DSLs are quicker to traverse) but also its expressiveness (larger DSLs allow the synthesizer to be used for more domains).

In most synthesis works, as well as in this thesis, we describe synthesis DSLs as grammars defined in the Backus-Naur Form (BNF), a notation system to define the syntax of programming languages and other formal languages. BNF specifications outline how the symbols in the language can be combined to form syntactically valid sequences. Each BNF grammar consists of three core components: a set of *non-terminal symbols*, a set of *terminal symbols*, and a series of *derivation rules*. Non-terminal symbols represent categories that can be replaced with other symbols, while terminal symbols are the fixed, literal elements (such as keywords or constants in the language) that appear in the final sequence. Derivation rules provide the instructions for replacing non-terminal symbols with specific combinations of symbols. A valid production of the grammar—in other words, a valid program in the DSL—is a sequence of terminal symbols that follow the derivation rules. The following is an example of a derivation rule in BNF notation:

$$\mathcal{S} \quad ::= \quad \textit{expression}$$

$\mathcal{S}$ is a non-terminal symbol, ::= means "is replaced by" and *expression* is the replacement, consisting of one or more sequences of symbols—either other non-terminal symbols, or terminal symbols, such as literal or constant values of the language—with options separated by a vertical bar (|) to indicate alternatives. Example 2.2 shows an example of a Context-free grammar of

sums of integer numbers in the BNF format.

**Example 2.2: Context-free grammar in BNF**

$$
\begin{array}{lcll}
\mathcal{F} & ::= & \mathcal{F} + \mathcal{N} & \textit{Factor} \\
\mathcal{N} & ::= & \mathcal{D} & \textit{Number} \\
 & | & \mathcal{N}\mathcal{D} & \\
\mathcal{D} & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 & \textit{Digit} \\
 & | & 5 \mid 6 \mid 7 \mid 8 \mid 9 &
\end{array}
$$

Above is an example context-free grammar in BNF notation. The productions of this grammar are additions of non-negative integers, such as 10+5, 42, and 1+2+3.

**Syntax-Guided Synthesis (SyGuS)** Syntax-Guided Synthesis (SyGuS) synthesis [4] is a branch of program synthesis that targets the synthesis of programs in arbitrary DSLs. SyGuS synthesizers accept from the user not only the specification of the program's desired behavior, but also the target DSL's syntax (defined as a context-free grammar) [108]. SyGuS synthesizers are developed with the goal of being used in any domain. On the other hand, syntax-guided synthesis requires the user to have a deeper technical knowledge not only on the specific domain on which he or she is working but also on formal languages and how to define them.

## 2.3 Constraint Solving

Constraint-solving techniques are used to solve problems where logical constraints constrain the space of solutions. In this proposal, we often rely on synthesis techniques that are implemented using Satisfiability Modulo Theories (SMT). SMT can be seen as a generalization of the Propositional Satisfiability (SAT) problem. The next sections provide a brief introduction to each of these formalisms.

### 2.3.1 Propositional Satisfiability (SAT)

In logic and computer science, SAT is the problem of, given a Boolean formula, determining if there exists an assignment of its variables that satisfies it. In addition to its theoretical importance, SAT has many practical applications in the field of Computer Science. In 1971, SAT was the first problem proven to be NP-Complete [33]. This means that numerous decision problems can be reduced to the SAT problem, and subsequently solved using an off-the-shelf SAT solver.

**Definition 2.1: Literal**
A literal $l$ is a Boolean variable ($l = x$) or its complement ($l = \neg x$).

**Definition 2.2: Clause**
A clause $c$ is a disjunction of literals: $c = l_1 \vee l_2 \vee ... \vee l_k$.

SAT formulas are usually represented in the Conjunctive Normal Form (CNF).

**Definition 2.3: CNF Formula**
A formula $\phi$ in CNF is a conjunction of clauses: $\phi = c_1 \wedge c_2 \wedge ... \wedge c_n$.

**Example 2.3: SAT Formula in CNF**

Consider the variables $X = \{x_1, x_2, x_3\}$. Then, $\phi_1 = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3)$ is a CNF formula over $X$.

**Definition 2.4: Assignment**

Given a formula $\phi$, an assignment is a mapping $\nu : X \to \{\text{true}, \text{false}\}$, where $X$ is the set of variables in $\phi$.

Given an assignment $\nu : X \to \{\text{true}, \text{false}\}$ and a variable $x \in X$, the positive literal $x$ is satisfied by $\nu$ if and only if $\nu(x) = \text{true}$ and the negative literal $\neg x$ is satisfied by $\nu$ if and only if $\nu(x) = \text{false}$; a clause is satisfied by $\nu$ if and only if at least one of its literals is satisfied by $\nu$; a formula in CNF is satisfied by $\nu$ if and only if all of its clauses are satisfied by $\nu$.

**Definition 2.5: Model**

Given a propositional formula $\phi$ and an assignment $\nu$, $\nu$ is a model of $\phi$ if and only if $\nu$ satisfies $\phi$.

**Example 2.4: Model of a Formula**

Recall the CNF formula from Section 2.3.1 → p.9: $\phi_1 = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3)$. The assignment $\nu_1 = \{x_1 \mapsto \text{false}, x_2 \mapsto \text{true}, x_3 \mapsto \text{true}\}$ is a possible model of $\phi_1$.

### 2.3.2 Satisfiability Modulo Theories (SMT)

Oftentimes, it is advantageous to express the problem at hand using more expressive logics. To encode the constraint problems in synthesis, we often use an extension to SAT: Satisfiability Modulo Theories (SMT). SMT is the problem of satisfiability of formulas with respect to some background theory $\mathcal{T}$, which defines the interpretations of certain function symbols. This allows us to incorporate fragments of first-order logic in CNF formulas.

**Definition 2.6: $\mathcal{T}$-atom**

A $\mathcal{T}$-atom $t$ is a ground atomic formula in theory $\mathcal{T}$.

**Definition 2.7: $\mathcal{T}$-literal**

A $\mathcal{T}$-literal is a $\mathcal{T}$-atom ($t$) or its negation ($\neg t$).

**Definition 2.8: $\mathcal{T}$-formula**

A $\mathcal{T}$-formula is then analogous to a propositional formula, but it is composed of $\mathcal{T}$-literals instead of propositional literals.

The values in an SMT assignment depend on the background theory $\mathcal{T}$. Commonly used theories include the theory of Linear Integer Arithmetic (LIA), whose values are integers, the theory of *strings*, whose values are strings, and the theory of Equality with Uninterpreted Functions (EUF), which assigns functions with a valid interpretation. Given an SMT assignment, $\nu$, and a $\mathcal{T}$-atom $t$, the positive $\mathcal{T}$-literal $t$ is satisfied by $\nu$ if and only if $\nu(t)$ is assigned trueaccording to theory $\mathcal{T}$ and the negative $\mathcal{T}$-literal $\neg t$ is satisfied by $\nu$ if and only if $\nu(t)$ is assigned falseaccording to $\mathcal{T}$; a $\mathcal{T}$-formula is satisfied by $\nu$ if and only if all of its clauses are satisfied by $\nu$.

SMT is then the problem of, given an SMT formula $\phi$, deciding if there exists an assignment of the variables and functions of $\phi$ that satisfies it.

**Example 2.5: SMT formula**

Let $\phi_3 = (b - a = 1) \wedge ((b < 5) \vee (a > 10))$ be a SMT formula in the theory of LIA. A possible model for $\phi_3$ is $\nu_3 = \{a \mapsto 1, b \mapsto 2\}$.

# 3

# Abagnale

## *Synthesis of Congestion Control Algorithms from Network Traces*

**Contents**



Analyzing CCAs is vital to our understanding of Internet traffic stability, fairness, and performance. For example, past analyses have shown that Additive Increase Multiplicative Decrease (AIMD) approaches such as Reno will converge to fair bandwidth shares [31] and that Google's BBRv1 will converge to unfair bandwidth shares in many scenarios [131]. Recent work has further introduced model checking to CCAs [7], providing the ability to prove performance guarantees for input CCAs given complex edge case scenarios. Of course, these analysis techniques rely on access to a CCA's implementation.

Unfortunately, implementations of CCAs are not universally available. Especially given the rise of user-space CCA implementations [86, 105], it is easier than ever to develop and deploy a proprietary CCA without revealing its implementation details. Indeed, researchers have both observed [24, 100] and publicly claimed [132] proprietary CCA implementations. Further, even with access to a CCA's implementation, many such implementations contain datapath-specific implementation details that obscure their behavior [105], making it difficult for analysts to determine their nature.

Thus, we seek to facilitate CCA analyses by *synthesizing* simple implementations directly from packet traces. These simple implementations make the analysis of CCAs with known (but complex) implementations *easier*, while making analysis of unknown CCAs *possible*.

We scope our design to synthesize *classically-designed congestion control algorithms*. We formalize what it means for a CCA to be 'classically designed' in Section 3.2.3 → p.17, but at a high level, this means that it can be constructed using a DSL derived from the set of existing

Linux Kernel supported CCAs. For example, we aim to design a tool that could reverse engineer TCP Westwood if it were brand new to the CCA landscape, since Westwood can be constructed using the same language as TCP Reno. Since most novel CCAs on the Internet today are variants or extensions of classical algorithms (Section 3.4 → p.23), 'classically designed' algorithms are a useful category of algorithms to specialize in. Importantly, however, this design excludes from our scope CCAs with non-deterministic behavior (including those using machine learning techniques).

The reason that program synthesizers target domain-specific regimes is tractability. At their core, all synthesizers frame a search space using a DSL defining the 'set of all possible programs' and aim to find a needle in this haystack: a program that, given the pre-specified inputs, produces the pre-specified outputs. Constraining the size of the DSL makes the haystack smaller. Nonetheless, even a constrained haystack is still large: even when constrained to CCAs following a 'classical' DSL, there are $10^{150}$ possible programs to explore.

To meet this challenge, we design Abagnale , a program synthesis pipeline that utilizes domain knowledge to produce approximate expressions representing CCAs in our scope. Abagnale cuts down on the intractability of synthesizing CCAs by taking an uncommon approach to program synthesis: formulating the problem as an *optimization* problem (in which a numerical objective is maximized or minimized) rather than a *decision* problem (in which a logical formula is 'satisfied' or 'unsatisfied'). *Our insight is that we can evaluate candidate programs in the search space based on a measurable distance between the visible CWND of the candidate CCA in simulation and the observed CWND of the true CCA in the wild. We then formulate our procedure to select a program that minimizes this distance.*

Our observation that CCAs *are* amenable to an optimization formulation provides cross-cutting gains across four dimensions of the synthesis process:

**Evaluation** When we have a candidate CCA proposed by the synthesizer, how can we evaluate whether it matches the expected behavior of the ground truth CCA? A key challenge with measurements of ground truth CCAs 'in the wild' is the presence of noise in our measurements of the true CWND: our observation of the CWND may be incomplete due to our measurement vantage point, packets may be dropped or delayed, *etc.* An optimization formulation allows us to accept candidate CCAs that are close, but imperfect matches to what is observed, compensating for this noise.

**DSL Formulation** We aim to identify any CCA that fits the DSL defined by 'classical' CCAs. Nonetheless, this search space still remains intractably large. Here, we leverage the fact that CCAs typically fall into 'families' (*e.g.*, Westwood is in the Reno family; Veno is in the Vegas family.) Hence, we break the DSL into sub-DSLs per CCA family. When given a new ground-truth CCA trace, we use existing classification techniques to constrain the search to a sub-DSL containing only operators and values for that CCA family.

**Program Search** Finally, the process of searching for candidate programs is greatly aided by the presence of a measurable function to declare which programs are 'closer' to the correct solution and which are 'further'. We perform several key optimizations here, including breaking down the search further across sub-DSL 'buckets' and parallelizing search across these buckets [22]. To prioritize which buckets to search first, we sample a few candidate CCAs from the bucket and

evaluate their distance to the ground-truth measurements; we then prioritize search in the buckets with closer measurements over CCAs with further measurements.

**Goal** We emphasize that, with Abagnale , we do not seek to produce the *precise* CCA implementation that produced the behavior observed in a given set of packet traces. Rather, our goal is to produce a succinct representative expression that captures a CCA's core behavior. Even this limited goal is an ambitious step relative to both modern CCAs as well as state-of-the-art program synthesis techniques. Thus, Abagnale is an initial step towards understanding unknown CCAs rather than the final word. For example, many of the most advanced congestion controllers today incorporate machine learning or statistical techniques, and reverse engineering these remains out of reach for Abagnale or any other existing technique. Similarly, as we discuss in Section 3.2 → p.15, Abagnale cannot discover hidden state variables in CCAs that affect their externally visible trace behavior.

**Key Results** Despite these challenges, as we show in Section 3.4 → p.23, Abagnale produces a closed-form expression that approximates BBR without using state variables to maintain the pulse state, as most existing implementations do. Further, Abagnale produces expressions matching those fine-tuned by a domain expert with knowledge of the CCA in question for 9 out of 16 CCAs distributed with the Linux kernel. Of the remaining 7 CCAs, 2 (LP and HTCP) miss conditional modes of operation, 2 (HighSpeed and CDG) are out of our scope due to their use of non-determinism and out-of-DSL operators, 1 (Cubic) exposes a limitation in our SMT-encoded search space constraints, and the last (BIC) has an expression depth too deep to find within Abagnale 's time-bound. Additionally, we find (Section 3.5 → p.29) that Abagnale is able to efficiently discard large and irrelevant portions of the search space from contention in all evaluated cases. In contrast, prior work on Mister880 [45] cannot synthesize any algorithm other than NewReno (measured without noise) and cannot handle noisy traces at all.

## 3.1 Motivation and Background

In this section, we first discuss why reverse engineering congestion control algorithms (CCAs) from packet traces is useful (Section 3.1.1 → p.13) and then discuss why existing state-of-the-art approaches to synthesizing programs from examples are insufficient to synthesize CCAs (Section 3.1.2 → p.14).

### 3.1.1 Why reverse-engineer CCAs?

Today's Internet presents an unprecedented explosion in congestion control diversity. Although NewReno and Cubic were often assumed to be the only players in the past, recent studies show tremendous diversity in CCA deployments, with one 2019 study reporting 6 algorithms with deployments across 2% or more of servers [100]. There is also an incredible amount of experimentation: in 2017, Google silently rolled out BBR'1.1' without fanfare; in 2019, Netflix deployed a custom variant of NewReno using its RACK stack in FreeBSD; cloud gaming providers today continue to develop bespoke proprietary CCAs [93].

Many researchers predict CCA diversity will increase in future years. First, with the rise of user-space networking stacks such as HTTP/3 (QUIC), modifying CCA code will become

easier for developers who no longer have to delve into kernel space to make modifications. Second, application developers are beginning to see gains from "bespoke" CCAs designed in a way that is specifically tailored to their application. Hence, we have seen proposals for CCAs tailored to video streaming [53, 106] or cloud gaming [115]. In addition, novel algorithms may be proprietary, with companies unlikely to share the 'secret behind their applications' competitive network performance.

The explosion in CCA diversity has important implications for many of the fundamental design goals of the Internet. For example, new CCAs may improve or harm any of capacity utilization, the Internet's fairness landscape, average latency, or burstiness. *Hence, it is no surprise that in recent years researchers have invested significant attention towards characterizing the explosion in CCA diversity.*

**CCA Classifiers** Most attempts to characterize the Internet CCA landscape focus on *classifying* CCAs: identifying whether a given Internet service is using a particular published CCA. State-of-the-art examples from this literature include Gordon [100], Inspector Gadget [56], and others [109, 118, 130].

Most CCA classifiers connect to a server under investigation and attempt to model or measure the CCAs visible congestion window (visible CWND): the number of outstanding bytes in flight, over time. They then use some classification algorithm (e.g., decision tree, neural network) to match this time series of CWNDs to known, ground-truth observations of existing CCAs (typically some subset of the 16 default CCAs available in the Linux kernel). Classifiers can neither provide insight into these unknown CCAs (other than that they are unknown), nor provide any insight into known CCAs (and indeed can also mis-classify known CCAs).

**The Impact of Unknown CCAs** As discussed above, CCAs determine crucial properties of the Internet's performance, such as link utilization, fairness, burstiness/variability, and latency. One useful way to understand unknown CCAs is performing measurement either "in the wild" [36] or in testbed environments [8, 100, 110, 117]. While these experiments *can* illuminate useful, empirical properties of the observed CCAs under study, they remain limited. Without knowledge of the underlying algorithm, it is not possible to *prove* bounds or guarantees about the algorithm's behavior [7]. It is also not possible to diagnose *why* a particular pessimal behavior is happening or make recommendations how to fix it: empirical tests simply discover *that* something is wrong. *Hence, we argue that to truly understand the impact of novel CCAs on the Internet performance landscape, it is crucial to understand the algorithm behind each and every CCA.*

### 3.1.2 Program Synthesis for Reverse Engineering

The process of generating a program based on its observed inputs and outputs is, by definition, a form of *program synthesis*. A large literature of PBE tools follows the blueprint of taking observed inputs and outputs and generating a program that maps from input to output. However, existing PBE tools cannot reverse engineer congestion control algorithms for two key reasons: **statefulness** and **noise**.

**Statefulness** Prior work has proposed many specialized PBE synthesizers to solve practical problems such as data structure transformations [51], spreadsheet data manipulation [60], data

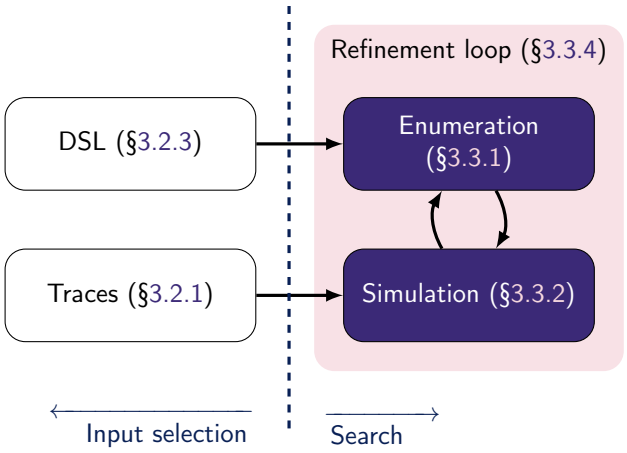



Figure 3.1: Abagnale system overview

preparation tasks [42], and applications to computer networks [28, 121, 138]. There are also general PBE synthesizers [95] that take any DSL and a set of examples as input and produce a program that satisfies the examples. To produce this program, synthesizers use machine learning [12, 29, 82, 87], constraint solvers [79, 125], or some combination thereof [30, 43].

Unfortunately, not all program outputs are merely the result of a stateless operation over visible input variables, and CCAs are one such case. For example, the output of TCP NewReno after a loss is not simply $\frac{1}{2}$, it is $\frac{1}{2} \times$CWND, the previously held congestion window. Each timestep of the CCA's progress depends on *both* the inputs observed (losses, packets ACKed, measured RTTs) *as well as the existing state of the system.* Because most PBE synthesizers cannot model this state, they cannot synthesize CCAs.

**Noise** Mister880 is a prototype CCA synthesizer [45]. Like us, the authors aim to use Mister880 to uncover the underlying algorithms of novel CCAs over the Internet. However, Mister880 formulates the synthesis as a *decision problem* (as do the vast majority of program synthesizers): it is only capable of modeling candidate synthesized programs as 'correct' or 'incorrect' and has no flexibility for programs that are 'close' to the correct solution but which do not perfectly replicate the observed behavior of the ground truth implementation. This matters because in the Internet, packet traces from a given CCA are *noisy.* Measurements of the ground-truth CWND may reflect the vantage point of the trace measurement; there may be packet delays or jitter that are unobserved; there may be unexpected timeouts or losses that are observed at the sender but not to our measurements, *etc.*. Hence, even if we had an exact copy of the ground truth system, it is not possible to guarantee that our measurements of the system 'in the wild' and our measurements of the system in our testbed will be truly identical. In this setting, Mister880 would discard even the correct algorithm as incorrect.

## 3.2 Model and Inputs

In the following sections, we describe Abagnale , the first program synthesizer to take on (a) stateful programs and (b) noisy input data. Abagnale 's stateful model of CCA behavior is similar

to the model used by Mister880, but its formulation of program synthesis as an *optimization* problem rather than a *decision problem* is an entirely different formulation.

Abagnale uses the measured outputs of candidate synthesized CCAs to compute a 'distance' between the proposed CCA and the ground-truth measurement; candidate CCAs with a lower distance are considered better than those with a higher distance. Because our goal is to minimize the distance, but not necessarily bring it to 0, Abagnale can produce algorithms which almost match the observed behavior of the ground truth CCA, thus accounting for noise.

We find that formulating the synthesis problem as an optimization procedure allows us to make further improvements to our synthesis procedure, including making the search for the best candidate CCA more efficient and making the search space of candidate CCAs smaller (Section 3.3.4 → p.21).

**Model** In general, a CCA contains multiple state variables–*e.g.* congestion window or link capacity estimate–that determine its behavior. The CCA reacts to multiple *events*–*e.g.* the arrival of an acknowledgment or the determination of a packet loss. A comprehensive model of CCAs would thus determine expressions, or *handlers*, to update each state variable upon the occurrence of each event. With Abagnale , we focus on a specific but important sub-problem: synthesizing an expression to update the congestion window upon an acknowledgment's arrival. While we believe Abagnale 's technique generalizes to synthesizing expressions to update other known state variables for other events, we do not evaluate such scenarios in this paper. We also leave synthesizing expressions to update a CCA's packet pacing rate to future work. A further generalization might consider *unknown*, or hidden, state variables, that affect the CCA's behavior. We do not address these cases with Abagnale and leave them to future work. However, we note that Abagnale 's model can in some cases (*e.g.* BBR, see Section 3.4.2 → p.26) nevertheless synthesize handlers that approximate the CCA's behavior despite not modeling hidden state variables.

Figure 3.1 → p.15 shows an overview of Abagnale 's synthesis process. In Section 3.3 → p.18, we discuss how Abagnale searches a space of possible programs to identify candidate CCAs to replicate the ground truth CCA. Recall that this search space is intractably large; without both optimizations and approximations we discuss, the synthesis process does not succeed. However, before discussing the search process, we must first identify a method to determine whether a candidate CCA has replicated a ground truth CCA (Section 3.2.1 → p.16); second, we must determine what space of possible programs to search (Section 3.2.3 → p.17).

### 3.2.1 Evaluating Candidate CCAs

Like CCA classifiers [56, 100, 130], we measure the observable CWND and other signals (RTT, packet rate, *etc.*) over time from a packet trace. Similarly to Mister880 [45], we execute each candidate handler function in simulation given the same events and inputs observed for the ground-truth CCA. For each packet received in the collected trace, we execute the candidate handler function, and, based on resulting CWND value, decide whether to send the next packet. Once this is done for the whole trace, we have a second time-series of the CWND produced by that handler. We call the trace resulting from this simulation the *synthesized trace*. Unlike Mister880 [45], we compare two candidate CCAs by computing the *distance* (Section 3.3.3 → p.21) between the two CWND traces. A handler is a better candidate than another handler if it has a lower distance to the ground truth trace. Using a distance measure rather than assuming the best handler will produce identical outputs to the ground truth CCA allows us to handle *measurement*

*noise* – e.g., unobserved losses or jitter between our vantage point and the server. In addition, using a distance measure allows us to use three optimizations that tune Abagnale 's exploration of the search space; we explain how we generate candidate handlers in Section 3.3.2 → p.20, how we select a distance metric in Section 3.3.3 → p.21, and finally how we use these pieces to explore the search space in Section 3.3.4 → p.21.

### 3.2.2 Trace Collection

An additional necessity in evaluating candidate handler functions is ensuring that we have *representative traces* for that handler function. We need a wide range of measurements of the ground truth CCA which capture the CCA's behaviors under varying conditions and events, otherwise we risk 'overfitting' to one particular trace and set of conditions (for example, we might return a handler that simply returns a constant CWND, the trace's BDP) [116]. To avoid this, we provide Abagnale with a *diverse* set of testing environments in order to observe more behaviors from the ground truth CCA while also doing so in a way that does not result in *too much data* for Abagnale to take in.

To achieve trace diversity, we use a controlled testbed from a prior study [131] to configure a virtual network with RTTs ranging between 10 to 100ms and bandwidth between 5 and 15Mbps. Collecting traces representing a wide range of conditions enables Abagnale to choose better candidate handlers. Indeed, when we attempt to synthesize Cubic based on traces, Abagnale fails to find a correct function when only given traces from any one configuration of RTT and bandwidth: it is only when we provide traces from a range of settings that Abagnale correctly synthesizes a Cubic function.

While providing a large number of traces improves fidelity, evaluating the distance function as described above in Section 3.2.1 → p.16 requires a fixed amount of work per packet in each trace. Thus, evaluating every packet of every trace is too costly. Instead, in Abagnale we first split flow traces into trace segments corresponding to periods between loss events. We infer loss events by searching for instances of triple-duplicate-ACKs. Abagnale increases the number of trace segments considered per candidate expression in each iteration of its refinement loop (Section 3.3.4 → p.21). Given a number of trace segments to consider in an iteration, Abagnale first randomly selects half the desired number of trace segments. For each of these sampled segments $t$, Abagnale then selects the remaining un-picked segment with the highest distance from $t$. This trace segment selection strategy makes it more likely to sample a diverse set of trace segments representing many network conditions; this in turn helps Abagnale avoid handlers that "over-fit" to specific traces.

### 3.2.3 DSL Curation

Finally, Abagnale takes as input a *domain-specific language* (DSL) from which it will produce an expression to match the input packet traces. Abagnale supports a large set of congestion signals and variables, based on prior work on frameworks for developing CCAs [105, 133]. This high expressivity comes at a cost: including all known signals and combinations of signals in the DSL would make the search space intractably large. Instead, we provide as input to Abagnale a CCA-family-specific DSL which uses only a subset of signals and operations. Of course, the user could use different DSLs in separate Abagnale invocations on the same traces.

$$
\begin{array}{rcll}
\mathcal{CS} & ::= & \textit{mss} \mid \textit{acked-bytes} \mid \textit{time-since-loss} & \textit{Congestion Signal} \\
 & \mid & {}^{\star}\textit{rtt} \mid {}^{\star}\textit{min-rtt} \mid {}^{\star}\textit{max-rtt} & \\
 & \mid & {}^{\star}\textit{ack-rate} \mid {}^{\star}\textit{rtt-gradient} & \\
\mathcal{N} & ::= & \textit{cwnd} \mid \mathcal{CS} \mid \textit{constant} & \textit{Numeral} \\
 & \mid & \mathcal{N}+\mathcal{N} \mid \mathcal{N}-\mathcal{N} \mid \mathcal{N}\times\mathcal{N} & \\
 & \mid & \frac{\mathcal{N}}{\mathcal{N}} \mid \mathcal{B}\,?\,\mathcal{N}:\mathcal{N} \mid {}^{\circ}\mathcal{N}^{3} \mid {}^{\circ}\sqrt[3]{\mathcal{N}} & \\
\mathcal{B} & ::= & \mathcal{N}<\mathcal{N} \mid \mathcal{N}>\mathcal{N} \mid \mathcal{N}\,\%\,\mathcal{N}=0 & \textit{Boolean}
\end{array}
$$

Figure 3.2: Non-colored elements are in the base Reno-DSL, violet-colored °-prefixed elements are extensions for the Cubic-DSL, orange-colored ⋆-prefixed elements are extensions for the rate/delay-DSL.

$$
\begin{array}{rcl}
\textit{reno-inc} & := & \frac{\text{ACKed}\times\text{MSS}}{\text{CWND}} \\
\textit{vegas-diff} & := & \frac{(\text{RTT}-\text{minRTT})\times\text{ack-rate}}{\text{MSS}} \\
\textit{htcp-diff} & := & \frac{\text{RTT}-\text{minRTT}}{\text{maxRTT}} \\
\textit{RTTs-since-loss} & := & \frac{\text{time-since-loss}}{\text{RTT}}
\end{array}
$$

Table 3.1: Pre-defined macros used in Abagnale 's DSLs

Indeed, groups of CCAs often use similar signals (e.g., Reno/Westwood/etc, BBR/Vegas/Veno). Hence, we only include these signals if called for in a chosen 'sub-DSL.' For example, the 'rate/delay' sub-DSL includes signals for the RTT and ACK rate, used by CCAs like BBR, Vegas, and Veno, but not by CCAs like Reno or Cubic. We use existing CCA classifiers to hint which sub-DSL Abagnale should use for a given set of traces. We show in Section 3.5.3 → p.31 that this strategy picks DSLs similar to those we would have chosen manually.

Figure 3.2 → p.18 shows how input DSLs can vary. Almost all useful DSLs will include the black-colored elements (*e.g.*, arithmetic operators), while the user may choose to exclude uncommon operators such as cube-root. Some of the operators are derived from others, *e.g.*, we offer a built-in EWMA operation since it is commonly used in CCA evaluation. While we could ask Abagnale to 'discover' the EWMA operation during the synthesis process, we have found that encoding common macros in the DSL enables Abagnale to more effectively identify fruitful candidate expressions. Table 3.1 → p.18 lists the macros used to simplify CCAs' commonly used expressions in Abagnale 's DSLs. *reno-inc* is Reno's CWND increment of one MSS per sent packet. *vegas-diff* is Vegas's estimation of the difference between the expected and the actual sending rate [21]. *htcp-diff* is the variation in RTT, as used by H-TCP [88]. *RTTs-since-loss* is the time since the last loss event scaled by the current RTT estimate, as used by BBR [26].

## 3.3 Exploring the Search Space

Once the input DSL and traces are defined, Abagnale explores the resulting search space. The main challenge is the search space size: even when considering a sub-DSL, the search space would remain intractable if traversed naively. To cope with the size of the search space, we adopt the

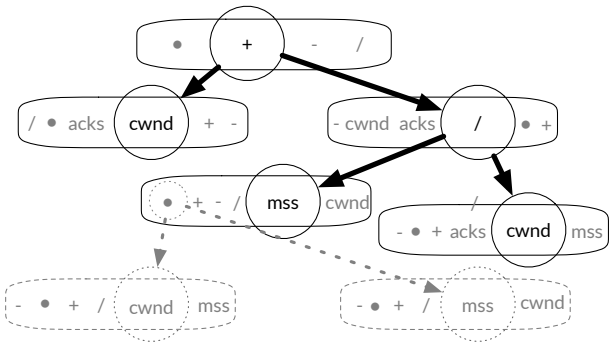



Figure 3.3: Visualizing the search through an AST.

following four key techniques:

- **DSL constraints:** First, we place constraints on the members of the DSL we will *enumerate* (*i.e.* consider to be candidate CCAs) (Section 3.3.1 → p.19).
- **Constant sampling:** Second, candidate CCAs will contain constant values, and the correct setting of those values is susceptible to trace noise, so rather than considering each setting separately, we consider a random sampling of constant assignments as a single set (Section 3.3.2 → p.20).
- **Bucketization:** Third, we devise a divide-and-conquer approach that splits the search space into partitions; searching through these partitions independently is faster than searching the entire space as a whole (Section 3.3.4 → p.21).
- **Bucket prioritization:** Fourth, we identify a bucketing metric that allows us to consider entire buckets of CCAs *as a whole* and prioritize which buckets to explore deeper into (by splitting them into sub-buckets) (Section 3.3.4 → p.21).

### 3.3.1 DSL Enumeration

Our search space is composed of all the trees of operations (*i.e.*, abstract syntax tree, or AST) that can be built by combining the DSL components. These ASTs may not correspond to concrete event handlers, since they may have nodes assigned the DSL component *constant* that do not receive a final value until the simulation phase (Section 3.3.2 → p.20). Until then, we call these incomplete handlers *sketches*. Possible sketches from this DSL include the Reno (Equation (3.1) → p.19) and Vegas (Equation (3.2) → p.20) sketches. The orange-colored and ⋆-prefixed elements represent the components specific to the Vegas-DSL, and $c_1, c_2, c_3, c_4, c_5$ represent undefined constants. Figure 3.3 → p.19 visualizes searching an AST within the Reno-DSL.

$$cwnd \mathrel{+}= mss \cdot \frac{acked}{cwnd} \tag{3.1}$$

$$cwnd \mathrel{+}= \left(\frac{cwnd}{{}^{\star}min\text{-}rtt} - {}^{\star}ack\text{-}rate < c_1\right) \; ? \\ c_3 \cdot mss \; : \\ \left(\frac{cwnd}{{}^{\star}min\text{-}rtt} - {}^{\star}ack\text{-}rate > c_2\right) \; ? \; c_4 \cdot mss : c_5 \tag{3.2}$$

The number of sketches we can build from DSL components is infinite – we could simply keep growing the expression tree – so we limit the search space by limiting the maximum depth of the AST. For a fixed depth, the number of possible sketches grows exponentially with the number of DSL components. This makes the search space very large: if we consider trees of maximum depth 7 with the 25 components of the DSL in Figure 3.2 → p.18, the correct sketch is one out of a universe of $\approx 10^{150}$.[1]

Dealing with large search space sizes is common in synthesis literature, so we start by leveraging techniques from previous work. First, we rely on an SMT formula to extract from this space only sketches that type-check [41, 62, 95]. We also specify that sketches should not be arithmetically simplifiable using the sympy [98] library. Second, we impose CCA-specific constraints: the output should have the correct units (in this case bytes) and should not monotonically decrease (since any reasonable CCA must grow the window at some point). We iteratively query an SMT solver to explore the search space with the resulting formula. Each solution to the SMT formula is a sketch with the desired properties. After obtaining a sketch, we can ask the SMT solver for a different sketch by adding a constraint that blocks the previous solution.

### 3.3.2 Concretizing Enumerated Sketches

The sketches the enumeration process returns can have unassigned constants, e.g., $c1, c2, ...$ in Equation (3.2) → p.20. To evaluate a candidate sketch, we first need to produce a concrete handler function with no unassigned constants from the sketch. One way to concretize the constants in a sketch is to try different concrete number values for each constant. The problem with this combinatorial search is that the set of concrete handlers associated with a single sketch grows exponentially: the number of ways we can assign $k$ variables with $n$ values is $k^n$. For example, the Vegas sketch has 5 unassigned variables, so if we considered 10 different values for each, we would get ≈10 million handlers for just one sketch out of millions. Ideally, these constants should be able to take any real value, but it would be prohibitively slow to solve the resulting real-valued optimization problem for each sketch. Rather, Abagnale focuses on identifying promising sketches with *approximate* concretization. That is, we limit the values constants can take to a small set of values observed in known CCAs to estimate a sketch's distance from a trace fragment. This strategy makes our approach incomplete, i.e., there are handlers that we will not explore, dependent on the predefined values chosen for the constants. However, in our experiments, this did not prevent Abagnale from returning useful sketches. After Abagnale returns a handler, it is possible to evaluate this handler's sketch using a broader set of constant values.

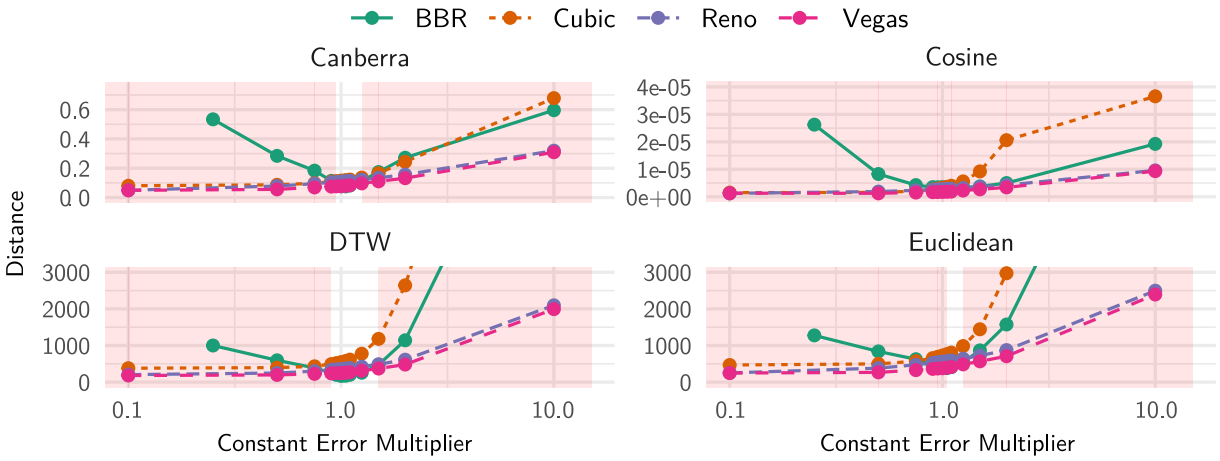



Figure 3.4: Comparison of distance metrics' tolerance to error in constant values, for traces from the BBR CCA. Red-shaded regions indicate that a synthesized CCA other than BBR had a smaller distance to the traces. Note the x-axis, showing the amount of error we introduce to the fine-tuned constant values, is in log-scale.

### 3.3.3 Selecting a Distance Metric

How should Abagnale determine whether a candidate handler matches a set of traces? Because of our optimization formulation of the synthesis problem, we require a distance metric. We consider various methods for computing the distance between two traces. Importantly, as described above, Abagnale cannot exhaustively evaluate all assignments of constant values since doing so would make the search space too large. As a result, it is important to select a distance metric that tolerates error to the greatest extent possible.

In Figure 3.4 →p.21, we show how four distance metrics respond to errors in handlers' constant values. We use packet traces corresponding to BBR,[2] and we calculate the distances using in-DSL expressions for BBR, Cubic, Reno, and Vegas written by a domain expert. We introduced a fixed amount of multiplicative error, from 0.1 to 10, to each constant in each handler, and measure the resulting handler's distance from the trace. We then determine for each amount of error whether the correct CCA's handler remained the closest to the trace. If another CCA's handler was closer using that distance metric, we shade the background in red. We observe that the Dynamic Time Warping (DTW) [15] distance remains correct for the widest range of constant error. This distance metric is alignment-based; *i.e.* it seeks to correct for temporal shifts between curves. Unfortunately, DTW distance is significantly more expensive to compute than Euclidean distance. For Abagnale , we find that in most cases, DTW's improved resilience to constant error is worth the additional runtime, and configure Abagnale to use it unless otherwise described.

### 3.3.4 Guiding the Search

Even when using curated sub-DSLs and the above techniques, Abagnale could not synthesize CCAs that use more complex DSLs. We tackled this problem by (1) splitting the search space into

[1] Our physical universe has $\approx 10^{79}$ atoms, so the sketch's universe is much larger.

[2] When selecting a primary distance metric, we additionally evaluated other CCAs' traces and other distance metrics, but we elide those results for brevity.

disjoint subspaces, and (2) using our distance metric (Section 3.2.1 →p.16) to prioritize subspaces that are more likely to generate CCAs with lower distance to the input traces.

**Partitioning the search space** Partitioning the space facilitates parallelization by allowing Abagnale to use a specialized solver invocation per bucket and performing the search across buckets in parallel [22]. This makes enumeration significantly faster not only because it can take advantage of multiple cores, but also because each solver is searching over a smaller space. Recall that (i) the solver query grows every time we query the solver for a new sketch, since we must exclude all previously returned sketches to get a new one; and (2) solver execution time grows rapidly with query size. Thus, smaller sub-search spaces can use smaller queries, which is faster. We call each of these sub-search spaces a *bucket*.

**Bucketing metric** How should we divide the space into buckets? We must ensure that each sketch in the DSL belongs to a single bucket according to a *bucket discriminator*: a metric we can express in the solver query which will cause it to only enumerate sketches in the bucket corresponding to the metric's value.

We want to choose a metric such that two sketches in the same bucket share not only some *structural* similarity—so that we can easily encode it into a dedicated solver—but also share some *behavioral* similarity. Thus, our approach is to pick a discriminator that ensures *behavioral* similarity between sketches in the same bucket, so that we can sample N sketches from all the buckets at the beginning, simulate them using the procedure described in Section 3.3.2 →p.20, and assign each bucket a score based on how close the traces in the sample were to the desired behavior. Using these scores, we order the buckets from most promising to least promising, drop the least promising buckets, and repeat this loop with an increased size of samples N and a reduced number of buckets.

Of course, this approach requires a bucketing metric that preserves behavioral similarity. We considered four such metrics: (1) fixing the operations of the nodes of the first 3 levels of the tree, (2) limiting the subset of DSL operators (addition, multiplication, power, etc) the sketch can use, (3) limiting the subset of congestion signals and state variables (delay gradient, minimum RTT, time since the last loss, etc) the sketch can use, and (4) fixing *all* DSL elements the sketch can use (so, a combination of (2) and (3)). We found that limiting the subset of DSL operators (addition, multiplication, power, etc.) the sketch can use–option (2) above–provided the best results: this metric was easy to integrate into the enumeration process and allows Abagnale to have an independent SMT solver for each bucket and enforce the bucket's metric value without much overhead in the formula.

**Search prioritization** We use our bucketing metric to guide the search. Abagnale uses a refinement loop as shown in Algorithm 1 →p.23. The algorithm takes as input the DSL, and each bucket's discriminator. It also takes the initial values of $N$, the number of samples we will consider from each bucket and $k$, the number of buckets that are retained to the next iteration. In each loop iteration, we sample $N$ sketches from each bucket (line 4). In line 5, we run the simulation procedure on each of them and save the best distance that a concrete handler built from that sketch can achieve. Then each bucket is assigned a score in line 6, equal to the minimum of these distances. Having computed all scores for all buckets, we sort them in line 6 from most promising to least promising, *i.e.* by increasing score value. Then, in line 8, we refine the search

**Algorithm 1** Abagnale 's refinement Loop

```
1: procedure SynthesisLoop(DSL, buckets, N, k)
2:     while buckets not exhausted do
3:         for all bucket ∈ buckets do                          ▷ in parallel
4:             samples ← enumerate N sketches from bucket
5:             distances ← distance for each sketch ∈ bucket
6:             bucket-score ← min(distances)
7:         end for
8:         buckets ← only-top-k(buckets, bucket-scores)
9:         N ← N · 2^3
10:        k ← k/2
11:    end while
12: end procedure
```

space by selecting only the most promising buckets to be explored further. *only-top-k* will return the subset of the buckets whose scores are lower than or equal to the $k$-th bucket score. This means that, if there are no ties, $k$ buckets are retained to the next iteration of the loop. Before going on to the next iteration of the loop, we update $N$ and $k$. Now that we know we are looking into the more promising subset of all buckets, we want to dig deeper into each one to find as good a handler as possible, so $N$, the sample size for each bucket, is increased by 8 times in l.9. As we get deeper into each bucket, we increase the trust in our scoring, so we want to get more and more conservative in how many buckets keep in the search. In l.10 we update $k$ to half its previous value. Since we expect each iteration to evaluate fewer handlers, we can afford to compute distances using more traces, so we also increase the number of distinct traces being used by two. Abagnale repeats this loop until either (1) there is one bucket left, in which case we exhaustively enumerate it and return the best handler within it, or (2) $N$ grows larger than the size of the largest bucket still in consideration, which means all buckets have already been exhaustively enumerated. During the whole loop duration, Abagnale stores the lowest distance handler it has found thus far, so if the user interrupts the loop (e.g., with a timeout), Abagnale will return that handler.

## 3.4 Results

We now show Abagnale 's synthesized expressions across two sets of packet traces. The first set of traces corresponds to the 16 CCAs with implementations distributed with the Linux kernel: BBR [26, 76], Cubic [65], Vegas [21], Reno [73], BIC [135], CDG [67], HighSpeed, H-TCP [88], Hybla [25], Illinois [91], LowPriority [85], NV [20], Scalable [83], Veno [55], Westwood [96], and YeAH [11]. These CCAs are implemented as Linux kernel modules in ~50–500 lines of C. The second set of CCAs is a publicly available dataset of novel CCAs written by students at a US university as part of a graduate-level networking class. These CCAs are implemented in between 50–150 lines of C++.

Abagnale produces arithmetically simple expressions—*i.e.* with a maximum AST depth of 5, which is significantly simpler than the original implementations—for all these CCAs.

**Implementation** To synthesize these expressions, we implemented Abagnale on Python 3.11.7. We ran all experiments using Intel Xeon Gold 6226R with 256GB of RAM, Intel Xeon

| CCA | Synthesized cwnd-ack handler | DTW distance | Fine-tuned cwnd-ack handler | DTW distance |
|---|---|---|---|---|
| BBR | $2 \times \text{ack-rate} \times \text{minRTT} + \{\text{CWND}\%2.7 = 0\}\ ?\ 2.05 \times \text{CWND} : \text{MSS}$ | 195.21 | $\text{minRTT} \times \text{ack-rate} \times (\{\textit{RTTs-since-loss}\%8 = 0\}\ ?\ 2.6 : 2.05)$ | 143.08 |
| Reno | $\text{CWND} + .7 \times \textit{reno-inc}$ | 18.84 | $\text{CWND} + .7 \times \textit{reno-inc}$ | 18.84 |
| Westwood | $\text{CWND} + \textit{reno-inc}$ | 86.99 | tel $\text{CWND} + .68 \times \textit{reno-inc}$ | 12.72 |
| Scalable | $\text{CWND} + .37 \times \textit{reno-inc}$ | 26.25 | $\text{CWND} + .37 \times \textit{reno-inc}$ | 26.25 |
| LP | $\text{CWND} + .68 \times \textit{reno-inc}$ | 18.2 | $\text{CWND} \times (\{\textit{htcp-diff} > .5\}\ ?\ .5 : 1) + .68 \times \textit{reno-inc}$ | 18.2 |
| Hybla | $\text{CWND} + 8 \times \text{RTT} \times \textit{reno-inc}$ | 35.77 | $\text{CWND} + 8 \times \text{RTT} \times \textit{reno-inc}$ | 35.77 |
| HTCP | $\text{CWND} + \textit{reno-inc}$ | 56.24 | $\text{CWND} + \textit{reno-inc} \times \{\textit{htcp-diff} < .25\}\ ?\ 1 : .2$ | 54.53 |
| Illinois | $\text{CWND} + 1.3 \times \textit{reno-inc}$ | 397.99 | $\text{CWND} + .3 \times \textit{reno-inc} + 5 \times \textit{reno-inc} \times \textit{htcp-diff}$ | 467.81 |
| Vegas | $\text{CWND} + \{\textit{vegas-diff} < 1\}\ ?\ .7 \times \textit{reno-inc} : 0$ | 24.36 | $\text{CWND} + \{\textit{vegas-diff} < 1\}\ ?\ .7 \times \textit{reno-inc} : \{\textit{vegas-diff} > 5\}\ ?\ -.7 \times \textit{reno-inc} : 0$ | 20.21 |
| Veno | $\text{CWND} + \textit{reno-inc} \times \{\textit{vegas-diff} < .7\}\ ?\ .35 : .16$ | 9.26 | $\text{CWND} + \textit{reno-inc} \times (\{\textit{vegas-diff} < .7\}\ ?\ .35 : .16)$ | 9.26 |
| NV | $\text{CWND} + \{\textit{vegas-diff} < 1\}\ ?\ .7 \times \textit{reno-inc} : 0$ | 58.1 | $\text{CWND} + \{\textit{vegas-diff} > 1\}\ ?\ .7 \times \textit{reno-inc} : \{\textit{vegas-diff} > 5\}\ ?\ -.7 \times \textit{reno-inc} : 0$ | 479.39 |
| YeAH | $\text{CWND} + \textit{reno-inc} \times \{\textit{vegas-diff} > 5\}\ ?\ .3 : 1$ | 33.41 | $\text{CWND} + \textit{reno-inc} \times \{\textit{vegas-diff} > 5\}\ ?\ .3 : 1$ | 33.41 |
| Cubic | $\text{CWND} + \text{time-since-loss}^3$ | 3580.67 | $\text{wmax} + (8 \times \text{time-since-loss} - \sqrt[3]{(.24 \times \text{wmax})})^3$ | 41.74 |
| Student 1 | 88 | 196.06 | – | – |
| Student 2 | $\left\{\frac{\textit{vegas-diff}}{\text{minRTT}} < 5\right\}\ ?\ \text{CWND} + \text{MSS} : \text{MSS}$ | 12203.07 | – | – |
| Student 3 | $.8 \times \frac{\text{ACKed}}{\text{minRTT}}$ | 7698.63 | – | – |
| Student 4 | MSS | 217.56 | – | – |
| Student 5 | $2 \times \text{MSS}$ | 32.69 | – | – |
| Student 6 | $\frac{\textit{cwnd}+150\times\text{MSS}}{\text{delay-gradient}}$ | 24406.14 | – | – |
| Student 7 | $\text{CWND} + \frac{2\times\text{ACKed}}{\text{RTT}}$ | 17541.93 | – | – |

Table 3.2: Results of running Abagnale on different input traces. The first column, "CCA" shows the ground truth, i.e., the algorithm that was running when the set of traces used for this task was collected. The second column shows Abagnale 's output cwnd-ack handler expression, and the sum of DTW distances between synthesized traces computed with this handler and the respective ground truth traces. The third column shows a domain expert's attempt at handwriting a cwnd-ack handler expression from the source code of the respective CCA, as well as the sum of DTW distances computed with these handlers.

E5-2630 v2 with 64GB of RAM, and Intel Xeon Silver 4110 CPUs with 64GB of RAM, with different numbers of cores and RAM. We used Z3 [102] version 4.8.10 for all SMT queries. Since scoring handlers (Section 3.3.3 → p.21) is a parallelizable task, we used Ray [101] to distribute the synthesis tasks among cores across different machines. For every experiment, we explored different depths of the same DSL on different parallel machines (we evaluate the impact of DSL depth in Section 3.5.3 → p.31). We ran all synthesis tasks to completion (*i.e.*, until Abagnale returned a result); in all cases, this took less than 48 hours per depth per CCA.

### 3.4.1 Results Overview

We show a summary of CCAs we attempted to synthesize in Table 3.2 → p.24. Each row in the table refers to an analysis of traces derived from a single CCA, identified in the first column. In the second column we show the expression Abagnale synthesizes, as well as the sum of distances between the synthesized traces and the respective collected traces. Note that these expressions

| CCA | Classifier output |
|---|---|
| BBR | BBR |
| Reno | Reno |
| Westwood | **Vegas** |
| Scalable | Scalable |
| LP | Unknown (Vegas) |
| Hybla | **BBR** |
| HTCP | HTCP |
| Illinois | Illinois |
| Vegas | Vegas |
| Veno | **YeAH** |
| NV | Unknown |
| YeAH | YeAH |
| Cubic | Cubic |
| Student 1 | Unknown (CDG, Vegas) |
| Student 2 | Unknown (CDG, Vegas) |
| Student 3 | Unknown (Scalable, Vegas) |
| Student 4 | Unknown (CDG, NV) |
| Student 5 | Unknown (CDG, Vegas) |
| Student 6 | Unknown (CDG, Vegas) |
| Student 7 | Unknown (CDG, Vegas) |

Table 3.3: Result of running a classifier (Gordon [100] for the Kernel algorithms, or CCAnalyzer [130] for the students algorithms) for the CCA. The CCA name in parenthesis after "Unknown" is the CCA that the classifier identified as closest, despite the output being Unknown. We color classifier outputs orange if correct and **purple** if incorrect.

use only the default constant values listed in Section 3.5.1 → p.30, but we arithmetically simplify the expressions where possible for readability. Abagnale computes these distance values shown over the trace segments used to synthesize each CCA. Since Abagnale synthesizes different CCAs using different sets of traces, the distance values shown for these handlers are not comparable across CCAs, i.e., across rows.

Within the same row, the difference between the synthesized handler distance and the fine-tuned handler distance gives us an idea of how close the behavior of these two handlers is. In rows where the synthesized handler distance is the same as, or very close to, the fine-tuned handler distance (e.g., BBR, Reno, Scalable, LP, Hybla, HTCP, Illinois, Vegas, Veno), Abagnale outputs a handler that closely matches the behavior of the fine-tuned handler in the traces used for synthesis. When the synthesized handler distance is much higher than that of the fine-tuned handler (e.g., Cubic), Abagnale 's refinement loop was unable to select the correct bucket for exploration, and the fine-tuned handler was never evaluated.

Before running Abagnale on the collected traces, we run a CCA Classifier, Gordon [100], on the same traces. Gordon establishes multiple connections to the server, and classifies each connec-

tion as running one of its known CCAs (BBR, Cubic, BIC, HTCP, Scalable, YeAH, Vegas/Veno, Reno, Illinois, and Westwood), or as "Unknown". Table 3.3 →p.25 shows Gordon's output. If Gordon determines that a majority of the test connections match a single CCA, we list that CCA in the table. If Gordon only matches a minority of the connections, we report that output in parentheses. Finally, when we ran Gordon on New Vegas ("NV" in the table), it classified all of the connections as "Unknown", so we report this result as "Unknown".

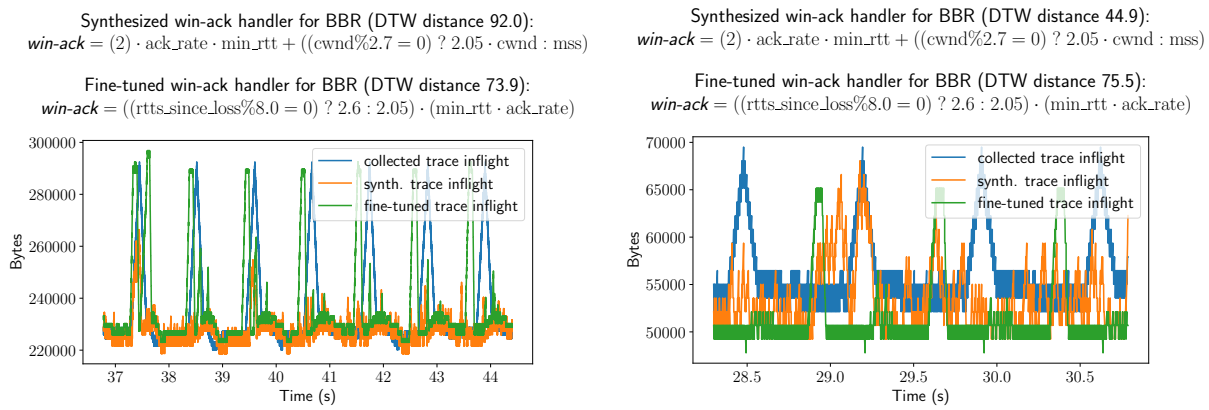
The CCAs from the class project dataset are implemented with a UDP transport, which Gordon does not support. Thus, for these, we run CCAnalyzer [130]. As expected (since these are all novel algorithms), CCAnalyzer outputs "Unknown" for all algorithms. Since this classifier uses a distance metric to compare with its known algorithms, we can also ask it for the closest known algorithms to the trace behavior. CCAnalyzer reported CDG and Vegas as the closest CCAs to all the students' algorithms but one, for which it reported Vegas and Scalable. As before, we use these classifier results to pick the DSLs we run Abagnale with.

**Fine-Tuned Handlers** We emphasize that it is not Abagnale 's goal to reproduce the CCA implementation that resulted in the collected packet trace; as described previously, these implementations can comprise hundreds of lines of code, covering both logic irrelevant to the CCA's behavior such as custom congestion signal measurement logic and edge cases that Abagnale does not attempt to capture. Thus, rather than using the implementation that generated the trace as our ground truth, we use fine-tuned versions of the synthesized handlers. To write these fine-tuned handlers, we used the synthesized expression as a starting point and used domain knowledge of the CCAs' implementations and descriptions of their behavior to write a handler with the same depth and within the same DSL that captures the CCA's behavior. During this task, we found (anecdotally) that it is easy to miss implementation details, and it is hard to simplify handlers' computations to fit the DSL, so these handlers are also not a perfect match of the CCA's behavior. In fact, as Table 3.2 →p.24 shows, some fine-tuned handlers have a greater distance to the collected traces than the respective synthesized handler. In these cases, our understanding of the original CCA's implementation from analyzing its implementation mismatched the CCA's actual behavior in the traces. This was either because we overestimated the impact of edge-case scenarios or there was a mismatch between descriptions of the CCA and its implementation. We show the fine-tuned handler for each CCA in the third column of Table 3.2 →p.24. We used these fine-tuned handlers to understand the handlers Abagnale produces more deeply. We show an evaluation of Abagnale 's accuracy relative to these fine-tuned handlers in Section 3.5.2 →p.31.

### 3.4.2 BBR

BBR's [26] core behavior consists of periodic pulses that probe for additional bandwidth, called "PROBE_BW" mode. In most implementations, these pulses are controlled by a state variable that determines whether the sending rate and congestion window are set above, below, or at the estimated bottleneck rate. Of course, Abagnale does not support hidden state variables and can only produce closed-form expressions, so we use BBR as a case study to better understand whether Abagnale can capture such algorithms' behavior.

In this case, Abagnale synthesizes pulsing behavior in a different way: if the CWND is an even number, it sets the CWND to $8 \times CWND$, and otherwise uses $2.15 \times$ minRTT $\times$ ack-rate. This expression captures BBR's "CWND gain" feature that seeks to maintain a standing queue [131]. By periodically increasing the CWND beyond this value, the handler will achieve




(a) A trace in which the fine-tuned handler achieves a lower distance than the synthesized one.

(b) A trace for which the fine-tuned handler achieves a higher distance than the synthesized one.

Figure 3.5: Even though the handler handwritten by a domain expert based on the BBR kernel implementation is a better visual match to the collected BBR trace, the synthesized trace with random "spikes" has a lower distance for some traces.

the same probing property as BBR, since if the true bottleneck bandwidth is a higher value than ack-rate, then the ack-rate will increase correspondingly. Digging deeper into this synthesized handler, we compare its fidelity to the fine-tuned handler (which uses $rtt\_since\_loss \ \% \ 8 == 0$ to implement pulses) in Figure 3.5 → p.27. Indeed, in Figure 3.5a → p.27 we see that this fine-tuned handler achieves a lower DTW distance than the synthesized handler. This matches visual intuition; in this trace, the fine-tuned handler's pulses are aligned with the pulses observed in the trace. However, this is not true for all traces. In the trace shown in Figure 3.5b → p.27, the synthesized handler achieves a lower distance. This example demonstrates a limitation of the DTW distance metric; because DTW purposely disregards temporal shifts, it is less likely that Abagnale will produce a synthesized handler that matches the original CCA's pulse behavior. Nevertheless, Abagnale produces a viable expression for BBR which is significantly simpler and more understandable than the original implementation.

### 3.4.3 Reno-Variant CCAs

The Reno, Westwood, Scalable, and LP CCAs all behave similarly to Reno, with minor modifications to the cwnd-increase function. Indeed, Abagnale produces similar expressions for traces generated by these CCAs. These CCA expressions matched the kernel implementations' behavior even with combinations of Abagnale 's default placeholder constants, and fine-tuning these CCAs only required modifying the constant values. Thus, Abagnale is able to confirm (without needing access to the source code) that these CCAs behave similarly to each other, and is able to estimate each CCA's relative aggressiveness.

Three more CCAs' traces result in handlers with the same Reno-variant structure: Hybla, HTCP, and Illinois. The objective of the Hybla CCA is indeed to increase similarly to Reno, but to scale the increase to compensate for high-delay links [25]. Indeed, the synthesized handler similarly scales the increase proportionally to the link RTT.

Surprisingly, Abagnale also returns a Reno-Variant handler when provided traces from HTCP [88] and Illinois [91]. This is unexpected because we expect both to depend on delay-

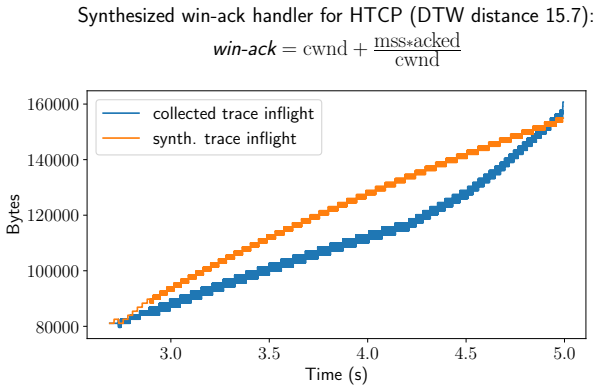



Figure 3.6: Although this HTCP trace exhibits an inflection point, a simple Reno-variant handler has a low enough distance that Abagnale does not explore more complex handlers.

based signals. Figure 3.6 → p.28 digs deeper into this result for HTCP (Illinois is similar). We find that, indeed, the trace segment depicted exhibits an inflection point in the congestion window growth. However, Abagnale is unable to find a handler that more closely represents the behavior observed in the traces, even at higher depths.

### 3.4.4 Vegas-Variant CCAs

The Vegas, Veno, NV, Illinois, and YeAH CCAs all use conditional expressions derived from a delay signal to determine their congestion window evolution. Abagnale consistently includes this feature in the synthesized handlers for these traces. Note that we include in the DSL the expression $(\text{RTT} - \text{minRTT}) \times \frac{\text{ack-rate}}{\text{MSS}}$, which is a commonly used estimator of the number of packets in the bottleneck queue. We highlight that even though the classifier is unable to identify NV, Abagnale correctly produces a Vegas-variant handler given traces from NV.

In fact, Abagnale 's output given traces from NV is identical to its output for traces from Vegas. We note that the CCAs Vegas and NV (*i.e.*, "New Vegas") use the same fundamental logic [20, 21]; their differences are only in the way they measure the number of packets in the queue. For example, NV uses a moving average of the delay and uses a hidden state variable to reduce the frequency of its window updates to once per RTT. Since Abagnale provides its own definitions of congestion signals and captures behavior rather than implementation details, it correctly returns the same handler for both these sets of traces.

### 3.4.5 Remaining CCAs

The remaining CCAs we consider from the Linux kernel are BIC, CDG, Cubic, and HighSpeed.

**BIC** The BIC CCA, at a high level, conditionally performs either binary search between the current window and the window at the time of the last loss, and linear probing [135]. The closest expressions Abagnale returns on BIC traces, meanwhile, grow the window according to the time since the last loss. As a result, we suspect that the correct handler for BIC has an AST depth too deep (with multiple levels of nested conditionals) for Abagnale to effectively explore.

**CDG** The CDG CCA calculates the probability of reducing the window based on the RTT value and randomly decides to decrease the congestion window based on this drop probability [67]. Since calculating random values is outside the input DSL, it is not possible for Abagnale to synthesize the correct handler. As a result, we do not run Abagnale on CDG traces.

**Cubic** When encoding our unit constraints as described above in Section 3.3.1 → p.19, we make a design decision to only encode integer-valued constraints, so that the enumerator formula remains a quantifier-free finite domain formula, which makes queries significantly faster. Unfortunately, as a result, Abagnale cannot unit-check cube-root operations. We thus run Abagnale with unit constraints disabled on Cubic-derived traces. Indeed, the returned expression captures a subset of Cubic's behavior - growing cubically with the time since the last loss - but this expression does not have consistent units.

**HighSpeed** The HighSpeed CCA uses logarithmic operations [52]. In the Linux kernel implementation, this is implemented with a large lookup table. As is the case with Cubic, our enumeration constraints cannot reason about exponentiation and logarithm operations. We did not run Abagnale on HighSpeed traces as a result.

### 3.4.6 Student CCAs

When we run the CCAnalyzer classifier on the student CCA dataset, the classifier indicates that all 7 CCAs have some similarity to Vegas. Unsurprisingly, when we run Abagnale on these traces many are of Vegas-variant form: Student CCAs 1, 2, 4, and 5 all modify the congestion window by comparing $(\text{RTT} - \text{minRTT}) \times \frac{\text{ack-rate}}{\text{MSS}}$ to a constant threshold value. Note that the synthesized result for the Student 5 CCA is simplifiable, since the first conditional expression is trivially false; as discussed above, Abagnale cannot reason about this simplification due to its reliance on sympy. We discuss the student CCAs in more detail, particularly exploring the impact of DSL depth, in Section 3.5.3 → p.31.

## 3.5 Evaluation

We previously described the intractably large search space Abagnale must navigate during the synthesis process. We now evaluate how well Abagnale navigates this space. We consider three aspects of Abagnale 's exploration:

1. How much of the search space does Abagnale evaluate in order to return the results in Section 3.4 → p.23? We show Abagnale 's exploration of the search space for Reno in Section 3.5.1 → p.30.

2. How far off was Abagnale from returning the fine-tuned CCA a domain expert developed with knowledge of both the synthesized handler as well as the nature of the ground-truth CCA? We discuss this in Section 3.5.2 → p.31.

3. How important are the DSL inputs to Abagnale in determining whether it will return a good sketch for an unknown CCA? We evaluate this in Section 3.5.3 → p.31.

| CCA | position after 1st iteration | position after 2nd iteration |
|---|---|---|
| BBR | 4/127 | 3/5 |
| Cubic | 7/27 | – |
| HTCP | 2/31 | 4/5 |
| Hybla | 4/7 | 1/5 |
| Illinois | 3/63 | 3/5 |
| LP | 1/63 | 1/6 |
| NV | 5/15 | 2/5 |
| Reno | 3/218 | 1/5 |
| Scalable | 1/218 | 1/5 |
| Vegas | 5/15 | 4/5 |
| Veno | 1/7 | 1/5 |
| Westwood | 1/218 | 1/5 |
| YeAH | 1/31 | 1/5 |

Table 3.4: Abagnale 's progress through the search space for the CCAs distributed with the Linux kernel.

### 3.5.1 Search Efficiency

We evaluate how efficiently Abagnale explores its search space by digging deeper into its exploration of traces produced by Reno. Recall that for this CCA, Abagnale returns the following expression with depth 3: CWND + .7 × *reno-inc*. Note that we encode *reno-inc* as a macro in Abagnale 's DSL, so that sub-expression does not increase the depth.

The Reno DSL is shown in Figure 3.2 → p.18. Between congestion signals, operators, and macros, this DSL contains 11 elements. The space of all depth-3 sketches that can be built in this DSL is then ~ 2 billion. From those, using the enumeration pruning techniques described in Section 3.3.1 → p.19, Abagnale reduces this space to 1,617 sketches. This represents the space of type-checked, unit-checked, non-simplifiable Reno-DSL cwnd-ack handler *sketches*. Each of these sketches can get expanded into concrete win-ack handlers, by filling out its holes. In total, the Reno-DSL search space has 101,000 concrete handlers.

Abagnale first partitions the search space into 218 disjoint buckets. The first iteration of the refinement loop enumerates and scores a sample of 16 handler sketches of each of the 218 buckets. To do this, Abagnale must concretize each sketch with constant values. Each sketch has between 1 and 273 completions, so in this first iteration Abagnale scores a total of 17,500 fully-populated handlers. Scoring these handlers is parallelizable, and completes in 7 minutes on the cluster described above. After this first iteration, Abagnale retains 5 of the 218 buckets. In the second iteration, it samples an additional 112 sketches (totalling 128 across the two iterations) from each bucket. 3 of the buckets contain fewer than 128 sketches in total; we enumerate those buckets exhaustively. Thus, in this iteration Abagnale scores 28,400 fully-populated handlers, in 13 minutes. After this iteration, Reno retains the 2 top buckets. These two buckets both contain

fewer than 128 sketches, so they had already been fully enumerated. So, Abagnale returns the handler with the lowest known distance, CWND + .7 × *reno-inc*. Overall, Abagnale finds this handler after exploring only about a third of the viable search space (*i.e.* the search space remaining after all enumeration constraints).

### 3.5.2 Search Accuracy

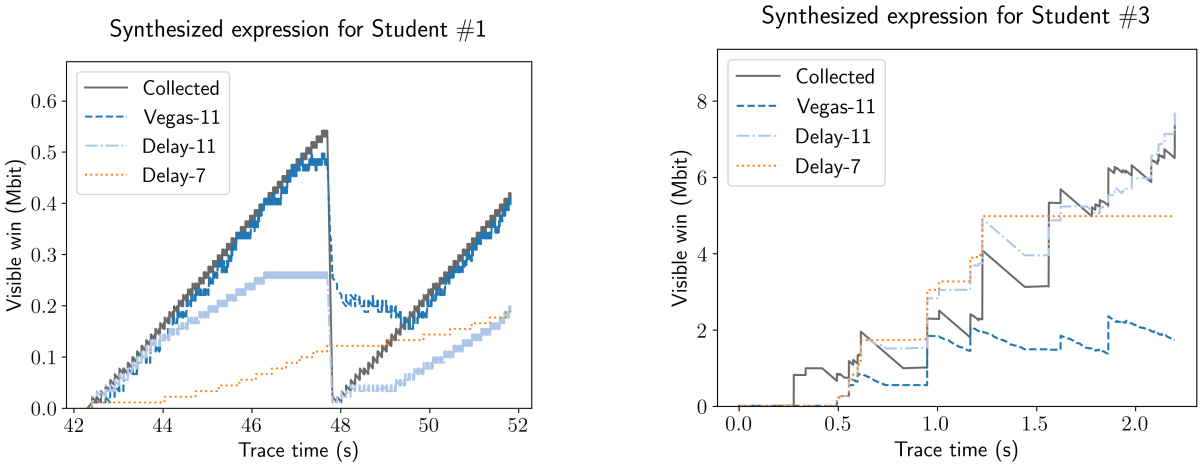
We next evaluate Abagnale 's accuracy relative to the fine-tuned handlers described in Section 3.4.1 → p.24. We measure where in the process Abagnale discarded the fine-tuned handler in favor of the one it eventually returned. Note that in some cases, such as with BBR (Section 3.4.2 → p.26), this fine-tuned handler does not have a lower DTW distance to the collected trace than the handler the synthesizer returned.

Table 3.4 → p.30 shows this result. Recall from Section 3.3.4 → p.21 that Abagnale 's search proceeds iteratively through "buckets" of the search space. The column "position after iteration 1" shows both the rank of the fine-tuned handler's bucket, and the number of possible buckets. For example, for BBR, "4 / 127" indicates that the fine-tuned handler's bucket had the fourth-lowest estimated distance out of 127 total buckets. In the first iteration of the refinement loop, Abagnale retains the top 5 buckets. Thus, in this example, Abagnale correctly discarded 122 of the 127 possible buckets. For Cubic, the first iteration of the refinement loop ranks the fine-tuned handler's bucket 7th. Since only 5 buckets are retained for the second iteration, the fine-tuned handler's bucket gets discarded. If this bucket had not been discarded, exhaustive exploration would have ranked the fine-tuned sketch at 7 / 4,794. Within these sketch completions, the fine-tuned handler would have ranked 1/36. For Cubic, unlike BBR, the fine-tuned handler has lower distance than the expression Abagnale returns. So, if the fine-tuned handler had been sampled from the respective bucket in the first iteration of the loop, Abagnale would have exhaustively searched that bucket and ultimately returned the fine-tuned handler.

The second column shows the same result after the second iteration of the refinement loop. This second iteration has more information about each bucket, because it samples 128 sketches from each of the 5 buckets (in the first iteration Abagnale samples only 16). For BBR, we see that the fine-tuned handler bucket was ranked 3rd in the second iteration, so it was not selected for exhaustive search. Similarly, for Vegas, the fine-tuned handler is in the fourth-ranked bucket after the second iteration of the refinement loop. In both cases, Abagnale exhaustively enumerates, concretizes, and scores the top-scoring buckets (which do not contain the fine-tuned handler). The fine-tuned handler's bucket in the Vegas DSL only contains one sketch; this means that, similarly to BBR, the fine-tuned handler has a higher distance than the handler Abagnale returned.

### 3.5.3 Impact of DSL Input

We use the student CCAs to evaluate the impact of the input DSL on Abagnale 's results. In Figure 3.7a → p.32, we show Abagnale 's results from three DSLs: a Delay DSL (Figure 3.2 → p.18), which includes RTT and rate signals, with constraints of depth 4 and up to 7 or 11 nodes (Delay-7 and Delay-11), and the Vegas DSL, which additionally includes a macro encoding the common sub-expression $(\text{RTT} - \text{minRTT}) \times \frac{\text{ack-rate}}{\text{MSS}}$, with depth 5 and up to 11 nodes (Vegas-11). We observe that with Delay-7, the best-scoring handler cannot capture the behavior of this CCA, while the best-scoring handler from Delay-11 starts to capture the triangular pattern. Finally,




(a) Student CCA 1 with different DSLs

(b) Student CCA 3 with different DSLs

Figure 3.7: Three different synthesized CCAs for Student 1 and Student 3 using DSLs identified by CCAnalyzer [130].

using Vegas-11 enables the macro, which frees up sketch nodes for other operations. This handler comes closest to matching the input trace's behavior.

In contrast, we show the result for student CCA #3 in Figure 3.7b → p.32. In this case, the best handler uses the Delay-11 DSL, not Vegas-11. This is because the DSL components that are part of the Vegas DSL but not the Delay DSL do not play a part in student CCA #3. This makes the search space bigger, which in turns means that if we timeout a search at any point, we are less likely to have already explored the lowest-distance sketch. So, even though the lowest-distance sketch was in both the space Abagnale explores with Vegas-11 as its input DSL and the space corresponding to Delay-11, by the time these searches timed out, Abagnale with Delay-11 had already evaluated it and saved it, but Abagnale with Vegas-11 had not.

## 3.6 Related Work

**Program Synthesis** Traditional approaches for program synthesis with examples (PBE) [42, 51, 59, 60] find a program that satisfies all given examples. Although this is the main focus of PBE research, there is some work [66] on handling cases where examples may have noise. In this scenario, prior work also formulated the synthesis problem as an optimization problem. However, they consider discrete data such as string or tabular data where the noise is limited and discrete, leaving the remaining parts intact and uncorrupted. In our case, we produce a trace of outputs for the same inputs observed in the collected trace and compare them to the outputs visible in the original trace. While we use the DTW distance to measure how good our synthesized CCA is, prior work on strings can use simpler methods like the number of failed examples or the edit distance between strings. Moreover, Abagnale uses the distance metric not only to evaluate a candidate handler's merit, but also to guide the search with our bucket prioritization strategy (Section 3.3.4 → p.21).

Smaller DSLs result in a smaller search space and faster performance but finding a small DSL expressive enough to capture the intended behavior is a challenging task. Chan et al. [27]

proposed to start with a generic large DSL and use gradient descent to find a sub-DSL that is effective for a specific problem. They train on several benchmarks and reward sub-DSLs that can quickly solve benchmarks and penalize those that fail to solve. Abagnale also has sub-DSLs for each class of CCAs from the Linux Kernel. Given a network trace, Abagnale runs a CCA classifier to map the trace to a known CCA in the Linux Kernel and uses that sub-DSL.

**Synthesis of CCAs** Mister880 [45] first proposed using program synthesis to reverse-engineer CCAs. Mister880 makes several simplifying assumptions that make it unsuitable for analyzing real CCAs. For example, it only considers a single simulated packet trace, and cannot cope with trace noise. Additionally, Mister880's simulation relies on an SMT solver for the simulation procedure, and does not scale to real-world traces, which can be hundreds of times larger than Mister880's simulated traces. It also attempts to fully enumerate the search space, which is impractical for all but the simplest CCAs. However, with Abagnale , we do take inspiration from Mister880's event-driven structure and use of distance to evaluate candidate CCAs.

Meanwhile, CCmatic [2, 3] recently proposed program synthesis techniques to produce novel congestion control algorithms that satisfy desired properties. This is fundamentally a different problem than reverse-engineering; while with Abagnale we seek to provide *fidelity* to an extant CCA, Agarwal et al.'s work need only consider a CCA's performance in some specific setting.

# 4

# Syren

## *Synthesis of API-composing Programs from Partial Traces*

**Contents**



Syren proposes an approach to synthesize general-purpose scripts from traces of its execution. These traces include sequences of messages exchanged between networked servers, logs of calls made to an Application Programming Interface (API), or traces of system calls made by a workload. Real-world traces are an incomplete view of a task: function calls with no side effects may not be recorded. A program `a=F(); b=h(a); return G(b)` where only the calls to `F` and `G` are logged may produce traces `F()=0 :: G("id-0")=true` and `F()=42 :: G("id-42")=false`, with no occurrence of `h`. The possibility of *hidden* function calls, not present in the traces, presents a significant challenge for trace-guided synthesis. In our example, the synthesizer infers from the data that there is a non-visible function that transforms `0` into `"id-0"` and `42` into `"id-42"`. These traces come from external recordings of program behavior. For example, the administrator of a cloud deployment might repeatedly follow a set sequence of steps to create a data store, and then attach an access policy to that data store. For convenience, they create a policy name by appending the string "policy" to a unique ID generated by the creation of the data store. The log of these actions kept on the cloud will contain the creation of the data store, the role, and the connection between the two. It will not contain the hidden function where the administrator bases the role name on the store's metadata.

In order to help automate such repetitive tasks, we propose a new synthesis technique, Syren. Syren is the first approach to synthesize programs from *partial* traces. It is able to infer both control flow and non-trivial hidden pure function calls with no additional input from the user. We

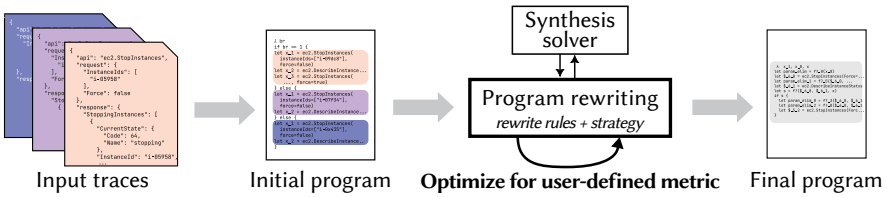



Figure 4.1: Overview of our synthesis approach.

combine optimizing program rewrites of an initial trivial solution with calls to a syntax-guided synthesizer (SyGuS) with input-output examples as specifications. Synthesis from input-output examples, also known as Programming-By-Example (PBE), has a long history of research and allows us to synthesize nontrivial functions from examples of desired input-output pairs. In Syren, we rely on a PBE synthesizer to synthesize hidden functions without additional input from the user. These hidden functions, in turn, let us perform further computation so that the rewrites can expose more intricate relations between data.

In example- and trace-based synthesis, we can assume the existence of a hidden *target program*, the program with the exact desired behavior. In the cloud administrator example above, the complete target program exists only in the administrator's mind. We can describe the *behavior* of a program as the set of traces that result from its execution on any possible input. When the synthesis specification is a set of traces, there is a trivial solution to the synthesis problem—the program that exactly reproduces the input traces. This trivial solution, though correct by construction (in the sense that it satisfies the specification), is likely not the target program that the user desires. The user probably wants a program that *generalizes* the provided traces. For example, our cloud administrator does not want a program that can reproduce all of the data store names they have used in the past, they want a program that takes a data store name as a parameter, allowing them to provide important information like the name, while saving them many repetitive clicks on a web interface. The trivially correct program is a lower bound on the behavior of all possible correct programs because it produces the minimal set of traces to be considered correct. The target program, on the other hand, is an upper bound of the desired behaviors—it would provide us with all the possible traces that are considered correct, but behaviors not exhibited by the target are undesirable. Since we do not have access to the target program, we need another way to quantify as accurately as possible how close to the target behavior a program in the space is. In other words, we need a *cost function* to efficiently traverse the program space.

Besides generalizing to traces beyond those observed (allowing more behavior), we considered another goal when building Syren's cost functions: ensuring the output program is human-readable. Like other synthesis works before us, we generally follow Occam's razor principle and favor shorter programs to achieve both goals. We evaluate Syren using two different cost functions we built, but our approach is agnostic to the cost function; a user can write different cost functions for Syren if they choose to optimize for something else. For example, a user might specify that a specific input to a given method call be generalized, that they want the smallest possible number of syntactic statements in the program, or a combination of both.

Figure 4.1 → p.35 summarizes our approach: an initial set of traces is provided, under the assumption that this set describes a task to be performed. We use that set of traces to construct an initial program. The core of our technique is the rewriting process guided by a user-defined

program metric (the cost function) that may use an underlying syntax-guided synthesizer to generate some of the program's components. We present Syren's rewrite rules and how we use the synthesizer in Section 4.4 → p.45.

While our synthesis problem could be encoded as an optimization Satisfiability Modulo Theories (SMT) problem or as a SyGuS problem, the large search space prevents off-the-shelf state-of-the-art solvers from solving it. In Syren, we efficiently traverse the search space by combining SyGuS with *program rewriting*. We start our search by building a trivial program, a lower bound on program behavior that is correct by construction. Then, we progressively rewrite it as long as we can decrease a cost function, generalizing the program and adding desired behavior, while *provably maintaining correctness*. Program rewriting is a natural approach when it comes to optimizing programs for a given cost. For example, compiler optimization and superoptimization [81, 84, 119, 136] uses rewrites to improve the performance of programs across various dimensions. A challenge to consider when developing a rewrite system is that an unsound rewrite can lead to incorrect programs. To reason about the correctness of our solution, we formally define a domain-specific language and a correctness statement that allow us to prove our rewrite rules correct w.r.t. the language and statement. The final result is guaranteed to be correct and is optimized for the user-defined metric.

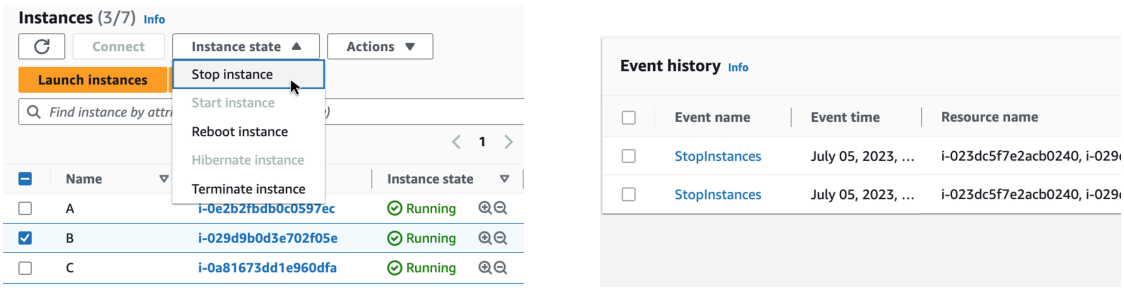
To show the practical applicability of Syren, we implemented the algorithm and evaluated it on benchmarks gathered from cloud automation, filesystem manipulation, and document edition scripts. We evaluate Syren in Section 4.6 → p.52 on a set of 54 benchmarks collected from custom tasks, existing AWS Automation Runbooks [9], Blink Automations [16] and related work [63]. We show that our approach generates scripts that accurately perform the task intended by the user for many tasks. This includes tasks with conditional control flow, loops, and hidden function calls between visible calls. We experiment with different strategies to apply rewrite rules and various metrics to optimize the final synthesized program. This shows that our approach is flexible and adaptable to different user requirements.

In summary, we make the following contributions:

- We describe a synthesis problem where the specification is a set of traces containing visible function calls, and the program to synthesize must perform those function calls and additionally implement hidden function calls and control flow.
- We implement Syren, using a new approach that combines optimizing rewrites with traditional input-output guided program synthesis.
- We evaluate Syren on a set of benchmarks built from AWS automation runbooks, previous work on API synthesis, and publicly available libraries.

## 4.1 From Partial Traces to Programs

We start by illustrating one execution of our synthesis procedure. In this example, we synthesize a cloud management script that shuts down computing instances. The specification is a set of traces built from logs collected by the cloud provider (in this case, AWS [10], one of the largest cloud providers) as the user performed the desired task manually a few times using a visual interface. The logs contain calls to the AWS API. For our synthesizer, the API calls are *visible functions* in our program, whereas the data transformations with input and output data used for these API calls will be *hidden functions*.




(a) Stopping instances in the AWS console

(b) The resulting log showing API calls

Figure 4.2: Performing and monitoring actions on the AWS console

```
λ instanceId.
let ids = list(instanceId)
let _ = ec2.StopInstances(instanceIds=ids, force=False)
let s = ec2.DescribeInstanceStatus(instanceIds=ids, includeAllInstances=True)
let status = extractInstanceStatus(s)
if status != "stopped" { let _ = ec2.StopInstances(instanceIds=ids, force=True) }
where
extractInstanceStatus := $.InstanceStatuses[0].InstanceState.Name
```

Figure 4.3: Example of a program that stops one EC2 instance.

**Motivating Example** An administrator might stop computing instances using the AWS console (shown in Figure 4.2a → p.37) by:

1. selecting the instances they want to stop,
2. clicking "Instance state", and
3. selecting "Stop instance" from the options in the drop-down menu that appears.

Clicks on the visual interface trigger calls to a cloud API, in this case, from the `ec2` service. A program could accomplish the same task by making the exact same API calls.

The program in Figure 4.3 → p.37 automates a task slightly more complex than the 3 steps described before. To perform this more complex task in the visual interface, a user would have to, after the previous steps:

4. click "refresh" to view the current status for their computing instances,
   (a) if the instance status is "stopped", then terminate the task here;
   (b) otherwise once again click "Instance state", and
5. select "Force stop instance" from the drop-down menu.

Figure 4.2b → p.37 shows the logs created in AWS by these actions. We use these logs, which from now on we will refer to as *traces*, as a specification for the synthesis problem. The synthesized program can be used to automate the task, reducing a many-click, repetitive, and error-prone task to a single press of a button. Our approach ultimately generates a program similar to the one shown in Figure 4.3 → p.37 from the traces. We now give an overview of how our approach solves this specific problem.

**Synthesis algorithm overview** We start with a set of traces, each containing the API calls made while carrying out a task. The following two traces exemplify the task described previously:

```
Trace #1 :=
  (ec2.StopInstances("InstanceIds": ["i-09dc8"], "force": false), { ... })
  (ec2.DescribeInstanceStatus("InstanceIds":["i-09dc8"]),
     {"Statuses": [{"InstanceState": {"Code":64, "Name":"stopping"}, ...}], ... })
  (ec2.StopInstances("InstanceIds": ["i-09dc8"], "force": true), { ... })
```

```
Trace #2 :=
  (ec2.StopInstances("InstanceIds": ["i-07f34"], "force": false), { ... })
  (ec2.DescribeInstanceStatus("InstanceIds": ["i-07f34"]),
     {"Statuses": [{"InstanceState": {"Code":80, "Name":"stopped"}, ...}], ...})
```

Each trace is a sequence of pairs, and each pair represents an API call: the first element shows the API method name and its inputs, and the second shows the response to that API call. Both example traces start with two calls to the API methods, `ec2.StopInstances` with the parameter `force` set to `false`, and `ec2.DescribeInstancesStatus`. In trace #1, the output of the call to `ec2.DescribeInstancesStatus` does *not* show the current status as `"stopped"`, so we see a second call to `ec2.StopInstances` with `force` set to `true`. The call to `ec2.DescribeInstancesStatus` shows the current status as `"stopped"` in trace #2, so no further calls are recorded.

The first step in our synthesis pipeline, as shown in Figure 4.1 → p.35, is to generate a trivially correct initial program, which provably generates all the input traces for some initial global state. We do this by branching the execution on the value of a fresh integer variable, $br$, and replaying each trace on a different branch. For our running example, the following initial program is generated with two branches, one for each trace:

```
λ br
if br == 1 {
   let x_1_1 = ec2.StopInstances(instanceIds=["i-09dc8"], force=false)
   let x_1_2 = ec2.DescribeInstanceStatus(instanceIds=["i-09dc8"])
} else {
   let x_2_1 = ec2.StopInstances(instanceIds=["i-07f34"], force=false)
   let x_2_2 = ec2.DescribeInstanceStatus(instanceIds=["i-07f34"])
}
```

Our approach progressively transforms the program by applying rewrite rules that decrease an optimization metric. In this example, we use a metric that measures the syntactic complexity of the program: we add 10 for each statement, 1 for each parameter, and 1 for each usage of $br$, which is a synthetic variable that should only be used by the initial program. Initially, the program has cost 62. The first rewrite applied to the program pulls the first call to `ec2.StopInstances` out of the if-statement and replaces its arguments, which were constants, with a ternary expression. A second application of the same rewrite rule extracts the call to `ec2.DescribeInstanceStatus`. Each rule reduces the cost by 9, eliminating one statement but introducing one usage of $br$. After applying these two rules, the intermediate program has cost 44:

```
λ br
let x_1_1 = ec2.StopInstances(
   instanceIds=(br==1)?["i-09dc8"]:["i-07f34"], force=false)
let x_1_2 = ec2.DescribeInstanceStatus(
   instanceIds=(br==1)?["i-09dc8"]:["i-07f34"])
if br == 1 { let x_1_3 = ec2.StopInstances(instanceIds=["i-09dc8"], force=true) }
```

To eliminate usages of `br` in the ternary expressions, we need to replace the conditional

expression `br==1` with another expression that evaluates to the same value but does not use *br*. There are two ways to achieve this: either introduce a new input parameter that takes the value of the expression, or synthesize a function that will eventually evaluate to the conditional expression value. To synthesize a (nonconstant) function, Syren considers as potential inputs all variables bound in the scope of the expression being replaced. In the first appearance of the expression `(br==1)?["i-09dc8"]:["i-07f34"]`, there are no variables bound in the scope that could be used as input to a data transformation. So, we have no choice but to introduce a new input parameter, `i_1`. Syren replaces all usages of the original expression with `i_1`. Within the conditional branches, Syren also replaces the usages of the value the expression evaluates to (considering the conditional). This results in the following program, with cost 43:

```
λ br, i_1
let x_1_1 = ec2.StopInstances(instanceIds=i_1, force=false)
let x_1_2 = ec2.DescribeInstanceStatus(instanceIds=i_1)
if br == 1 { let x_1_3 = ec2.StopInstances(instanceIds=i_1, force=true) }
```

The final rewrite for this example replaces the last conditional that depends on *br* with the output of a new data transformation $\phi$ over all variables in scope. After this last rewrite rule, the synthesized program is parametric on an implementation of that data transformation:

```
Λ ϕ. λ i_1
let x_1_1 = ec2.StopInstances(instanceIds=i_1, force=false)
let x_1_2 = ec2.DescribeInstanceStatus(instanceIds=i_1)
let c = ϕ(i_1, x_1_1, x_1_2)
if c { let x_1_3 = ec2.StopInstances(instanceIds=i_1, force=true) }
```

This rewrite is valid only if we can provide an implementation `f` for $\phi$ that ensures the program can reproduce the input traces. During the rewrite process, we maintain a mapping from the identifiers in the program to corresponding values in the traces. Then, we use these mappings to compute a set of input-output constraints that `f` must satisfy. For the traces and program in this example, we can extract the following two input-output pairs for the desired implementation `f`:

```
f(["i-09dc8"], {"StoppingInstances": [...], "ResponseMetadata": {...}},
  {"Statuses":[{"InstanceState":{"Code":64, "Name":"stopping"},...}],...}) = true
```
for trace $\tau_1$,

```
f(["i-07f34"], {"StoppingInstances": [...], "ResponseMetadata": {...}},
  {"Statuses":[{"InstanceState":{"Code":80, "Name":"stopped"},...}],...}) = false
```
for trace $\tau_2$.

We encode the problem into a syntax-guided synthesis solver to generate a solution, which yields:[1]
`f := (i_1, x_1_1, x_1_2) -> x_1_2.InstancesStatuses[0].InstanceState.Name != "stopped"`.
Substituting $\phi$ for `f` yields a program that is correct by construction, with a minimal cost of 41. This final program is syntactically equivalent to the one in Figure 4.3 → p.37.

## 4.2 Background and Definitions

In this section, we formally introduce concepts necessary to explain our approach to synthesizing scripts that compose visible side-effecting function calls with conditionals, loops, and (hidden)

[1] In practice, we need at least one more trace and its respective input/output example to synthesize this solution. If we consider only the two example traces shown, a simpler implementation is synthesized for `f`: `f := (i_1, x_1_1, x_1_2) -> i_1 == ["i-09dc8"]`.

$$
\begin{array}{lcll}
\mathcal{P} & ::= & \lambda\overline{x}.\mathcal{I}\ \texttt{where}\ \overline{f := \mathcal{F}} & \textit{Program} \\
\mathcal{I} & ::= & \epsilon \mid \mathcal{S}\ \mathcal{I} & \textit{Instructions} \\
\mathcal{S} & ::= & \texttt{let}\ x = \mathcal{E} & \textit{Pure binding} \\
 & \mid & \texttt{if}\ \mathcal{B}\ \{\mathcal{I}\}\ \texttt{else}\ \{\mathcal{I}\} & \textit{Conditional} \\
 & \mid & \texttt{retry}\ \{\mathcal{I}\}\ \texttt{until}\ \{\mathcal{B}\} & \textit{Retry until} \\
 & \mid & \texttt{for}\ x \in L\ \{\mathcal{S}\} & \textit{Foreach loop} \\
 & \mid & \texttt{return} & \textit{Return} \\
\mathcal{E} & ::= & \mathbb{A}(\overline{x}) & \textit{Visible function call} \\
 & \mid & f(\overline{x}) & \textit{Hidden function call} \\
 & \mid & \mathcal{B}\ \texttt{?}\ \mathcal{E}\ \texttt{:}\ \mathcal{E} & \textit{Ternary expression} \\
\mathcal{B} & ::= & \top \mid \bot \mid \mathcal{B} \vee \mathcal{B} \mid \mathcal{B} \wedge \mathcal{B} \mid \neg\mathcal{B} & \textit{Predicates} \\
 & \mid & x = \mathcal{C} & \textit{Value check} \\
 & \mid & x(\geq \mid > \mid \leq \mid <) y & \textit{Value comparison} \\
\mathcal{F} & ::= & ?? & \textit{Pure Function} \\
\mathcal{C} & ::= & s \in \texttt{string} \mid n \in \mathbb{Z} \mid b \in \{\texttt{true}, \texttt{false}\} & \textit{Constants}
\end{array}
$$

Figure 4.4: Core scripting language.

pure function calls. In Section 4.2.1, we define the domain-specific language (DSL) syntax of our synthesized programs. This DSL is an intermediate representation that we can easily convert to most common scripting languages. Next, in Section 4.2.2, we define *program traces*, which we use as an input specification for synthesis. Finally, in Section 4.2.3, we define the semantics of our language, which relates a program to input and output states, as well as to traces. These semantics allow us to prove properties about the manipulation of the DSL to prove our approach correct.

### 4.2.1 Core Language Syntax

Figure 4.4 presents the syntax of our core DSL. A program $\mathcal{P}$ is a function with input variables[2] $\overline{x}$ and a set of hidden functions $\overline{f := \mathcal{F}}$. The body of the program is an instruction list $\mathcal{I}$, either empty ($\epsilon$) or with statements. A statement $\mathcal{S}$ can be a simple binding, a conditional, a loop, or the instruction that marks the end of the script. A binding $\texttt{let}\ x = e_1\ \ s_2$ binds $e_1$ to $x$ in $s_2$. Conditionals $\texttt{if}\ b\ \{s_1\}\ \texttt{else}\ \{s_2\}\ s_3$ execute $s_1$ if $b$ is true, otherwise $s_2$, and then $s_3$. Our language has two forms of loops. $\texttt{retry}\ s\ \texttt{until}\ b$ (retry until) executes the instructions in $s$ at least once, until $b$ is true, or some predefined maximum number of retries is reached. $\texttt{for}\ x \in L\ \{s\}$ iterates through the list $L$, binding $x$ to each element and executing $s$.

Expressions $\mathcal{E}$ can be visible or hidden function calls. A visible function call $\mathbb{A}(\overline{x})$ is a call to some externally defined function $\mathbb{A}$ with arguments $\overline{x}$, and a hidden call $f(\overline{x})$ is a call to a pure function $f$ whose implementation $\mathcal{F}$ is defined in the program. Visible functions can have side effects on the outside world, changing the results of future calls. However, they do not change the local state of the DSL execution. The hidden functions are *pure*, and their specific syntax depends on the chosen domain.

We clearly separate pure function implementations $\mathcal{F}$ from the rest of the program for two reasons: to simplify reasoning about variable usage and whole-program rewrites, and to highlight the fact that our DSL is agnostic of the hidden functions domain. In our implementation of Syren, we consider a minimal language of hidden functions, which includes a JSONPath as well

[2]We write $\overline{x}$ to denote zero or more occurrences of $x$.

as other basic operations over strings, numbers, and Booleans. However our approach can be generalized to any other language for hidden functions, as long as expressions in that language can be synthesized.

> **Example 4.1:**
>
> The script in Figure 4.3 is an example of a script written in our DSL. The script takes a single input, `instanceIDs` and defines one data operation **`extractInstanceStatus`**. The visible functions are the API methods `StopInstances` (called twice), and `DescribeInstanceStatus`.

### 4.2.2 Program traces

Our synthesis starts from *observable* traces that can be produced by the program's execution. The *observable* traces contain only records of the visible function calls made by the program and their results; the hidden function calls do not appear in traces . Formally, a trace is a (possibly empty) finite sequence of records of all visible calls that the run of the program makes:

$$\tau := \langle\rangle \mid (\mathbb{A}(\overline{v}), e) :: \tau \quad \text{(traces)},$$

where $\langle\rangle$ is the empty trace, operator $\bullet$ performs concatenation, and $\langle\rangle \bullet \tau = \tau \bullet \langle\rangle = \tau$ for any trace $\tau$. Each record is a pair $(\mathbb{A}(\overline{v}), e)$, where the first element states the name of the function $\mathbb{A}$ and the inputs to the call $\overline{v}$; the second element, $e$, is the response to the call. $e$ is an expression in the same language as the hidden functions.

### 4.2.3 DSL Semantics

Next, we define the semantics of our language as the relation $\Rightarrow$, presented in Figure 4.5. The relation $\Rightarrow$ maps a pair of a program body and state to a triple of local state, trace and continuation token. In our DSL semantics, we refer to two different notions of state. The local state, $\sigma$, stores the bindings of every variable assigned in the program at a given point. The global state, $G$, represents external resources accessed by the visible functions in the trace. The notion of global state is necessary because visible functions are not pure functions of their inputs; depending on the resources they access, two calls to the same function with the same inputs might return different outputs. The continuation token is either `cont`, indicating that evaluation must continue, or $\lightning$, indicating that the evaluation must stop.

The rule Seq specifies how a statement $s$ followed by instructions $S'$ is evaluated sequentially and traces are concatenated. The rule Seq-S-Term handles the case where the first statement *terminates* the evaluation of the program. Ret states that the statement `return` always terminates early with an empty trace. Emp generates an empty trace, does not change the local state, and always continues evaluation. In Ite-$\top$ and Ite-$\bot$ for conditionals, either the branch with instructions $S_\bot$ or $S_\top$ are evaluated depending on whether the local state entails $b$ or not. The continuation or termination token $cr$ of the if-then-else is the same as the token in the evaluation of the branch, in particular, the statement terminates the evaluation when the branch terminates evaluation.

The rules Retry-Until-Continue, Retry-Until-Stop, and Retry-S-Term define how the retry-until statements are evaluated. Note that retry-until does not have the same semantics as a while loop: it will always terminate, and the predicate $b$ is not guaranteed to hold when the loop ends. We assume a constant $K$ that bounds the number of times the body of a retry-until

$$\textsc{Seq}\quad \frac{(s,\sigma) \Rightarrow (\sigma', \tau, \texttt{cont}) \qquad (S', \sigma') \Rightarrow (\sigma'', \tau', cr)}{(s\ S', \sigma) \Rightarrow (\sigma'', \tau \bullet \tau', cr)}$$

$$\textsc{Seq-S-Term}\quad \frac{(s,\sigma) \Rightarrow (\sigma', \tau, \lightning)}{(s\ S', \sigma) \Rightarrow (\sigma', \tau, \lightning)}$$

$$\textsc{Ret}\quad \frac{}{(\texttt{return}, \sigma) \Rightarrow (\sigma, \langle\rangle, \lightning)}$$

$$\textsc{Emp}\quad \frac{}{(\epsilon, \sigma) \Rightarrow (\sigma, \langle\rangle, \texttt{cont})}$$

$$\textsc{Ite-}\top\quad \frac{\sigma \models b \qquad (S_\top, \sigma) \Rightarrow (\sigma', \tau, cr)}{(\texttt{if}\ b\ \{S_\top\}\ \texttt{else}\ \{S_\bot\}, \sigma) \Rightarrow (\sigma', \tau, cr)}$$

$$\textsc{Ite-}\bot\quad \frac{\sigma \not\models b \qquad (S_\bot, \sigma) \Rightarrow (\sigma'', \tau', cr)}{(\texttt{if}\ b\ \{S_\top\}\ \texttt{else}\ \{S_\bot\}, \sigma) \Rightarrow (\sigma'', \tau', cr)}$$

$$\textsc{Retry-Until-Continue}\quad \frac{(S,\sigma) \Rightarrow (\sigma', \tau', cr) \qquad \sigma' \not\models b \wedge \#_\iota < K \qquad \texttt{retry}\ \{S\}\ \texttt{until}\ \{b\}, \sigma'[\#_\iota \to \#_\iota + 1]) \Rightarrow (\sigma'', \tau'', cr)}{({}^{\#_\iota}\texttt{retry}\ \{S\}\ \texttt{until}\ \{b\}, \sigma) \Rightarrow (\sigma'', \tau' \bullet \tau'', cr)}$$

$$\textsc{Retry-Until-Stop}\quad \frac{(S,\sigma) \Rightarrow (\sigma', \tau', cr) \qquad \sigma' \models b \vee \#_\iota \geq K}{({}^{\#_\iota}\texttt{retry}\ \{S\}\ \texttt{until}\ \{b\}, \sigma) \Rightarrow (\sigma'[\#_\iota \to 0], \tau', cr)}$$

$$\textsc{Retry-S-Term}\quad \frac{(S,\sigma) \Rightarrow (\sigma', \tau', \lightning)}{({}^{\#_\iota}\texttt{retry}\ \{S\}\ \texttt{until}\ \{b\}, \sigma) \Rightarrow (\sigma', \tau', \lightning)}$$

$$\textsc{For-Continue}\quad \frac{(S, \sigma[x \to L[\#_\iota]]) \Rightarrow (\sigma', \tau', cr) \qquad \sigma' \models \#_\iota < |L| \qquad ({}^{\#_\iota}\texttt{for}\ x \in L\ \{S\}, \sigma'[\#_\iota \to \#_\iota + 1]) \Rightarrow (\sigma'', \tau'', cr)}{({}^{\#_\iota}\texttt{for}\ x \in L\ \{S\}, \sigma) \Rightarrow (\sigma'', \tau' \bullet \tau'', cr)}$$

$$\textsc{For-Stop}\quad \frac{(S,\sigma) \Rightarrow (\sigma', \tau', cr) \qquad \sigma' \#_\iota \geq |L|}{({}^{\#_\iota}\texttt{for}\ x \in L\ \{S\}, \sigma) \Rightarrow (\sigma'[\#_\iota \to 0], \tau', cr)}$$

$$\textsc{For-S-Term}\quad \frac{(S,\sigma) \Rightarrow (\sigma', \tau', \lightning)}{({}^{\#_\iota}\texttt{for}\ x \in L\ \{S\}, \sigma) \Rightarrow (\sigma', \tau', \lightning)}$$

$$\textsc{Hidden}\quad \frac{\sigma \models f := \mathcal{F} \wedge \exists e \cdot \mathcal{F}(\overline{y}) = e}{(\texttt{let}\ x = f(\overline{y}), \sigma) \Rightarrow (\sigma[x \to e], \langle\rangle, \texttt{cont})}$$

$$\textsc{Visible}\quad \frac{\sigma \models \exists \overline{v} \cdot \overline{y} = \overline{v} \qquad \mathbb{A}(G, \overline{v}) \downarrow e}{(\texttt{let}\ x = \mathbb{A}(\overline{y}), \sigma) \Rightarrow (\sigma[x \to e], (\mathbb{A}(G, \overline{v}), e), \texttt{cont})}$$

Figure 4.5: Big-step semantics

statement can be "retried". For any set of traces, we can select a $K$ higher than the longest trace. This ensures that $K$ is high enough that any trace can be regenerated by the transformed program without timing out. We discuss this more in Section 4.6.

Each retry-until statement is given a unique identifier, $\#_\iota$, and each of those identifiers is assigned 0 in the initial local state. The rule RETRY-UNTIL-CONTINUE states that a retry-until statement with identifier $\#_\iota$ evaluates to state $\sigma''$ and trace $\tau' \bullet \tau''$ when one iteration results in $\sigma'$ and $\tau'$, $b \wedge \#_\iota < K$ holds in state $\sigma'$, and evaluating again the retry-until statement with the state $\sigma'$ where $\#_\iota$ is incremented results in $\sigma''$ and $\tau''$. The rule RETRY-UNTIL-STOP handles the case where $b \wedge \#_\iota < K$ does *not* hold after evaluating the body of the loop. The rule RETRY-S-TERM handles the case where the body of the retry loop returns, and therefore the entire program returns. The rules for for-loops (FOR-CONTINUE, FOR-STOP and FOR-S-TERM) are similar, the main difference being that the variable $x$ is bound at each new iteration and the stop condition depends on the size of the list $L$, not the value of the Boolean $b$.

We differentiate binding on the type of expression they bind. If it is a call to a hidden function $f$ (rule HIDDEN) then the local state is modified by binding $x$ to the value $\mathcal{F}(\overline{y})$ evaluates to in the current state $\sigma$, according to the semantics of the data-transformation domain and assuming $\mathcal{F}$ is $f$'s implementation. The trace is unchanged by the hidden function. If it is a call to a visible function (rule VISIBLE), the arguments of the call are evaluated in $\sigma$, the result of the call is bound to $x$ in the local state, and the call to $\mathbb{A}$ with the input values is recorded in the trace.

The relation $\mathbb{A}(G, \overline{v}) \downarrow e$ means that the call to externally defined function $\mathbb{A}$ with input $\overline{v}$ in global state $G$ returns a response $e$. We implicitly update the global state as a function of each visible call and transfer it through sequences. This means that two programs that start in the same global state and execute the same visible calls receive the same responses to those calls. This formulation allows us to reason about the semantics of the program given existing pairs of input-output examples of calls (the traces) without actually executing any of the calls; we only need to assume the initial global state is the same as in the traces. A limitation of this approach is that time is not considered, so our approach will be unsound in situations where responses implicitly depend on time as opposed to ordering.

Finally, we introduce notation for relating traces with programs and states.

**Definition 4.1: Program Evaluation**

Let $P := \lambda \overline{x}.S$ `where` $\overline{f := \mathcal{F}}$ a program and $\sigma$ a state mapping every variable in $\overline{x}$ to some value and every $f$ into the corresponding $\mathcal{F}$. Then, given a starting global state, there might exist exactly one trace $\tau$ and termination token $cr$ such that $(S, \sigma) \Rightarrow (\sigma', \tau, cr)$. If and only if the trace and termination token exist, we say that $\tau$ is a *trace of* $P$ with input $\sigma$ and write $P(\sigma) = \tau$. While not all syntactically valid programs fully evaluate, all *synthesized* programs evaluate by construction. Given that all programs we discuss are synthesized, we no longer need to consider programs that do not successfully evaluate.

## 4.3 Synthesis Problem

As we illustrated with our motivating example, the synthesis problem solved in this paper consists in finding a program $P$ in the language described in Figure 4.4 → p.40 that reproduces a set of input traces $T_{in}$. In this section, we formalize this intuitive correctness constraint and define our synthesis problem as a combination of the correctness constraint and another constraint on the quality of the program.

### 4.3.1 Correct Solutions

We use the notion of *trace subsumption* to describe one program that can generate at least the same traces as another:

> **Definition 4.2: Subsumption $\sqsupseteq$**
>
> A program $P'$ subsumes a program $P$ ($P' \sqsupseteq P$) if and only if for every state $\sigma$ and trace $\tau$ such that $P(\sigma) = \tau$, there exists a state $\sigma'$ such that $P'(\sigma') = \tau$.

Note that $\sqsupseteq$ is a partial order on programs. If $P \sqsupseteq P'$ and $P' \sqsupseteq P$ then $P$ and $P'$ are *trace equivalent*. In general, we are interested in transformations that preserve subsumption (i.e. $P \rightsquigarrow P'$ only if $P' \sqsupseteq P$, where $\rightsquigarrow$ is a transformation), not just trace equivalence. Formally, our correctness constraint $\Psi$ is

$$\Psi(P, T_{in}) \equiv \forall \tau_i \in T_{in} \cdot \exists \sigma \cdot P(\sigma) = \tau_i. \tag{4.1}$$

The set $T_{in}$ is a set of finite input traces $\tau_1, \tau_2, ..., \tau_t$[3]. There is always a trivial solution to $\Psi$ for a given set of traces $T_{in}$. It can be constructed using a single integer parameter $br$ and $|T_{in}|$ branches, where each branch can be selected with a value for $br$, and the branch makes the API calls contained in the $br$-th trace of $T_{in}$. We show in Section 4.1 → p.36 how this solution is built directly from the traces; it simply replays each of the traces, and the set of possible traces of the program is exactly $T_{in}$.

Another less trivial solution would consist of combining visible function calls when possible but leaving all the inputs of the visible calls as parameters of the program, thus always discarding the output of the visible function call. This is also not an acceptable solution. Although it generalizes to other inputs, the generalization only comes from the program being entirely parameterized.

### 4.3.2 Quality Constraint

Although trivial solutions exist, they usually will not be what a user would expect as output; there is an expectation that the solution is a generalization of the traces. There are also infinitely many correct (solutions to $\Psi$) programs that are very general; consider, for example, a program listing all possible syntactic productions that satisfy the correctness constraint in branches. However, those programs are also not typically what the user expects.

To address this challenge, we assume the existence of a *program cost function* $\chi$ that, given a program $P$ and a set of traces $T_{in}$, returns a positive number. This program cost function reflects what the user expects; a good program is one with a low cost. For example, the program cost function could return the count of branches and the count of parameters of the program, indicating that the user desires a program with low complexity that is likely to generalize well. The goal of the synthesizer is to find a program that minimizes the cost function. Formally, the goal is to solve, given a fixed set of traces $T_{in}$,

$$\min_{\forall P \cdot \Psi(P, T_{in})} \chi(P, T_{in}). \tag{4.2}$$

In this paper, we describe a generic algorithm that is parametric in $\chi$, first by describing our rewrites in Section 4.4 → p.45, and then by describing the search approach in Section 4.5 → p.50.

[3] We always assume that $|T_{in}| > 1$.

## 4.4 Rewriting Programs

Our synthesis algorithm applies a succession of rewrite rules to transform an initial trivial program into a more general and user-friendly one. Each of these rewrite rules provably maintains the program's correctness constraint, $\Psi$, so that all intermediate programs can generate all input traces in $T_{in}$. We split our rules into two categories, *synthesis* rules and *refinement* rules, depending on how they maintain correctness.

> **Lemma:**
> Subsumption preserves correctness: $\Psi(P, T_{in}) \bigwedge P' \sqsupseteq P \implies \Psi(P', T_{in})$.

> **Proof:**
> If $P$ can generate all traces in $T_{in}$, and $P'$ can generate all traces that $P$ can generate, then $P'$ can generate all traces in $T_{in}$.

*Refinement rewrite rules* preserve subsumption: for all $P$ and $P'$, if a refinement rule rewrites $P$ into $P'$, $P \rightsquigarrow P'$, then $P' \sqsupseteq P$. By Lemma 4.1, these rules preserve the correctness when applied to a correct program. The following is an example of the application of a refinement rule that extracts an identical instruction $\mathcal{R}$ from both branches of an if-then-else statement. This rewrite does not change the semantics of the program but improves its readability by reducing its number of instructions.

$$\begin{array}{l} \lambda\ \overline{x}. \\ \mathcal{U}\ \texttt{if}\ \mathcal{C}\ \{\ \mathcal{R}\ \mathcal{S}\ \}\ \texttt{else}\ \{\ \mathcal{R}\ \mathcal{T}\ \}\ \mathcal{V} \end{array} \quad \rightsquigarrow \quad \begin{array}{l} \lambda\ \overline{x}. \\ \mathcal{U}\ \mathcal{R}\ \texttt{if}\ \mathcal{C}\ \{\ \mathcal{S}\ \}\ \texttt{else}\ \{\ \mathcal{T}\ \}\ \mathcal{V} \end{array}$$

*Synthesis rewrite rules* all follow the same pattern: replace an expression $e$ with the output of a call to a to-be-synthesized pure function $\phi$. $\phi$ is not visible in the traces, so we refer to it as a *hidden* function. All the bound variables available at the location are used as arguments to $\phi$, except when the rule is trying to eliminate a parameter. The correctness of a synthesis rewrite rule is conditioned by the existence of a solution for the hidden function calls they introduce. Formally, we denote by $\Lambda\overline{\phi} \cdot P$ a program $P$ parametric on a set of hidden functions $\overline{\phi}$. For a given set of implementations $\overline{f}$, $(\Lambda\overline{\phi} \cdot P)(\overline{f})$ is a valid program in our DSL. A single synthesis rule $\rightsquigarrow$ rewrites $P$ to a program $P' := \Lambda\phi \cdot P_s$ parametric on some hidden function $\phi$. The rewrite rule $P \rightsquigarrow P'(f)$ is correct for some function $f$ only if $P'(f) \sqsupseteq P_{in}$ where $P_{in}$ is the initial program. By Lemma 4.1 → p.45, if $\Psi(P_{in}, T_{in})$ and $P'(f) \sqsupseteq P_{in}$, then $\Psi(P', T_{in})$. Note that we can chain multiple synthesis rules together and check for correctness only later, i.e. rewrite $P \rightsquigarrow \Lambda\phi \cdot P_s \rightsquigarrow \Lambda\phi, \phi' \cdot P'_s$ and then find $f$ and $f'$ later to instantiate $\phi$ and $\phi'$. We explain how to find the implementation of hidden functions $f$ in Section 4.4.2 → p.48. For a list of Syren's rewrite rules, the reader can refer to [47].

> **Example 4.2:**
> We illustrate below how we apply a sequence of rewrite rules to generalize programs and produce an acceptable solution. Suppose that we have constants `c1,c2,c3`, visible functions `A,B` and some initial program $P_{in}$ as shown below:

$P_{in}$ :

```
λ br.
if br = 1 {
  let x1 = A(c1)
  let y = B(c2)
} else {
  let x2 = A(c3)
}
```

$\rightsquigarrow$

$P_1$ :

```
λ br.
let a = (br = 1) ? c1 : c3
if br = 1 {
  let x1 = A(a)
  let y = B(c2)
} else { let x2 = A(a) }
```

$\rightsquigarrow$

$P_2$ :

```
λ br.
let a = (br = 1) ? c1 : c3
let x = A(a)
if br
= 1 { let y = B(c2) }
```

$\rightsquigarrow$

$P_3$ :

```
λ br, d.
let x = A(d)
if br = 1 {
  let y = B(c2)
}
```

$\rightsquigarrow$

$P_4$ :

```
Λ ϕ λ br, d.
let x = A(d)
let c = ϕ(d, x)
if c { let y = B(c2) }
```

$\rightsquigarrow$

$P_5$ :

```
Λ ϕ λ br, d.
let x = A(d)
let c = ϕ(d, x)
if c { let y = B(c2) }
```

We rewrite $P_{in}$ using a refinement rule that introduces a new variable `a`, which is bound to the constants `c1` or `c3` in the conditional, and then used as argument to the calls to `A`. $P_1$ is the resulting program. Then, we apply to $P_1$ the refinement rule shown in Section 4.4, which factors the calls to `A` out of the conditional, resulting in $P_2$. The third rewrite eliminates the expression `let a = if br= 1 {c1} {c3}` which depends on *br* and introduces a parameter `d` that takes its value. This rewrite provably maintains correctness and produces a program, $P_3$, generalized to any input `d`. A fourth rewrite introduces a function parameter $\phi$ (to be synthesized) to eliminate *br* from the conditional, resulting in $P_4$. The final rewrite eliminates the unused parameter *br*.

### 4.4.1 Trace Valuation

Rewrites maintain correctness by ensuring that, given an implementation for the hidden function introduced, the program can still generate the initial set of traces. While refinement rewrite rules are correct for all inputs of the program, synthesis rewrite rules require more attention. To keep track of this correctness constraint, we maintain an augmented local state $\sigma$, the trace valuation of the program, which relates variables in the program with a specific trace and a concrete value (and an iteration number for variables in loops). The trace valuation stores the relationships necessary for the rewritten program to reproduce each trace $\tau \in T_{in}$, and each rewrite rule application modifies that state to maintain the invariant. This ensures that the value of all expressions of the program for a certain trace is always known, either because the variable's value is known, or the expression's value can be computed from those known values. Given an expression $e$, trace valuations $\sigma$ and trace $\tau$ we denote the value of $e$ in trace $\tau$ and state $\sigma$ by $[\![e]\!]_{\sigma,\tau}$.

The initial program has only one variable *br*, and initially $[\![br]\!]_{\sigma,\tau_i} = i$ for each trace $\tau_i \in T_{in}$. Then, each rewrite rule $\rightsquigarrow$ in our system is accompanied with a trace valuation transformation $t$, which we denote by $\rightsquigarrow_t$. We introduce a new function $I$, which extracts the program parameters from a state, and overload $\rightsquigarrow$ to apply to sequences of instructions as well, instead of entire programs only. Each $t$ and associated rewrite $\rightsquigarrow_t$ is correct when for each input trace, when a rewrite and corresponding trace valuation transformation are applied, if the program runs with the inputs of the updated state then it produces the same input trace:

$$\tau_i \in T_{in} \wedge t(\sigma_0') = \sigma_1' \wedge (S, \sigma_0) \Rightarrow (\sigma_0', \tau_i, cr) \wedge S \rightsquigarrow_t S' \implies (S', I(\sigma_1')) \Rightarrow (\sigma_1', \tau_i, cr)$$

The function $t$ will encapsulate the parameter updates of the rewrite and any functions introduced. This rule requires the correctness of $S$ (in the third conjunct of the hypothesis), and the conclusion directly implies the correctness of $S'$, where $I(\psi_1')$ witnesses the existential needed by the correctness statement.

Synthesis rules replace an expression or number of expressions with a hidden function.

**Example 4.3:**

The refinement rule of Example 4.2 → p.45 can specified with its transformation $t_3$:

$P_2$:

```
λ br.
let a = if br= 1 { c1 } else { c3 }
let x = A(a)
if br= 1 { let y = B(c2) }
```

$\rightsquigarrow_{t_3}$

$P_3$:

```
λ br, d.
let x = A(d)
if br= 1 { let y = B(c2) }
```

where $t_3(\sigma) = \sigma[(d, \tau) \mapsto [\![\texttt{if br= 1 c1 else c3}]\!]_{\sigma,\tau}]$

That is, the trace valuation transformation $t_3$ corresponding to this rewrite assigns the resulting value of evaluating the eliminated expression `if br= 1 c1 else c3` to the new parameter `d`. Concretely, if the program above is synthesized from the two traces:

$$\tau_1 = (\texttt{A(c1)}, o_1) :: (\texttt{B(c2)}, o_2) \quad \text{and} \quad \tau_2 = (\texttt{A(c3)}, o_3)$$

Given that $[\![br]\!]_{\sigma,\tau_1} = 1$, we have $[\![\texttt{if br= 1 c1 else c3}]\!]_{\sigma,\tau_1} = \texttt{c1}$, and therefore $[\![d]\!]_{t_3(\sigma),\tau_1} = \texttt{c1}$. For the second trace, we would have $[\![d]\!]_{t_3(\sigma),\tau_1} = \texttt{c3}$.

In a following step in Example 4.2 → p.45, we apply the following synthesis rule to the program:

$P_3$:

```
λ br, d.
let x = A(d)
if br= 1 { let y = B(c2) }
```

$\rightsquigarrow_{t_4}$

$P_4$:

```
Λ ϕ λ br, d.
let x = A(d)
let c = ϕ(d, x)
if c { let y = B(c2) }
```

where $t_4(\sigma) = \sigma[(c, \tau) \mapsto [\![br = 1]\!]_{\sigma,\tau}]$

The trace valuation for program $P_4$ will map `d` to the correct boolean value in each trace, that is, $[\![c]\!]_{\sigma,\tau_1} = \texttt{true}$ and $[\![c]\!]_{\sigma,\tau_2} = \texttt{false}$. Additionally, the values for the inputs of $c$ and $x$ will also be known from the state, for example for trace 1 $[\![d]\!]_{\sigma,\tau_1} = \texttt{c1}$ and $[\![x]\!]_{\sigma,\tau_1} = o_1$ (see trace $\tau_1$).

**Example 4.4:**

The rewrite rules that introduce retry loops are synthesis rules because the condition on which to stop the loop needs to be synthesized. Syntactically, the rewrite identifies a sequence of statements, possibly with conditionals, and rolls them into a loop. The following is an example of a loop introduction rewrite:

```
λ br, ȳ.                          Λ ϕ λ br, ȳ.
let a = A(c1)                     let a = A(c1)
let b1 = B(c2)          ⇝t        retry {
let b2 = B(c2)                     let b = B(c2)
if br=1 { let b3 = B(c2) }         let s = ϕ(b, a, ȳ)
                                  } until s
```

where $t(\sigma) = \sigma[(\texttt{(b,b,b)}, \tau) \mapsto [\![\texttt{(b1,b2,b3)}]\!]_{\sigma,\tau}] \cup [(\texttt{(s,s,s)}, \tau) \mapsto \texttt{(false,}[\![\textit{br}\texttt{ != 1}]\!]_{\sigma,\tau}\texttt{,true)}]$

In the rewritten program syntax, a new variable `s` is bound to the result of the hidden function $\phi$ and used as a stopping condition for the retry loop. The key in ensuring this is a correct rewrite is in the valuation transformation $t$. The new trace valuation maps *iterations* of `b` to the values of each statement that has been captured in the loop, represented by the vector `(b1,b2,b3)`. When evaluating the program for a trace, the variable `b3` will not be defined for the traces where $br \neq 1$, in which case the value is null. The valuation of condition `s` is also a vector `(s,s,s)` that is computed by assigning the truth value of whether the statement in the trace should be the last one; at the end of the second iteration, `s` is `true` for the trace where *br*`!=1`.

### 4.4.2 Synthesizing Hidden Functions

The correctness of the result of applying a synthesis rewrite rule $P \rightsquigarrow P'(f)$ depends on satisfying a set of constraints imposed on $f$ by the condition $\exists f.P'(f) \sqsupseteq P_{in}$. As we explained in the previous section, all rewrite rules update an extended state that keeps track of the valuations of the variables in a correct program. Given the value in the extended state, the synthesis of $f$ is reducible to a standard programming-by-example (PBE) synthesis problem, deducible from $\sigma$ only. Those constraints are solved with an off-the-shelf synthesizer to produce either an implementation for $f$ or an unsatisfiability result. In the latter case, the synthesis rule cannot be applied while maintaining correctness.

**Generating Input/Output Examples** Synthesis rewrite rules replace an expression $e$ in the program with a function call $\phi(\overline{x})$ whose result is bound to a new variable `y`. Before the rewrite, for each trace $\tau$ we had some value for $e$, i.e., $[\![e]\!]_{\sigma,\tau} = v_\tau$. To maintain trace subsumption, the transformation $t$ ensures $[\![\texttt{y}]\!]_{t(\sigma),\tau} = v_\tau$ by mapping the new variable `y` to the appropriate value. A correct implementation for $\phi$ must satisfy for each trace $\tau$ the input-output constraint $\phi([\![\overline{x}]\!]_{\sigma,\tau}) = v_\tau$.

$$
\begin{array}{lcll}
\mathcal{B} & ::= & \mathcal{J} == \mathcal{V} & \textit{string or integer equality} \\
 & | & \textit{empty}(\mathcal{J}) & \textit{emptiness check} \\
 & | & !\mathcal{B} & \textit{negation} \\
\mathcal{J} & ::= & \$ & \textit{input} \\
 & | & \mathcal{J}.\mathcal{K} & \textit{select child by name} \\
 & | & \mathcal{J}..\mathcal{K} & \textit{select descendants by name} \\
 & | & \mathcal{J}[\mathcal{I}] & \textit{select by index} \\
 & | & \mathcal{J}[\mathcal{I}:\mathcal{I}] & \textit{slice by index} \\
 & | & \textit{length}(\mathcal{J}) & \textit{length} \\
 & | & \mathcal{V} + \mathcal{J} & \textit{numerical addition} \\
 & | & \mathcal{V} \bullet \mathcal{J} & \textit{string concatenation} \\
\mathcal{K} & ::= & k \in \textit{keys} & \\
\mathcal{V} & ::= & v \in \textit{values} & \\
\mathcal{I} & ::= & i \in \textit{indices} &
\end{array}
$$

Figure 4.6: Hidden functions synthesis DSL.

**Example 4.5:**

Recall Example 4.3. Program $P_4$ is parametric on $\phi$, which appears in `let c = ϕ(d,x)`, and is correct for a specific implementation of $\phi$ iff for all traces $\tau$, $\phi([\![d]\!]_{\sigma,\tau}, [\![x]\!]_{\sigma,\tau}) = [\![c]\!]_{\sigma,\tau}$. Since we have two traces we have the constraints $\phi(\texttt{c1}, o_1) = \texttt{true}$ and $\phi(\texttt{c3}, o_3) = \texttt{false}$.

**Synthesizing Solutions** The input-output pairs for each hidden function are used to synthesize the expression for that hidden function. To achieve this, we encode the problems of our synthesis domain into an existing example-based program synthesizer. The hidden functions synthesis can be done in any domain, as long as it is supported by the example-based synthesizer. In this section, we illustrate using the domain of our running example: cloud automation scripts. The visible functions are typically APIs that accept parameters and return responses in JSON format [23]. Thus, our domain targets JSON data manipulation scripts, including small predicates for generating conditions. This domain covers the majority of use cases of hidden functions between visible function calls in the automation scripts we observed. JSON is a lightweight, language-agnostic data interchange format widely used in web applications and APIs. The main (recursive) datatypes are lists and dictionaries, which map unique string keys to other JSON objects. The base datatypes are booleans, strings and numbers. Our synthesis domain is summarized in Figure 4.6, which presents a grammar that includes basic comparison between objects and values, and JSONPath [54] operations. The non-terminal $\mathcal{B}$ in the grammar symbolizes the boolean expressions we consider in our DSL, and $\mathcal{J}$ the JSONPath expressions. Those operations allow the selection of specific indices, members, or descendants of JSON data structures. For example, the path `$.element[0]` selects the `element` field of the object, and then the first element in that list.

To the best of our knowledge there is no synthesizer that targets this domain, despite the ubiquity of JSON to represent data in applications. Solving it requires encoding our problem into

a domain supported by a general purpose synthesizer that allows specifications using input-output examples. In Syren, we encode the JSONPath synthesis problem grammar into Rosette [127], a solver-aided programming language with synthesis constructs. Rosette does not support symbolic strings, thus in our encoding, all strings are constant values extracted from the input and output examples. The string values are used for *keys* and *values* in the grammar in Figure 4.6, and are the result of enumerating all keys in the dictionaries in input-output constraints, and all values, respectively. In some problems, the size of this set of constants becomes a bottleneck because the objects returned by API calls contain hundreds of keys and values. We parallelize the search for a solution by producing sub-grammars for the problem, using different sets of keys and values, and different grammar sizes [22, 50]. Rosette was able to synthesize most Boolean expressions in our benchmarks in the grammar including a subset of JSONPath and string operations, shown in Figure 4.6.

We considered an alternative approach to using Rosette: encoding the synthesis problem into SMT theories, and using a SyGuS solver supporting those theories. The SyGuS language [108] allows users to specify synthesis problems with input/output pairs as specifications. We encoded JSON data structures and JSONPath operations using a combination of list and user-defined datatypes for dictionaries and lists, and string and integer theories for the base types. We tested this encoding on CVC5 [13] alongside Rosette [127] on our benchmarks, and found that Rosette consistently outperforms CVC5. CVC5 was unable to solve the problems in our JSONPath benchmarks in reasonable time. We also experimented with PBE problems in the domain of arithmetic operations, and CVC5 and the SyGuS encoding outperformed the grammar defined in Rosette. We conclude that the performance of the PBE solver to which Syren offloads the hidden function synthesis depends very highly on the domain. Syren is agnostic to it, and we provide support for using either CVC5 or Rosette, as well as for parallel portfolio solving.

## 4.5 Rewrite Strategies

The synthesis algorithm is a rewrite process that starts with an initial program $P_{in}$ that trivially satisfies the correctness criterion $\Psi$, but is not likely to minimize the cost function $\chi$. The goal is to transform the initial program by applying refinement and synthesis rules until a program minimizing $\chi$ is found. Naturally, a naive solution would be to enumerate all possible ways of rewriting $P_{in}$. However, depending on the program and the set of rewrites available, there may not be a finite set of programs. We consider different strategies to explore the search space of all rewrites efficiently, with the goal of optimizing for $\chi$.

**Initial Program** We start with a trivially correct program $P_{in}$ that provably generates all input traces $T_{in}$. This program is constructed by introducing a single parameter $br$ and a program body that consists of $|T_{in}|$ branches. Each branch is guarded by a condition $br == i$, with $0 \leq i < |T_{in}|$. The statements in the then-branch are the API calls of trace $\tau_i$, written as API call bindings to fresh local variables. The else-branch contains the other branches. Figure 4.7 → p.51 shows the constructed program.

In Section 4.4 → p.45, we distinguish two types of rewrite rules: *refinement rules* $\mathcal{R}^\star$, which simply rewrite the program maintaining trace subsumption, and *synthesis rules* $\mathcal{R}^?$ which introduce a data transform synthesis constraint and are correct by construction of the solution of these constraints. Intuitively, refinement rules use less memory and computation, whereas synthesis

```
λ br
if br == 1 {
   let y_1 = A_1^1(x_1^1) let y_2 = A_2^1(x_2^1) ... let y_N1 = A_N1^1(x_N1^1) (** Replays trace τ_1 *)
} else if br== 2 {
   let y_1 = A_1^2(x_1^2) let y_2 = A_2^2(x_2^2) ... let y_N2 = A_N2^2(x_N2^2) (** Replays trace τ_2 *)
} else if br== 3 {
   let y_1= A_1^3(x_1^3) let y_2 = A_2^3(x_2^3) ... let y_N3 = A_N3^3(x_N3^3) (** Replays trace τ_3 *)
} else ...
```

Figure 4.7: The initial program $P_{in}$ takes a single integer parameter `br`, and has $|T_{in}|$ branches, where each branch $i$ simply replays the API calls in trace $\tau_i \in T_{in}$. $A_i^q$ is the $i$-th function call in trace $\#q$, and $x_i^q$ the corresponding input.

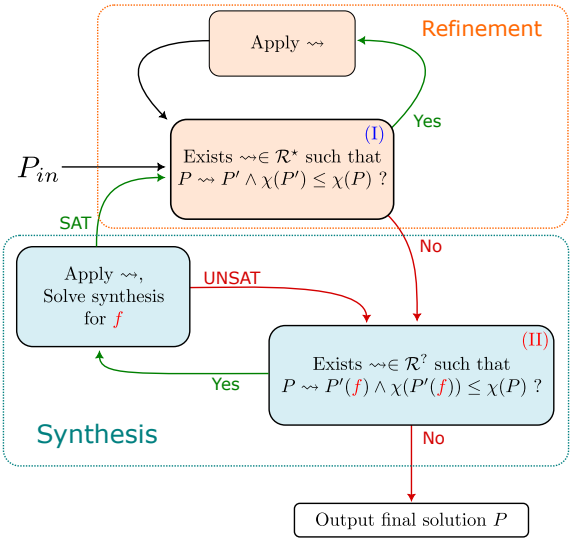



Figure 4.8: Cost-directed alternating rewrite rule application.

rules should be applied more carefully. For scalability, one should use refinement rules as much as possible until applying synthesis rules is necessary.

**Alternating Refinement and Synthesis** Figure 4.8 → p.51 illustrates our main algorithm, which alternates between refinement rule and synthesis rule applications until no rewrite rule is applicable. We start with the initial program $P_{in}$ and look for refinement rules (in $\mathcal{R}^\star$) to apply in a way that reduces the cost of the program (step (I)). The rule that yields the lowest cost is selected first. If such a rule can be found, we apply it to the current program. We repeat these two steps until no refinement rule can be found. In that case, the algorithm moves inside the bottom loop. It searches for a synthesis rule in $\mathcal{R}^?$ that reduces the most the cost of the program (step (II)). If no rule can be found, the synthesis terminates with the current program. Otherwise, the algorithm attempts to apply the rewrite rule $\exists f \cdot P \rightsquigarrow P'(f)$ and solve for $f$ using the synthesis process described in Section 4.4.2 → p.48. There are two possible answers: either a solution is found (SAT) or the synthesizer returned UNSAT. In the first case, the algorithm returns to the upper loop and repeats the entire process. In the other case, the algorithm backtracks on the synthesis

rewrite and attempts to find another synthesis rule to apply. When no synthesis rules apply, the algorithm returns the final program.

This algorithm applies synthesis rules parsimoniously compared to refinement rules. A synthesis problem is solved only when no refinement rules can further lower the program cost, and as soon as a synthesis rule is applied, the algorithm attempts to use more refinement rules.

### 4.5.1 Baselines

We give a brief overview of baseline algorithms we implemented as a basis for testing our hypothesis, starting with the observation that motivates them.

**Refine-then-Synthesize** Experimentally, we observe that, in many cases, it may be sufficient to apply rules in *only three phases*. First, apply all possible refinement rules to simplify the trivial program $P_{in}$. Then, we apply every possible synthesis rule that reduces the program's cost. Finally, a final round of refinement rewriting is necessary to clean up the program with the new data transformations. The intuition is that refinement rules operate mostly on the control flow of the program, while synthesis rules operate on the data flow, and interaction between the two is minimal.

With that insight in mind, the *refine-then-synthesize* algorithm (denoted RTS) first applies refinement rules, reducing the cost of the program until no refinement rule is applicable, and then synthesis rules until no rule can be found, and finally another round of refinement rules. In other words, it is a modification of the algorithm in Figure 4.8 → p.51 where the SAT arrow instead points to step (II) and updates the program, and the No arrow returns to refinement for one round.

**$k$-Bounded Search** One problem of the two previous algorithms is that they *may get stuck on a local minimum of the cost function*. A completely different approach that does not have this problem is a bounded exhaustive search starting from $P_{in}$. In the $k$-bounded search algorithm (denoted by $k$-search), rewrite rules from both the refinement set $\mathcal{R}^\star$ and synthesis set $\mathcal{R}^?$ are applied, independently of their effect on program cost. Rules are applied again to the resulting programs until all programs resulting from applying $k$ rules (where $k$ is a constant) are obtained.

Once all possible rewrites are enumerated, the algorithm ranks all rewrites by increasing cost and attempts to find the program with the lowest cost whose underlying synthesis constraints are satisfiable. Note that in this version of the algorithm, we do not solve the synthesis problem when a synthesis rule is applied. The enumeration is done without a single call to the synthesis solver, which is used only for the programs with low scores.

## 4.6 Evaluation

We implemented our synthesis approach in a tool, Syren, and evaluated the different algorithms and cost functions against a set of benchmarks, showing a promising approach for synthesizing real-world API composing functions. Since no existing tool can solve the problem out of the box, we compare against the baseline algorithms introduced in Section 4.5 → p.50: *refine-then-synthesize* (RTS) and $k$-bounded search ($k$-search). Although comparison against a monolithic syntax-guided synthesis approach may be possible (e.g., encoding the problem in Rosette), the

limitations in the scalability of Rosette to solve even only the subproblems indicate that it would not scale to the entire problem.

### 4.6.1 Implementation

All experiments were run on a 2022 Macbook Pro with an M1Pro processor (10 physical cores) and 32GB memory. Syren is implemented in OCaml and Python 3.12 and uses the Rosette [127] solver-aided language (version 4.1 running on Racket version 8.11) with its default solver Z3 (version 4.12) [103] to synthesize data transformations. The implementation uses a synthesis constraint cache to avoid repeated calls to the solver with the same constraints. This is especially useful since the algorithm will attempt many synthesis rewrites that will not have a solution.

### 4.6.2 Benchmarks

We test Syren on a set of 54 benchmarks that implement various tasks that require branching and looping (in the form of retries) using various APIs. The full description of each task is available in the appendix of the original paper [47]. We grouped our benchmarks into four different categories to indicate their origin. The first category is a set of *custom benchmarks* that we wrote to perform some tasks on cloud infrastructure, some shell scripts, and SVG manipulation scripts. We then collected tasks from the Blink automation library [16], where various APIs are interfaced. A similar set of tasks comes from AWS Systems Manager Automation Runbooks [9]. Our final category consists of tasks adapted from previous literature; we adapt the nested loop-free benchmarks from ApiPhany [63] that use Stripe and Slack APIs[4] In general, in the benchmarks used to evaluate Syren, there is a clear separation between the visible function calls (cloud API calls, system calls or library calls) and the local operations (which can be encoded into some solver's theory). When constructing the set of traces, two important parameters have to be considered: whether the different sequences of visible calls exemplify the desired program's control flow paths, and whether the various input values to the calls are sufficient to infer the hidden function's implementation in a given domain (e.g. JSON transformations requires fewer examples than arithmetic).

We wrote programs for each benchmark and collected the inputs for the synthesizer by simulating the traces those programs would produce. We ran each program for enough different inputs that the produced traces exercise all program paths; we manually inspected the synthesized program and added more traces when it did not exemplify all the behaviors of the target benchmark program. We collected between 2 and 10 traces (median 4) for each benchmark. This is not the smallest number of traces necessary to describe the task, but a reasonable amount that the user could provide.

Our benchmarks include synthesis tasks of varying complexity so we can gauge how well Syren scales. Although we cannot predict how complex a given task is to synthesize, we can estimate it by the complexity of the smallest program that performs that task. We do so by considering the number of conditionals, loops, and hidden functions in the program.

[4]We collect benchmarks from ApiPhany [63] by simulating traces from their solutions that do not have list comprehensions. However, we cannot make a direct comparison since our specifications are traces and theirs are the types of the desired inputs and outputs.

### 4.6.3 Cost Functions

We ran experiments with two different cost functions to evaluate the flexibility of our approach with respect to different user-defined notions of "best program." We follow the general idea that good programs are simple programs that generalize well. The rewrite rules, especially the refinement rules, are generally geared towards syntactic simplification of the program. The cost functions match this high-level goal and generally assign a lower cost to simpler programs; the exact meaning of simplicity depends on the user.

**Syntactic Complexity** The first method we use, denoted by $\chi_{\text{syn}}$, is a straightforward cost function that describes the syntactic complexity of the program. This intuitively corresponds to a user who desires a program that is syntactically as simple as possible. This function computes a weighted sum of the number of conditionals, loops, and parameters in the program and a penalty for using the dummy branching variable *br* introduced in the initial program. We use a simplified version of the function in the running example of Section 4.1 → p.36. This function can easily be customized, for example, by modifying the weights of each characteristic in the summation. Typically, we prioritize fewer statements, then fewer parameters, and set the weights accordingly. The penalty for the *br* variable ensures that the algorithm will prioritize eliminating this parameter over all else.

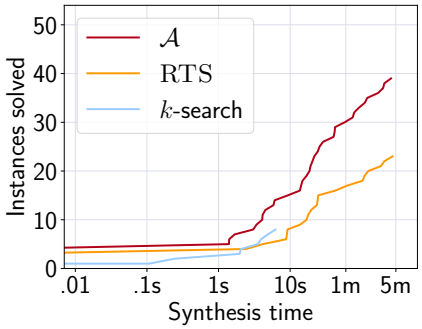
**Reuse Across Traces** Our second cost function, denoted by $\chi_{\text{T}}$, measures how many times API call statements are reused with respect to the input set of traces. For each trace in $T_{in}$, we count how many times each API call statement in the program must be called to produce the trace. The cost is the total number of API calls in the traces plus the number of statements minus the sum of the counts for all statements and all traces. Intuitively, a program with a lower cost means that the API call statements are reused more often; for example, unrolling a loop would increase the cost. We also add a penalty for using the *br* variable. This is another measure of the program's simplicity. This cost function is coarser in that it assigns the same cost to many different programs.

### 4.6.4 Results

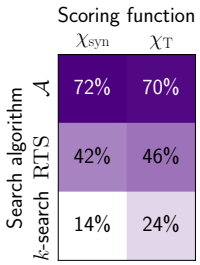
The goal is to synthesize a program that is equivalent to the one used to collect traces from; when we report synthesis success, this means the synthesized program is syntactically equivalent to the desired program (modulo variable renaming). Since Syren may terminate with a correct solution that is not optimal (i.e. has a larger cost than our desired program), we also report when the tool terminated but did not solve the benchmark.

Figure 4.9 → p.55 plots the synthesis time required for each benchmark and combination of cost function ($\chi_{\text{syn}}$ or $\chi_{\text{T}}$) and algorithms ($\mathcal{A}$ for our algorithm, RTS and $k$-bounded for the baselines). Each experiment runs 10 times with a timeout of 10 minutes. We use a fixed $k = 6$.

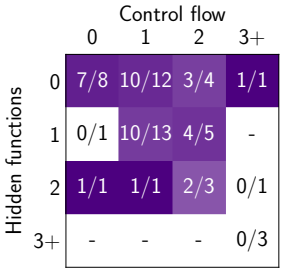
Our proposed algorithm $\mathcal{A}$ finds the most solutions across benchmarks and cost functions (39 or 72% optimally synthesized for syntactic cost, and 38 or 70% for trace reuse cost out of 54 benchmarks). The simplified algorithm, RTS, produces fewer optimal solutions, 23 (42%) for $\chi_{\text{syn}}$ and 25 (46%) for $\chi_T$); in many cases the solution produced is not optimal because the algorithm did not attempt enough rewrites. This is especially the case for the more complex benchmarks containing loops. We found that the choice of scoring function has little impact on solving time.




(a) Number of benchmarks solved (optimal solution synthesized) for each algorithm of Syren with score function $\chi_{\text{syn}}$ against synthesis time.




(b) Percentage of benchmarks for which Syren synthesized the optimal solution by search algorithm and scoring function.




(c) Count of optimally synthesized / total benchmarks, grouped by complexity measures (# hidden functions, # control flow statements) of the program.

Figure 4.9: Comparison of synthesis times and quality of synthesized programs using different search algorithms and cost functions. The background color in the heat map in 4.9b reflects the same data as the labels. In 4.9c, we show Syren's ability to scale to complex benchmarks. The darker the color the the better Syren preformed for benchmarks of that complexity.

However, $\chi_T$ is more coarse in the sense that more programs may have the same score, and when manually inspecting solutions, we found that they are usually farther from the ideal solution. $\chi_{\text{syn}}$ is better at characterizing the ideal solution. The solutions obtained required from 4 to 36 rewrites. The appendix of the original paper [47] lists detailed synthesis times for all benchmarks.

The bounded search performs poorly across the benchmarks (13 optimally solved for $\chi_T$), either yielding a poor solution (because of a small $k$) or timing out. The size of the search space for the bounded search is a combination of both the number of rewrite rules and the complexity of the initial program. In general, we observed that scaling $k$ to the same number used to find a solution using the other algorithms would produce an intractable search space. Interestingly, we observe many timeouts for the $\chi_T$ cost function; there are many programs with the same cost, but most of them have unrealizable synthesis subproblems.

Even when the best synthesizable solution does not require a data transformation, the rewrite process might still attempt to find some. Synthesis time is dominated by the number of synthesis rewrites attempted rather than the number that ends up being used, so time often does not correspond to the total number of rewrites. For example, synthesizing a solution for the CreateTable benchmark takes 78s, despite not requiring any data transformation. However, since there are two API calls with many arguments, the algorithm must ensure, by attempting to synthesize data transformations, that none of the arguments of the second API call can be computed from the results of the first API call.

**Comparison against Large Language Models** Large Language Models (LLMs) have become a tool for generating code from natural language specifications. One can speculate on whether specifications could also be given as a list of traces, as in our problem. To compare Syren against LLMs, we queried Claude 3.5 Sonnet [6] with a prompt explaining the problem to solve, followed by the same traces used by Syren, in JSON format (the complete prompt is available in the appendix of the original paper [47]). The output is expected to be a Python

script that should be correct (in the sense of Equation (4.1) → p.44) and semantically close to the ideal program. With that success criterion in mind, the LLM synthesizes a correct and optimal solution for 29 benchmarks (53% of total).

Note that the evaluation of correctness and equivalence is manual work on our part. This highlights a crucial difference between the two methods. When Syren succeeds and generates a program, the user trusts it is correct, in the sense that it will reproduce the traces. If Syren cannot generate a good program, it will fail or otherwise generate a program that is not general enough, but the output is always safe to use. Specifically, it will never make API calls different from those in the traces, or calls with new inputs. On the other hand, LLMs can fail to synthesize a correct program silently; their output can be subtly incorrect, and verifying this output would require effort (that Syren avoids by generating a program correct by construction). For example, in the task described in §4.1, Claude synthesizes the condition `status == "running"` instead of `status != "stopped"`, which does not satisfy traces where the instance status is `"stopping"`.

**Limitations** Our evaluation compares Syren's default search strategy to our own baselines. To the best of our knowledge, there is no other tool that currently solves this problem, in which the specification is a set of *partial* traces.

Our diverse range of benchmarks includes a few examples where Syren times out or does not return a good solution. Some of those are due to the limited expressive power of the syntax-guided synthesis approach used to create the data transformations. For example, the benchmark ReportLongRunningInstancesToSlack requires reasoning about dates and time intervals, and our solver cannot currently synthesize a solution. The objective program cannot be represented in our synthesis DSL, so we cannot indicate the number of ifs or loops such a program would have. Similarly, the CreateImage benchmark requires reasoning about string operations. Improving the underlying PBE solver to handle more and more complex date, integer, and string operations is an interesting and promising direction of future research. Such improvements would result in an improvement of Syren's performance without further modification. The rewrite system also has limitations when considering benchmarks with complex control flow. For example, our tool fails to synthesize the automation AWSSupport-CopyEC2Instance, which has a loop, 5 conditionals, and 16 data transformations to synthesize. Our approach also cannot synthesize any control flow that happens before the first API call in the automation to be synthesized. For example, the benchmark AWS-ConfigureS3BucketVersioning contains empty traces that correspond to situations where the user ran the script, but it terminated before performing any API call. Our approach is unable to infer this behavior, since it cannot observe what the potential inputs to the decision are.

Our current representation of loops limits the programs we can synthesize. We are unable to synthesize programs that loop a fixed number of times and then time out. If Syren is given the traces that result from this pattern, it will instead attempt to discover a condition that is consistent across all the final API calls, which will not exist in general. We omitted the benchmarks from ApiPhany[63] that could not be generated because either they contain nested loops or the loop iterates on parameter of the script. We believe those limitations could be lifted in future work, allowing Syren to synthesize programs with more complex structures.

## 4.7 Related Work

In this section, we provide a more in-depth overview of previous work related to ours. Although we found strong work on problems related to the one Syren solves, there is no previous solution to solve the same problem. We compare our work against techniques that target similar outputs, i.e. programs with API calls. Then, we look at work that consider similar trace-based specifications, which we can mainly classify in programming-by-demonstration. Finally, we look at work similar in their approach: rewriting techniques and syntax-guided synthesis.

**Synthesis with API Calls** SyPet [41], TYGAR [61], RbSyn [64] and ApiPhany [63] propose type-guided approaches to synthesis of programs containing sequences of API calls. All three address component-based synthesis, which focuses on finding a composition of components (API calls or library functions) that implements some desired task. These systems differ from Syren in more than one aspect. First, SyPet, TYGAR, RbSyn, and ApiPhany take as a specification the desired input and output types of a program. Syren, on the other hand, synthesizes a program using logs of the desired task executed manually. The contrasts between their approach and ours go beyond the input specification: in their approach, the main challenge SyPet tackles is the large search space of API calls that it has to consider, which results from the ever-increasing expressivity and size of API libraries. TYGAR and ApiPhany propose additional techniques to better represent and understand API calls, thus effectively reducing this search space. The challenge in our problem is not to discover which API calls need to be made because they are present in the traces, and thus, the API size does not impact our problem. However, we consider more complex interactions between API calls and synthesize data operations that correspond to the non-observable part of the traces. Conversely, due to the nature of our approach, where we synthesize scripts based on logs, we already know which API calls are made, so the complexity of our synthesis procedure is not affected by the size of the underlying APIs. DemoMatch [137] discovers code snippets explaining API usage, but in the authors' own words "While DemoMatch produces code, it is not a program synthesis tool; it is an API discovery tool."

**Programming-by-Demonstration (PBD)** There are examples of work in PBD that target the automation of web tasks, one of the domains to which we applied Syren, but there are significant differences in the setting and method that justify our claim of novelty. Ringer [14] explores ways to represent one recording of a user interacting with a web page and outputs a script that reproduces a single execution of the demonstrated behavior. Our work attempts to generalize multiple executions into a program from logs; the generalization over data required in our setting is not a problem for Ringer. Approaches such as Konure [120], DemoMatch [137], PUMICE [89], and Sketch-n-Sketch [32] are distinct from ours because we synthesize programs only from the logs of the API calls they make, *without* examples of the local operations of the program to be synthesized. None of the works cited above attempts to synthesize hidden functions, especially with conditions that depend on the outputs of visible function calls. Furthermore, some of the approaches rely on interactive demonstrations to generalize their data (WebRobot [37]) or queries (Konure [120]), or on an existing program to extract dynamic traces from (Chisel [94]). Instead, we rely only on a fixed set of examples. Inferring conditionals and loops when a user or algorithm can test the program paths through demonstrations is a different task from inferring them only by optimizing for a cost function. Like Syren, WebRobot [37] and Arborist [90] use a rewriting

strategy to synthesize programs, with speculative rewrites to generalize patterns. However, their method is interactive and relies on the user to validate the heuristic speculations made by the algorithm. In contrast, we rely on data transformation synthesis to validate the applicability of a synthesis rewrite rule (for conditions and loops) and on the cost function to direct the application of rewrite rules. Neither attempts to synthesize decision-making control flow, i.e., if-statements, or loop conditionals (they synthesize for-each loops and while(true) loops, whereas we focus on loops that are stopped by a condition). The synthesis and generalization of these structures is the main challenge in our approach. Our work differentiates itself from Konure [120] in two significant ways. First, Konure is an active learning system that requires querying of the system that needs modeling. Our system is completely passive (closer to PBE than to PBD) and relies only on a fixed set of examples. Second, Konure has full observability over what it generates. For example, the predicates in their DSL are SQL queries, which are observable in the trace. The authors mention that to support more sophisticated implementations in their DSL they suppose that they would need to have a more fine-grained observation of the application. Instead, we propose that fine-grained data operations can be synthesized using a synthesis solver for small expressions. Our paper shows that this challenge requires careful search over the space of rewrite rules.

**Program Rewriting** There is a rich history of work in applying rewriting strategies for program analysis, refactoring, or optimization [104, 128]. Superoptimization [81, 84, 119] problems, where programs must be rewritten to optimize for a given cost, are classic applications of such techniques. Our application differs in that our rewrites are whole-program rewrites that need to consider the state of the program, as opposed to small local rewrites. Note that our approach would not scale when applying many rewrites in an exploratory search: we also need to solve the underlying syntax-guided synthesis problems, which are much more computationally expensive than syntactic rewrite rules. To the best of our knowledge, this work is the first to introduce a system where the application of a rewrite rule is conditional on solving an input-/output-based synthesis problem. Rewrite approaches have also been used to solve other synthesis problems, such as automatic parallelization in Mold [112]. Although they also consider a two-phase approach with refinement and exploration, they use solely fixed rewrite rules instead of our synthesis rules, and the exploratory phase does not have the same expressive power as our synthesis-based solution. Szalinski [104] describes an approach that takes a flat, hard-to-read program and introduces map and fold operators using optimizing rewrites. Although its goal is similar to the rewrite part of our approach, Szalinski has no way to synthesize hidden logic in control flow.

**Off-the-shelf SyGuS solver** The problem Syren tackles could, in theory, be encoded into Syntax-Guided Synthesis (SyGuS) [4] and solved using an off-the-shelf solver, such as CVC5 [13]. We attempted to implement this approach and ran into two problems. First, it is challenging to encode side-effecting functions as uninterpreted functions because the output is not a direct function of the inputs alone. Their output depends on some global state, updated by other functions in the program. More complex encodings can be designed, but they would be a novel contribution on their own, in particular, if they could be made to perform well. With any straightforward encoding, the resulting formula would be too complex to be solved in a reasonable time by CVC5. As explained in Section 4.4.2 → p.48, we were unable to use CVC5 for even the JSON synthesis subproblems.

# 5

# [Ongoing] HyGLAD

***Synthesis of Anomaly Detection Filters from Logs***

**Contents**



Modern-day software systems rely on extensive logging to record information about their operation. Because they are ubiquitous, anomaly detection techniques often use them as a source of data to detect when systems behave unusually. Log events are the effect of operations on a system, and they contain a lot of information relevant for the proper analysis of that system in the event of a failure or a security breach.

The problem of anomaly detection from logs has received a lot of interest but remains unsolved in many settings. The studies in [68, 69, 139] identify some of the key remaining challenges. The first main challenge is that unsupervised approaches are significantly less accurate than supervised approaches, yet reliable data to train supervised approaches is usually not available in practice. In addition to accuracy, existing systems fall short of today's requirements in expressivity, interpretability, and efficiency. This is especially true for anomaly detection techniques that apply to general-purpose logs, with no prior knowledge about their data, which are desirable because they can flag anomalies relating to entities involved in logs from multiple sources. This motivates the need to improve the accuracy of new approaches that do not need to rely on labeled data.

Interpretability of an anomaly finding is lacking in most systems, especially machine learning and statistics-based approaches, which are not able to provide much insight as to why some event is marked as anomalous. It is rare for an anomaly detection approach to provide both detection and explanation capabilities simultaneously, because the models used for detections are not interpretable, or analyze log events in batches, and are unable to pinpoint a single anomalous event.

Finally, existing approaches have mainly focused on improving accuracy for anomaly detection at the expense of efficiency. This is a blocker to the adoption of these techniques in the industry: the volume of logs to analyze and the need for instant alarming make the efficiency of the technique a requirement.

Our algorithm, implemented in HyGLAD , attempts to solve these problems. We focus specifically on the analysis of structured logs without labeled anomalies. Our algorithm learns a model for a set of logs from the past and detects any deviation from those logs in the future; this is a variation of the usual weakly supervised learning setting, where all events in training are assumed to be normal. Our technique is also interpretable: the model generated is a set of readable filters for log events, which allows us to generate an explanation of why some event is anomalous. Interpretability also allows HyGLAD 's users to understand the reasoning behind an anomaly, and to identify eventual false positives, updating the model as it runs. Finally, it is also designed to be efficient: training a model requires a single pass on the logs, and monitoring is designed to work in real-time on a stream of events.

In HyGLAD , we target structured logs that follow a schema with well-defined fields, such as timestamps, event types, and resource and user identifiers. Many applications, such as cloud providers, emit structured logs already by default. Unstructured logs, such as those printed by humans when developing systems, can be parsed and fit into a structure [75]. The underlying structure of these logs makes them inherently more suitable for automated analysis because there is additional information that we can exploit.

The model resulting from training on normal logs is a set of filters built from regular expressions that match the events in the training set logs. The regular expressions in the filters are a generalization of the values that appear in the logs, in the sense that they match at least those values and also other *similar-looking* values that behave in similar ways. However, generalizing solely based on the values themselves would introduce rules that capture undesirable behavior. For example, a user name with read-only permissions should not be captured with the same regex as one with write permissions because the model would then consider any write from the read-only user normal.

The key insight of our approach is that the relations between entities implied by the log events can be used to guide the generalization over the values. The events in a log can be seen as hyper-edges in a graph, where the vertices are the entities appearing in the structured log. The synthesis of regular expressions capturing multiple vertex values is then driven by a similarity measure over the hypergraph. When two values are similar enough in the graph, the algorithm synthesizes a regular expression that captures both and may generalize over them. This is especially desirable when the values contain automatically generated IDs that vary over time and thus have no meaning, but the pattern of the value does. For example, new user names may be generated at each instance of a script running to perform some action, with a common pattern between users but some randomly generated suffix. Only the pattern matters in understanding normal behavior in the logs.

The interpretability of the model solves more challenges than just the possibility to interpret the anomalies identified. This also allows us to palliate one of the issues encountered in our learning model: since we assume there are no anomalies in our training data, the presence of anomalous events can make the resulting model less accurate. Deep learning techniques usually have to show that their approach is robust against noise in the dataset, *i.e.* they forget the noise. Our technique does not forget, but since we can interpret the model we produce, we can amend

```
{
  "userIdentity": {
    "arn": "role/
      AttrService-InstanceRole-BTDN"
  },
  "requestParameters": {
    "instanceID": "i-12345",
    "eventName": "GetInstanceStatus",
    "asnDesc": "AMAZON-AES"
  }
}
```

Figure 5.1: Example of a structured cloud log

it to remove the bad behavior that it has learned.

At the time of writing this thesis proposal, we have not finished running HyGLAD's evaluation. We aim to show that our technique is both efficient and accurate. We will evaluate HyGLAD on two proprietary Cloudtrail datasets, as well as public unstructured datasets.

## 5.1 Motivating Example

Suppose that you have a system running in a cloud provider and it generates sets of structured logs like the one in Figure 5.1. Each of these logs describes the execution of an event. An event is always executed by an entity (described by `"userIdentity"`) with some parameters (`"requestParameters"`). In the event described above, we see that a user with Amazon Resource Name (ARN) `role/AttrService-InstanceRole-BTDN` execute an event named `DescribeInstanceStatus` on some computing instance with ID `i-12345`. Over time, we will observe that different users execute different actions on EC2 instances. Our goal is to create a model of normal events from a set of events that occurred during the normal execution of the program.

Most of the information in these events is represented as strings. Our goal is then to create filters for strings, i.e. a set of rules such that when they are applied to strings in structured logs, those that are not representative of normal behavior get filtered out. Regular expressions (regexes) are a very good fit for this goal, as they allow users to describe patterns that can be efficiently matched against strings.

Consider the following shorthand for different ARNs.

```
arn1 ← role/AttrService-InstanceRole-BTDN
arn2 ← role/AttrService-DataRole-QRIU
arn3 ← role/AttrService-DataRole-AUIB
arn4 ← role/ModelService-InstanceRole-ZXWI
arn5 ← role/ModelService-InstanceRole-PWER
arn6 ← role/AnotherService-DataRole-AUIC
```

Suppose that over time, we observe hundreds of logs from the normal execution of your system, where the ARNs above perform different events, as shown in the following table, on instance `i-12345`.

| **Event name** | arn1 | arn2 | arn3 | arn4 | arn5 | arn6 |
|---|---|---|---|---|---|---|
| CreateInstance | ✓ | | | ✓ | ✓ | |
| StartInstance | | ✓ | ✓ | | | ✓ |
| GetInstanceStatus | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| StopInstance | | ✓ | ✓ | | | ✓ |
| DeleteInstance | ✓ | | | ✓ | ✓ | |

Our goal is to develop an anomaly detection filter, that is, a sequence of rules that capture events representative of normal behavior of a system, and every other event gets flagged as anomalous. In this example, we could consider a solution that simply creates a rule to allow each of the `arn`s above to perform the events we have observed with the same arguments. Every other event is considered anomalous. Although this filter works for the observed events, it is likely not performing the desired functionality. We also want to accept events *like* the ones we have seen, not simply flag everything unobserved as anomalous.

We could also note that all the ARNs shown can be captured by the regex `role/[A-Za-z]{8,11}Service-[A-Za-z]{5,7}Role-[A-Z]{4}`, so another possible filter we could consider would allow any ARN that matches that regex to execute any of the five events above. This would now allow for all the observed events and also others that are similar, but it is still likely not the desired behavior. Note that, in the table above, we see that all ARNs can execute the GetInstanceStatus event on `i-12345`, but all other events are executed only by a subset of the ARNs.

So, can we create rules that allow ARNs similar to `arn1`, `arn4`, and `arn5` to call CreateInstances, GetInstanceStatus, and DeleteInstance, and ARNs similar to `arn2`, `arn3`, and `arn6` to call StartInstances and GetInstanceStatus? To do this, we need a regex that matches `arn1`, `arn4`, and `arn5` but not `arn2`, `arn3`, and `arn6`. Does such a regex exist? To answer this question, we need a regex synthesis engine that, given sets of positive and negative strings, returns a regex that matches all the positive strings and none of the negative strings.

In this example, the answer is yes. The ideal filter rules for this example are

- ARNs that match `role/[A-Za-z]{8,11}Service-DataRole-[A-Z]{4}` can call CreateInstance and DeleteInstance
- ARNs that match `role/[A-Za-z]{8,11}Service-InstanceRole-[A-Z]{4}` can call StartInstance and StopInstance
- ARNs that match `role/[A-Za-z]{8,11}Service-[A-Za-z]{5,7}Role-[A-Z]{4}` can call GetInstanceStatus

In this example, we considered only the interaction between two event fields, the event name and the ARN. As more events and fields come into play, complexity increases, and the two-dimensional table shown above becomes a very large n-dimensional table. For easier visualization and manipulation of the data, we represent the events and their fields as a hypergraph.

## 5.2 Modeling the Ruleset as a Hypergraph

Given a set of accepted observed events in a structured log, we want to build a log anomaly detection algorithm that creates a set of rules such that at least one of those rules accepts each

given event. The set of these rules may then work as an anomaly detection algorithm, which we will also refer to as an anomaly filter.

We start by defining events as triples of key, value (as they appear in the JSON format), and the type of the value.

**Definition 5.1: Event**

An event is a set of triples $(k, v, \tau)$ where $k$ is a key, $v$ is a value and $\tau$ is the type of value $v$. The keys in an event must be unique.

**Definition 5.2: Event Signature**

The signature of an event $\{(k_i, v_i, \tau)\}_{1 \leq i \leq n}$, denoted by $\Sigma(e)$, is the set of keys and types $\{(k_i, \tau_i)\}_{1 \leq i \leq n}$.

**Example 5.1:**

Figure 5.1 shows an event in JSON format. The event can also be represented as the following set of triples:

$$\begin{aligned}\{&(\texttt{userIdentity.arn}, \text{role/AttrService-InstanceRole-BTDN}, \texttt{IAM :: Role}),\\ &(\texttt{requestParameters.instanceID}, \text{i-12345}, \texttt{EC2 :: Instance}),\\ &(\texttt{requestParameters.eventName}, \text{GetInstanceStatus}, \texttt{EC2 :: EventName})\}\end{aligned} \quad (5.1)$$

A *rule* is also a key-value-type triple, where the universes of rule keys and types are the same as those of events. The values of rules are sets of accepted strings, compactly represented as a regular expression (regex). Any event can be transformed into a rule simply by interpreting its values as regexes that only accept that value string.

**Definition 5.3: Rule**

A rule is a set of triples $(k, r, \tau)$ where $k$ is a key, $r$ is a regex pattern, and $\tau$ is the type of values accepted by $r$. As events, the keys in a rule must be unique, and the signature of a rule is the sorted sequence of its keys.

**Definition 5.4: Rule/Hyperedge Signature**

The signature of a rule (or a hyperedge in the hypergraph model) $\{(k_i, r_i, \tau)\}_{1 \leq i \leq n}$, denoted by $\Sigma(e)$, is the set of keys and types $\{(k_i, \tau_i)\}_{1 \leq i \leq n}$.

A rule accepts an event if they have the same signature and the event's value is accepted by the pattern in the rule's value for the same key. Given an edge $e$ with signature $\{(k_i, \tau_i)\}_{1 \leq i \leq n}$, $e[(k_i, \tau_i)]$ denotes the value in the vertex with key $k_i$ and type $\tau_i$.

**Example 5.2:**

The following is an example of a rule that accepts the event in Section 5.2 → p.63.
(userIdentity.arn,
`role/[A-Za-z]{8,11}Service-InstanceRole-[A-Z]{4}`,
IAM::Role).
(requestParameters.instanceID,
`i-12345`,
EC2::Instance)

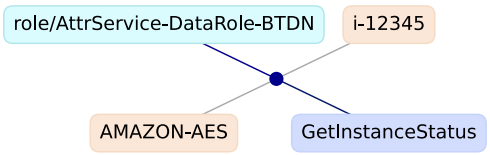



Figure 5.2: A hypergraph showing only one rule that accepts a single event. For readability, each node is labeled only with the triple's value instead of the whole triple. A hyperedge is represented as a dark-blue dot connecting all triples (i.e., nodes) in the rule.

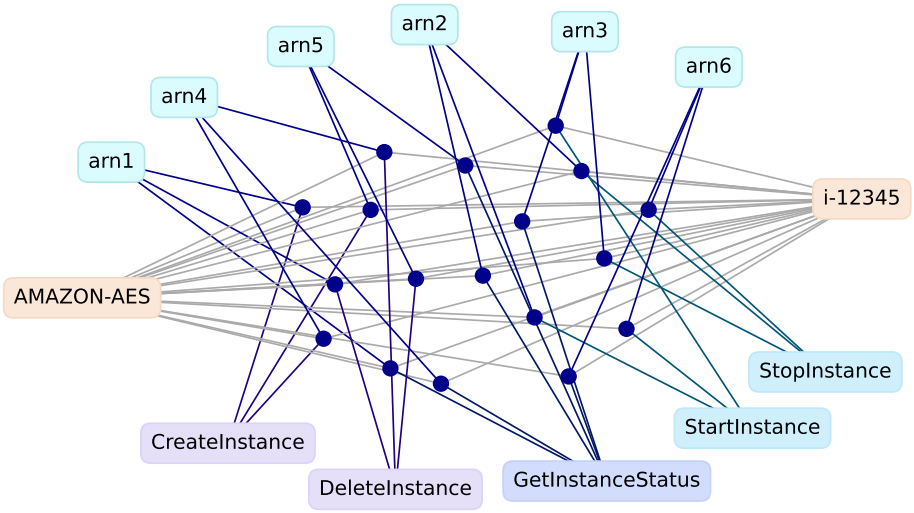



Figure 5.3: A hypergraph representing 18 different rules. Each hyperedge connects to an ARN (light blue vertices on top), and an event name (purple and blue vertices on the bottom). Additionally, they connect to the ARNDesc and the instance name `i-12345`. Since all the hyperedges connect to the same ARNDesc and instance ID, the lines to those are shown in a lighter gray color.

> (requestParameters.eventName,
> `GetInstanceStatus`,
> EC2::EventName)
> The signature of this rule is the same as the signature of the event shown in Section 5.2 → p.63. Otherwise, this rule could not accept the event.

To help us visualize and reason about rules and the values in them, we represent rules in a hypergraph, where each vertex is one of the key-value-type triples, and the hyperedges are the rules. We borrow the formalism from Dai and Gao [35].

> **Definition 5.5: Hypergraph**
> A hypergraph $\mathbb{G}$ is defined by a set of vertices $\mathbb{V}$ and a set of edges $\mathbb{E}$. The structure of the hypergraph can be represented as an incidence matrix $\mathbf{H} \in \{0,1\}^{|\mathbb{V}|\times|\mathbb{E}|}$.

Figure 5.2 shows an example of a hypergraph with a single rule that accepts only the event in Section 5.2 → p.63.

The goal of our approach is to output a set of rules, which we call a filter. A filter accepts

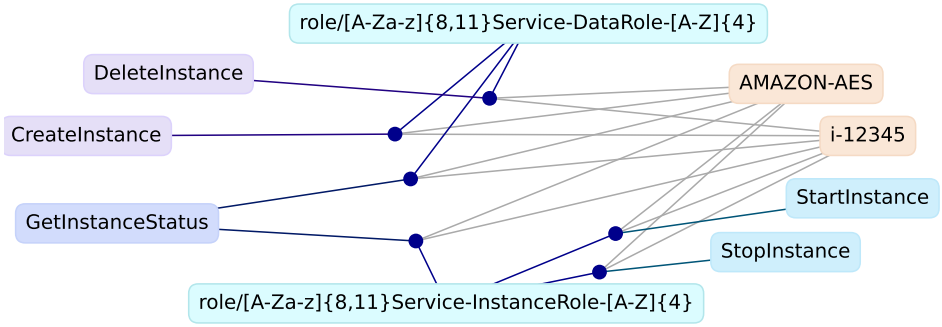



Figure 5.4: A hypergraph showing 6 rules.

an event if any of its rules accept it. The set of rules can be represented in a single hypergraph, allowing us to visualize and reason about the relations between events and their triples. Figure 5.3 shows the hypergraph resulting from all the rules created by converting all the events described in the previous section to rules. In this graph, we are using again the shorthand that renames the ARNs into `arn[1-6]`. From the graph structure, we can observe the same relation that we saw before: ARNs 2, 3, and 6 have the same hyperedge neighbors because they call the same events. The same is true for ARNs 1, 4, 5.

After building this graph model, our goal is to *generalize* rules in this filter. So, we modify the hypergraph in a way that allows more and more events to be accepted, but in such a way that the new accepted events behave similarly to the ones we have observed. So, for example, we want to allow ARNs similar to `arn1` to execute `CreateInstance`, but not `arn2` or ARNs similar to `arn2`, because we have not observed `arn2` performing this event. To generalize the ruleset we *compress our hypergraph by merging similar nodes*. To ensure we generalize conservatively, we merge nodes that have the same key, the same type, similar values, and similar edge structures.

Figure 5.4 shows the graph that results from merging `arn1`, `arn4`, and `arn5` into one single node, and `arn2`, `arn3`, and `arn6` into another. The ruleset in Figure 5.4 → p.65 is a generalization of the ruleset in Figure 5.3 → p.64, since the latter accepts all the events accepted by the former. To choose which nodes to merge, we use a graph compression strategy.

## 5.3 Compressing The Event Graph

HyGLAD 's approach starts by creating a ruleset, modeled as a hypergraph, that accepts exactly the events that have been observed. Then, our goal is to compress this hypergraph in a way that allows similar events to be accepted by the ruleset. To do so, we select similar rule entities, which we represent as nodes in the hypergraph, and we merge them.

### 5.3.1 Merging Vertices

The algorithm consists in merging vertices together as much as possible. In this section we assume the existence of two oracles: MergeRegex and SimScore. MergeRegex takes two sets of regular expressions $R^+$ and $R^-$ as input and returns a new regular expression that subsumes the expressions in $R^+$ (positive examples) and does not intersect with the expressions in $R^-$. In most cases, we will be merging two regexes together and $|R^+| = 2$. The second oracle, SimScore,

takes two nodes and a hypergraph and returns a score measuring the similarity between the two nodes, taking into account their neighborhood in that hypergraph.

**Definition 5.6: Inter-neighbor Relation**

The inter-neighbor relation $N \subset \mathbb{V} \times \mathbb{E}$ on a hypergraph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ with incidence matrix $\mathbf{H}$ is defined as:

$$N = \{(v, e) \mid \mathbf{H}(v, e) = 1, v \in \mathbb{V}, e \in \mathbb{E}\}$$

**Definition 5.7: Hyperedge Neighbor Set**

The hyperedge inter-neighbor set of vertex $v \in \mathbb{V}$ is defined as:

$$N_e(v) = \{e \mid vNe, v \in \mathbb{V}, e \in \mathbb{E}\}$$

**Invariant: Rule Uniqueness** The algorithm maintains the invariant that the edges in the graph are unique: there are no two edges with the same signature (same keys and types) and with intersecting values (*e.g.* when values are regexes). This can be expressed in terms of how many edges (*i.e.* rules) match a given event:

**Definition 5.8: Edge Uniqueness and Event Matching**

The edges in $\mathbb{E}$ are unique iff for any given event $\eta = \{(k_i, r_i, \tau_i)\}_{1 \leq i \leq n}$, there is at most one edge $e \in \mathbb{E}$ such that:

- $e$ has the same signature as $\eta$, i.e. there exists values $s_1, \ldots, s_n$ such that $e = \{(k_i, s_i, \tau_i)\}_{1 \leq i \leq n}$
- the values match, i.e. $\forall 1 \leq i \leq n \cdot s_i \simeq r_i$.

In order to maintain this invariant, merging regexes has to be done with global information about the other edges with the same signature. When merging two vertices $v_1$ and $v_2$, the set of negative examples $R^-$ is is calculated by taking all the edges that are not in the immediate neighborhood of $v_1$ and $v_2$ but have the same signature, and extract the values corresponding to the same key and type in those edges that have the same values for the keys that are not $k_1$. Note that by construction, it should be true that $r_1 \notin R^-$ and $r_2 \notin R^-$. Formally, let

$$S := \{e \mid e \in \mathbb{E} \wedge \exists e' \in N_e(v_1) \cup N_e(v_2) \cdot \Sigma(e') = \Sigma(e) \wedge \\ \forall (k, \tau) \in \Sigma(e) \text{ s.t. } k \neq k_1, \tau \neq \tau_1 \cdot e[(k, \tau)] = e'[(k, \tau)]\} \quad (5.2)$$

be the set of edges that are not in the neighborhood of $v_1$ and $v_2$ but have the same signature as some edge in the neighborhood and differ from that edge only by the value at $k_1, \tau_1$. Then the set of negative examples is:

$$R^- := \bigcup_{e \in S} \{e[(k_1, \tau_1)]\}$$

**Fully merging two vertices** When merging two vertices, we take in account the similarity between the hyper-edges that are connected to those two vertices. Two vertices $v_1 = (k_1, r_1, \tau_1)$ and $v_2 = (k_2, r_2, \tau_2)$ in a hypergraph $\mathbb{G}$ can be merged together if:

- $k_1 = k_2$ and $\tau_1 = \tau_2$, that is, they have the same type and key.
- $\text{MergeRegex}(\{r_1, r_2\}, R^-) \neq \emptyset$, that is, their values can be captured by the same regex.

- $\text{SimScore}(v_1, v_2, \mathbb{G}) > c$ where $c$ is some predefined threshold for similarity.

If two vertices can be merged, then the graph is updated by removing vertices $v_1$ and $v_2$, and replacing $v_1$ and $v_2$ in the edges by a new vertex $v_3 = (k_1, \text{MergeRegex}(\{r_1, r_2\}, R^-), \tau_1)$. Duplicate edges are merged together.

**Partially merging vertices** Sometimes $\text{SimScore}(v_1, v_2, \mathbb{G}) \leq c$ but there exists a subgraph $H \subset \mathbb{G}$ such that $\text{SimScore}(v_1, v_2, H) > c$. A new vertex $v_3 = (k_1, \text{MergeRegex}(\{r_1, r_2\}, R^-), \tau_1)$ is added to the graph, and $v_1$ and $v_2$ are replaced by $v_3$ in the edges in $H$. The edges in $\mathbb{G} \setminus H$ are unchanged. Intuitively, the nodes are similar only if some of the hyperedges are considered.

**Algorithm progress** The algorithm merges vertices together until a fixpoint is reached and there is no possible merge left. Partially merging two vertices adds a new vertex to the graph while keeping the two vertices. However, in all cases, an update to the graph results in fewer or equal number of edges.

### 5.3.2 Merging Regular Expressions

The merging of two graph nodes depends on $\text{MergeRegex}(\{r_1, r_2\}, R^-)$ being able to synthesize a regex that matches all the words matched by $r_1$ and $r_2$, but does not match any of the words matched by any of the regexes in $R^-$. This operation may run more than once for every vertex merge in the execution of the algorithm. There is a trivial implementation of MergeRegex: Compute the union of $r_1$ and $r_2$, $r_1|r_2$. If any of the words matched by some $r^- \in R^-$ is also matched by $r_1|r_2$, fail. Otherwise, return $r_1|r_2$. The problem with this trivial solution is that computing the intersection of two arbitrary regexes may be exponential in the size of the regexes [74], and, by successively computing the union between any two arbitrary regexes, we allow the node regexes to grow arbitrarily large, potentially making subsequent calls to MergeRegex prohibitively slow.

So, we opt for an implementation of MergeRegex that instead of the union $r_1|r_2$, outputs a generalization $r^\bullet$ such that all words matched by the union are also matched by $r^\bullet$. At the same time, we ensure $r^\bullet$ is small by restricting our regexes to a specific subset of regular expressions sufficient to represent the strings usually encountered in logs.

The strings we encounter in the structured logs we target follow a very precise structure. They correspond to IDs (user IDs, resource IDs), hashes, or date-time formats. These are strings that share constant prefixes, suffixes, or infixes interleaved with variable substrings and standard separators, like '-' or '/'. In the case of hashes, they typically consist of sequences of alphanumeric characters of a finite length.

In our regular expression language, we consider two base types of regex units:

1. unions of string literals (ULs), which match one or more specific strings, and
2. repeat character classes (RCs), of the form `c{min, max}` with `min` $\leq$ `max` finite positive integers, which match any sequence of characters in character class `c` of length `min` to `max` (inclusive).

We use the acronym RCL to refer to a regex that is *either* an RC or an UL. Then, our language of regex filters consists of any concatenation of sequences of RCL, which we refer to as $\text{RCL}^+$. This grammar is formalized in Figure 5.5 → p.68.

| | | | |
|---|---|---|---|
| RCL | ::= | RC \| UL | *RCL is either a RC or a UL,* |
| | \| | RC? \| UL? | *or an optional RC or optional UL.* |
| RC | ::= | S{N} | *A RC is a string literal repeated N times,* |
| | \| | S{N, M} | *or repeated between N and M times.* |
| UL | ::= | S \| UL\|S | *UL is a union of string literals.* |
| S | ::= | *string* | *String literal.* |
| N | ::= | *int* | *Integer literal.* |
| M | ::= | *int* | *Integer literal.* |

Figure 5.5: HyGLAD's regex synthesis DSL

**Example 5.3:**

The regexes `role/`, `i-`, and `Attr|Model|Another` are ULs. The regex `[0-9]{5,5}` is a RC. Thus, all regexes above are RCLs, and the regexes `i-[0-9]{5,5}` and `role/(?:Attr|Model|Another)` are RCL$^+$s.

For any set of strings, it is always possible to efficiently compute RCL$^+$s that match all strings. Furthermore, for any two RCL$^+$s, it is efficient to compute a third RCL$^+$ that includes their union. Finally, it is efficient to compute the intersection between two RCL$^+$s.

**Example 5.4:**

Consider the strings "i-12345", "i-12739", and "i-12119". The RCL$^+$s `i-(?:12345|12739|12119)`, `i-12(?:345|739|119)`, `i-12[0-9]{3}`, and `i-[0-9]{5}` are all valid RCL$^+$s that match all three strings.

Then, the implementation of MergeRegex($\{r_1, r_2\}, R^-$) starts by enumerating candidate $r^\bullet$s in our sublanguage of regexes and match all the strings matched by $r_1|r_2$. From these candidates, we remove those whose set of matched strings include any string matched by any regex $r^- \in R^-$. For the small regexes we include in our language, this operation can be computed efficiently.

Finally, to pick among valid candidates for a solution, we rely on a heuristic measure of *cost* of an RCL$^+$ to sort possible candidates. We consider two values for the cost function:

1. number of nodes in the regex (*i.e.*, number of tokens and operations). Smaller regexes are desirable because they are more readable and more efficient to match and merge.

2. number of words accepted by the regex. The generalization of regular expressions results in the filter accepting more strings. This generalization should come at a cost, since we are choosing to consider normal values that were not previously observed.

These two components of cost help the algorithm weigh between saving observed strings as literals in a UL or generalizing them as a RC. Computing this cost for arbitrary RCL$^+$s has linear time complexity in the number of nodes of the regex.

### 5.3.3 Measuring Similarity

To assess the similarity of two entities in the graph, it is not sufficient to only look at the two entities alone, but we need to take into account the relations those entities have with others in

the graph. SimRank [77] is an algorithm that uses the idea that "two nodes are similar if they are pointed to by similar nodes". SemRank [99] and later LSimRank [134] modify the algorithm to take into account the similarity between labels in the calculation. We use an algorithm similar to LSimRank [134] adapted to our context: we explain here how we use it on hypergraphs.

**LSimRank** LSimRank defines the similarity between vertices $S(u, v)$ with the following recursive equation:

$$S(u,v) = \begin{cases} \frac{c \cdot L(u,v)}{|In(u)||In(v)|} \Sigma_{a \in In(u), b \in In(v)} S(a,b) & \text{if} \quad u \neq v \\ 1 & \text{if} \quad u = v \end{cases}$$

where $In(x)$ is the set of vertices connected by edges to $x$ and $L(u, v)$ is the label similarity between $u$ and $v$, and $c$ is a *decay factor* in $[0, 1)$.

The recursive equation has a natural iterative solution. Given an adjacency matrix $A$ representing the connection between vertices, a label-similarity matrix $L$ ($L_{ij} = L(i, j)$) and the normalization matrix $N$ such that $N_{ij} = \begin{bmatrix} \frac{1}{|In(i)||In(j)|} & \text{if} \quad i \neq j \\ 0 & \text{if} \quad i = j \end{bmatrix}$.

The similarity matrix $S$ is the limit of the following equation:

$$S_0 = \mathbb{1} \quad S_{n+1} = \mathbb{1} + c \cdot L \circ N \circ A^\top S_n A \tag{5.3}$$

where $\circ$ is the Hadamard product. Note that the sequence converges if $0 \leq c < 1$ [134].

**SimScore: LSimRank for hypergraphs** To adapt the similarity computation to hypergraphs, we have two straightforward solutions: applying LSimRank to a clique expansion of the graph or a star expansion. The two encodings will intuitively yield different solutions: the clique expansion flattens the hyperedges, whereas the star expansion retains the higher-order information.

We choose the second approach to make use of the higher-order relations that represent connections between entities through common events: entities should be similar if they appear in similar events. Since we do not have labels for the vertices introduced by the star expansion, we set the label similarity to be $L(u, v) = 0$ when $u \neq v$ and $u$ or $v$ is a hyper-edge vertex, and $L(u, v) = 1$ when $u = v$. Note that the incidence matrix of the hypergraph $\mathbf{H}$ is the adjacency matrix of the star expansion; it suffices to substitute $A$ for $\mathbf{H}$ in Eq. 5.3.

We compute the similarity matrix iteratively:

$$S_0 = \mathbb{1}_{|\mathbb{V}|+|\mathbb{E}|}$$

$$S_{n+1} = \mathbb{1}_{|\mathbb{V}|+|\mathbb{E}|} + c \cdot L \circ N \circ \mathbf{H}^\top S_n \mathbf{H}$$

In practice, we set a limit $k$ on how many iterations the algorithm goes through. The higher $k$, the farther apart the vertices that contribute to the similarity score are. Because of the star expansion encoding, it makes sense to have $k > 2$ so that at least neighboring vertices are considered (as opposed to solely the hyper-edge vertex of the encoding).

We define $\text{SimScore}(v_1, v_2, \mathbb{G}) = (S_k)_{v_1 v_2}$.

### Label Similarity in Typed Event Graphs

The similarity label $L$ encodes local knowledge about the similarity between entities in the graph. In its most basic form, similarity compares the raw data in each label together. Recall that in our case each vertex label is an triple $(k, r, \tau)$ where $k$ is a key to the value $r$ of (semantic) type $\tau$ in some event. Vertices that should not be merged should not score highly in the similarity matrix. Therefore, we define the case where keys and types are different to return zero:

$$k \neq k' \vee \tau \neq \tau' \implies L((k, r, \tau), (k', r', \tau')) = 0$$

The label similarity function is where we would expect domain knowledge to be encoded. The algorithm is highly adaptable to various domains, provided that the similarity of the label reflects a good measure of similarity between entities of a certain domain. For example, values that belong to some enumeration probably should not be merged together, but values in a continuous domain should be generalized. However, the similarity of the label is purely local: the similarity score computation outlined previously will handle the context. This significantly simplifies the definition of a similarity function.

### Similarity Between Regexes

When the types and keys of two vertices are equal, the similarity of the label considers the values. In our setting, those values can be represented as $\text{RCL}^+$. The distance between the string representation of the regular expressions is not a good measure of their similarity. The distance should be a measure of how close the set of strings accepter by each regex are to each other. We adapt a well know set distance metric, the Hausdorff distance, to measure distance between regular expressions. The distance between two regular expressions $r_1$ and $r_2$ is the Hausdorff distance between the two sets of strings accepted by those regular expressions:

$$d(r_1, r_2) = d_H(S_1, S_2) = \max\left(\sup_{s \in S_1} \delta(s, S_2), \sup_{s' \in S_2} \delta(s', S_1)\right)$$

where $\delta(s, T)$ computes the infimum of the string-distance between string $s$ and the set of strings $T$, given a string distance $d_s$.

$$\delta(s, T) = \inf_{t \in T} d_s(s, t)$$

Since $S_1$ and $S_2$ might be infinite, we compute the distance by only sampling from the sets.

### Using Type Information

When the system is given type-specific information, it can use that type information to specialize the distance metric to a specific domain. For example, one might decide that strings that represents service names should never be considered as strings, because they are semantically an enum. The user can specify whether an object should be viewed as an object of the given semantic type first, or of the datatype. Objects of different types and same datatyp can be merged together if they are datatype-first. Objects of different semantic types cannot be merged together if they are semantic-first. Additionally, one can consider an object of semantic type to be completely non-mergeable; this is the case for values that are never expected to merge with other values, for example API names. While this seems like a significant burden for the user to specify, most of

the algorithm is assumed to rely on using the data types only. The user only has a few semantic types to specify, and this specification is only done once for one source of structured logs. The algorithm is still agnostic to the domain of the log, and the user does not have to tune any other parameters than the discrete information provided.

## 5.4 Evaluation Plan

At the time of writing this proposal, we are still conducting HyGLAD's evaluation, where we aim to answer the following research questions:

**Q1:** How does HyGLAD compare against pre-existing anomaly detection techniques in detecting anomalies in structured logs?

**Q2:** What kind of anomalies can HyGLAD find, and what information are we able to extract from them?

We will test HyGLAD and related work tools in two proprietary Cloudtrail structured log datasets, in the format described in Section 5.1 → p.61. We will also evaluate HyGLAD's performance on the BETH public dataset [71, 72]. In addition to HyGLAD, we will run preexisting deep learning anomaly detection tools in these datasets [38, 40, 97, 107]. We will show a quantitative comparison between the tools and a qualitative analysis of the anomalies found by HyGLAD.

# 6

# [Proposed] DaPDIS

## *Synthesis of Business Processes from Execution Logs*

**Contents**



The two completed works presented in this thesis show a trade-off in the synthesis of programs from partial traces. Abagnale (Chapter 3 → p.11) synthesizes CCAs from network traces, and it is very robust to unstructured errors in the input traces, at the expense of generality of the synthesis domain (the search space pruning techniques it uses are very specific to congestion control) and efficiency (it takes 12 to 24 hours to synthesize one algorithm). Syren (Chapter 4 → p.34), on the other hand, can synthesize general-purpose scripts from partial execution traces much faster (each script is synthesized in 5 minutes) and it is much more general (Syren's language can be easily extended to include new rewrite rules and different PBE synthesis DSLs). Syren's main weakness is that it attempts to produce a program that can generate *all* input traces. It assumes that the input traces are consistent and correct representations of the target program's intended behavior. If there is some inconsistency in the input traces that Syren's synthesis DSL cannot capture, the synthesis procedure fails. This assumption limits Syren's applicability to real-world domains where traces are collected from error- or variation-prone environments, such as those captured from human interactions or remotely monitored systems. These variations can be action reordering, errors in input values, or the presence of outlier behaviors that are not necessarily representative of the intended task.

As the last project in this thesis, we propose to develop DaPDiS, a novel synthesis framework based on Syren but extended with the ability to handle noisy or inconsistent traces. Figure 6.1 shows where the proposed work stands in comparison to the previous systems, in the plot style as Figure 1.1 → p.3. Our goal is to retain Syren 's generality while adding robustness to noisy or inconsistent traces. To evaluate DaPDiS, we will synthesize business processes from records of humans performing business processes.

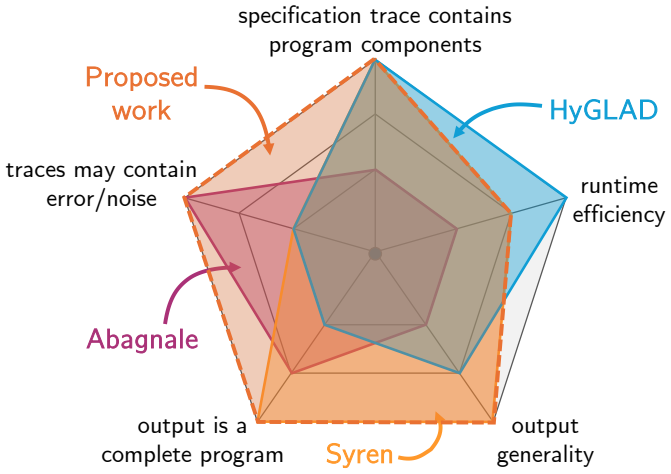



Figure 6.1: Visual comparison of the synthesis systems, including the proposed final system in this thesis. The proposed work, DaPDiS, will extend Syren to be robust to errors in the traces.

## 6.1 Synthesis for Business Process Discovery

Business processes govern the structured execution of tasks within organizations. From onboarding a new employee to processing insurance claims, processes define who does what, in what order, and under what conditions. As organizations evolve, so do their processes, which rely on complex interactions between actors, data, and automated systems. Understanding and improving these processes is the core objective of Business Process Management (BPM) [39]. Business process discovery, a subfield of process mining, aims to automatically reconstruct a process model from recorded executions, in the form of event logs [1]. These logs capture the sequence of activities performed, their timestamps, associated resources, and relevant data payloads. Discovery allows organizations to verify compliance, automate parts of processes, or adapt processes to new contexts.

Syren was originally designed to synthesize executable programs from partial traces. Its main strength is its ability to infer hidden functions and control structures using a combination of program rewriting and SyGuS. Business processes discovery requires the inference of hidden logic between visible actions, which may use arbitrary data in the visible actions. This makes Syren very well-suited for the BPM domain.

However, Syren's correctness constraint, which states that the synthesized program must be able to produce *every* input trace makes it too brittle to work with BPM logs. In business contexts, process logs very often contain deviations caused by user error, system anomalies, or concurrent execution of independent tasks. Our goal is thus to develop DaPDiS, which relies on a synthesis procedure akin to that of Syren, but is resilient to errors in the input traces.

## 6.2 Preliminary Methods

DaPDiS will be a new system that extends Syren with the ability to account for recording errors, concurrency, human error, or variations in the input traces. We hope to achieve these goals by exploring four preliminary ideas.

**Anomaly detection preprocessing.** One way to add robustness to errors in DaPDiS's synthesis procedure is to eliminate noisy traces before the synthesis begins. This would entail a preprocessing step that applies a classification or clustering methodology to partition the input traces into groups of consistent behaviors and flag anomalous behaviors. We will explore ways to adapt the strategy used in HyGLAD (Chapter 5 →p.59), which builds regex-based models of trace events that reflect the normal behavior of a system and identifies anomalous events in the traces, for this purpose. In addition, HyGLAD supplies explanations of why a certain trace event is flagged as an anomaly, providing additional information about the discrepancy in the trace that can potentially be useful for guiding DaPDiS in the synthesis. By filtering or deprioritizing anomalous traces, DaPDiS may focus on uncovering patterns that are representative of the dominant process behavior.

**Optimizing for trace coverage.** Instead of ensuring that the synthesized program exactly reproduces every trace, we can relax DaPDiS's synthesis objective to find a program that explains *as many input traces as possible*. Then, trace coverage becomes a measurable, tunable objective: if a candidate program cannot generate all the input traces, it may be considered as a potential solution until another optimizer that covers more traces is found. This means that not being able to reproduce one or more input traces does not invalidate the candidate program.

**Similarity-based cost functions.** Another way to make DaPDiS robust to noise is by incorporating a trace similarity metric into its cost function. Similarly to Abagnale's approach, traces that are slightly different from those generated by the candidate program are still considered successful, but their difference from the desired trace adds to the cost according to a distance metric. For example, variations in visible events order, inconsequential argument differences, or missing optional steps should not prevent the program from being accepted. This tolerance is essential when dealing with human-executed tasks, where rigid adherence to strict sequence or timing is unrealistic.

**DSL extensions for concurrency.** Finally, DaPDiS could extend Syren's underlying DSL with operations that predict the kind of errors we expect to encounter in the traces. For example, events that are independent can be represented as parallel blocks, allowing the trace resulting from the program execution to arbitrarily interleave them. This change fundamentally expands the expressiveness of the synthesized programs.

## 6.3 Evaluation Plan

To evaluate DaPDiS, we will use benchmarks from the process mining community. Specifically, we target datasets from the BPI Challenge series [17, 18, 19], which provide annotated logs of real-world business processes. We will also test DaPDiS on synthetic datasets where the ground truth is known. This will allow us to evaluate the system's behavior under controlled levels of error in the traces.

# 7

# Proposed Timeline

Table 7.1 outlines the proposed timeline from the presentation of this thesis proposal to the defense of the thesis, covering the period from June 2025 to August 2026. In the summer of 2025, I plan to complete the final requirements of my PhD before the thesis defense: the thesis proposal and writing skills requirement, consisting of writing a blog post targeted at the CS community. I will also incorporate the feedback I receive on the proposal. In June, I will present Syren at PLDI 2025 in Seoul. In July, I will begin planning and laying out the groundwork on DaPDiS, focusing on the literature review and collection of benchmarks for evaluation of the system. In August, I will prepare the camera-ready version of the HyGLAD paper if it gets accepted at ASE'25, or prepare the resubmission to KDD'26 if it gets rejected. From late summer through the fall, I will be interning at Amazon. In December, I will resume the design and implementation of DaPDiS.

In the spring of 2026, I will continue developing DaPDiS, completing the system's design and implementation by March, shifting the focus to writing the paper by late spring. Finally, in the summer of 2026, I will write my thesis and schedule the defense, thus finishing the PhD.

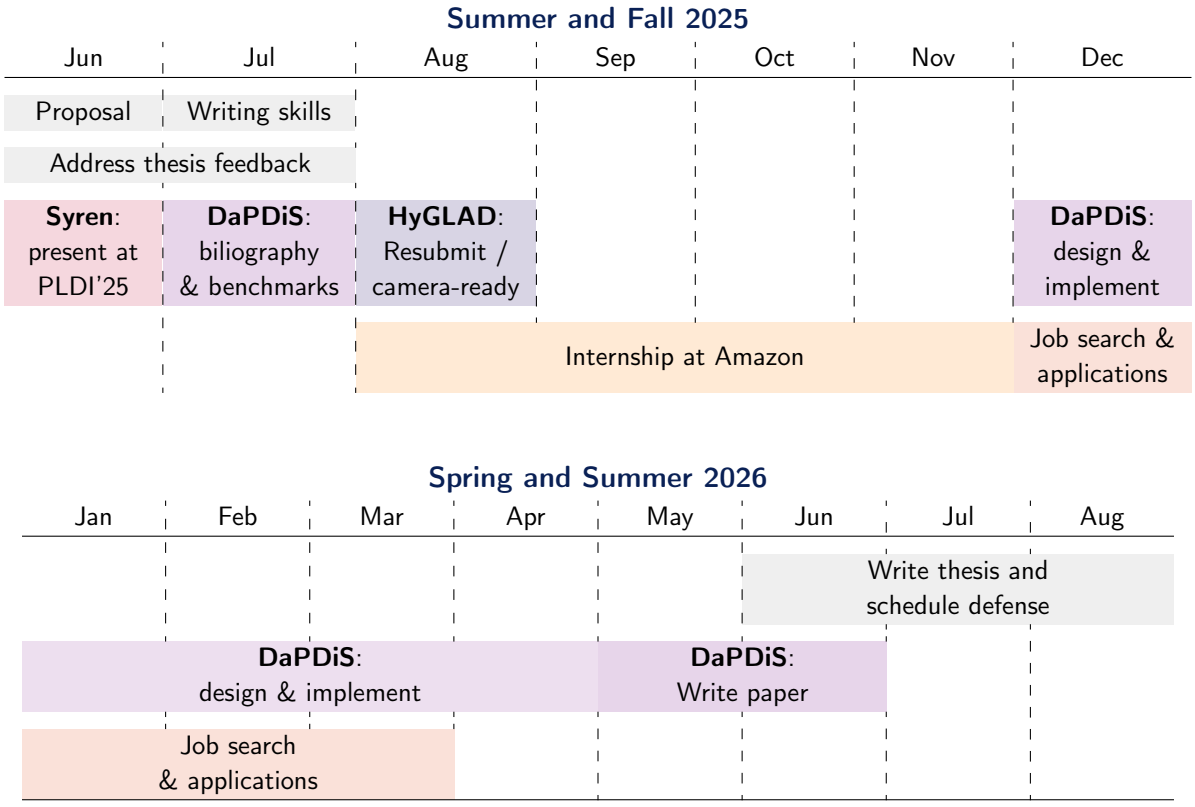



Table 7.1: Proposed timeline from thesis proposal until thesis defense

# 8

# Conclusion

The ubiquity of execution traces in computing systems presents an opportunity to use them as passive specifications for program synthesis. Traces are automatically collected during the regular operation of systems and log information about their behavior in various scenarios. Unlike traditional synthesis approaches that rely on explicit, high-effort specifications, such as logical formulas, input-output pairs, or natural language, trace-based synthesis allows users to derive meaningful programs from data already readily available to them. This can help system managers understand, analyze, and evaluate their systems, automate manual tasks, or reverse engineer other systems or components where the source code is unavailable.

This thesis addresses the challenge of synthesizing stateful programs from execution traces, even when those traces are incomplete, noisy, and lack explicit representations of the desired program's internal state. We describe in depth three synthesis systems, Abagnale, Syren, and HyGLAD, each tackling different aspects of this challenge. Abagnale focuses on reverse-engineering CCAs from network traces. It introduces an optimization-based synthesis approach that tolerates noise and missing data by minimizing the distance metric between candidate and observed behaviors. This method synthesizes simplified models of CCA behavior without access to their implementations. Syren synthesizes general-purpose programs from partial traces where only some of the function calls are visible. It uses optimizing rewrites to introduce control flow and infer hidden functions, enabling the synthesis of complete, executable programs that go beyond the observed behavior. HyGLAD, still under development, shifts focus to anomaly detection. Rather than reconstructing the system, it synthesizes regex-based filters that model normal behavior and flag anomalies in unseen execution traces. Based on the insights from these systems, the proposed next step is DaPDiS, a synthesis approach for data-aware business process discovery. Our goal is to generate executable process models from logs produced by humans carrying out complex tasks. These logs are often noisy and inconsistent, so DaPDiS will combine Syren 's general-purpose synthesis approach, Abagnale 's noise robustness, and HyGLAD 's anomaly detection to learn generalizable, correct-by-construction business process logic.

The successful implementation of DaPDiS will allow us to *synthesize provably correct stateful programs from execution traces, even when the traces are incomplete, noisy, and lack information about the program's internal state*, as we set out to do in the thesis statement. The synthesized programs will help programmers and other computer users analyze their systems, automate manual tasks, or reverse-engineer systems whose source code is not available.

# Bibliography


[1] W. M. P. van der Aalst. *Process Mining: Data Science in Action*. Springer, 2016 (cit. on p. 73).

[2] Anup Agarwal et al. "Automating Network Heuristic Design and Analysis". In: *HotNets*. 2022 (cit. on p. 33).

[3] Anup Agarwal et al. "Towards Provably Performant Congestion Control". In: *NSDI*. 2024 (cit. on p. 33).

[4] Rajeev Alur et al. "Syntax-guided synthesis". In: *FMCAD*. Portland, OR, USA: IEEE, 2013, pp. 1–8. DOI: 10.1109/FMCAD.2013.6679385 (cit. on pp. 8, 58).

[5] Anonymized. "Hypergraph-Guided Regex Filter Synthesis for Anomaly Detection". In: *Under submission*. 2025 (cit. on p. 2).

[6] Anthropic. *Introducing Claude 3.5 Sonnnet*. https://www.anthropic.com/news/claude-3-5-sonnet. June 20, 2024 (cit. on p. 55).

[7] Venkat Arun et al. "Toward Formally Verifying Congestion Control Behavior". In: *SIGCOMM*. 2021 (cit. on pp. 11, 14).

[8] Rukshani Athapathu et al. "Prudentia: Measuring Congestion Control Harm on the Internet". In: *SIGCOMM N2Women Workshop*. 2020 (cit. on p. 14).

[9] AWS. *AWS Automation Runbooks Reference*. https://docs.aws.amazon.com/systems-manager-automation-runbooks/latest/userguide/automation-runbook-reference.html. Mar. 11, 2023 (cit. on pp. 36, 53).

[10] AWS. *What is AWS?* https://aws.amazon.com/what-is-aws/. Mar. 11, 2023 (cit. on p. 36).

[11] Andrea Baiocchi, Angelo P Castellani, Francesco Vacirca, et al. "YeAH-TCP: Yet Another Highspeed TCP". In: *Proc. PFLDnet*. Vol. 7. 2007 (cit. on p. 23).

[12] Matej Balog et al. "DeepCoder: Learning to Write Programs". In: *ICLR (Poster)*. 2017 (cit. on p. 15).

[13] Haniel Barbosa et al. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *TACAS (1)*. Vol. 13243. Lecture Notes in Computer Science. Munich, Germany: Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24 (cit. on pp. 50, 58).

[14] Shaon Barman et al. "Ringer: web automation by demonstration". In: *OOPSLA*. Amsterdam, The Netherlands: ACM, 2016, pp. 748–764. DOI: 10.1145/2983990.2984020 (cit. on p. 57).

[15] Donald J. Berndt and James Clifford. "Using Dynamic Time Warping to Find Patterns in Time Series". In: *KDD Workshop*. 1994 (cit. on p. 21).

[16] Blink. *Blink | The Security Automation Copilot*. https://www.blinkops.com/. Mar. 11, 2023 (cit. on pp. 36, 53).

[17] *BPI Challenge 2020*. https://data.4tu.nl/collections/BPI_Challenge_2020/5065541 (cit. on p. 74).

[18] *BPI Challenge 2021*. https://icpmconference.org/2020/bpi-challenge/ (cit. on p. 74).

[19] *BPI Challenges: 10 Years of Real-life Datasets*. https://www.tf-pm.org/newsletter/newsletter-stream-2-05-2020/bpi-challenges-10-years-of-real-life-datasets (cit. on p. 74).

[20] L. Brakmo. "TCP-NV: Congestion Avoidance for Data Centers". In: *Linux Plumbers Conference* (2010) (cit. on pp. 23, 28).

[21] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. "TCP Vegas: New Techniques for Congestion Detection and Avoidance". In: *SIGCOMM*. 1994 (cit. on pp. 18, 23, 28).

[22] Ricardo Brancas et al. "Towards Reliable SQL Synthesis: Fuzzing-Based Evaluation and Disambiguation". In: *FASE*. Vol. 14573. Lecture Notes in Computer Science. Luxembourg City, Luxembourg: Springer, 2024, pp. 232–254. DOI: 10.1007/978-3-031-57259-3_11 (cit. on pp. 12, 22, 50).

[23] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. Dec. 2017. DOI: 10.17487/RFC8259. URL: https://www.rfc-editor.org/info/rfc8259 (cit. on p. 49).

[24] Deklin Caban, Devdeep Ray, and Srinivasan Seshan. "Understanding Congestion Control for Cloud Game Streaming". Pittsburgh, PA, USA, 2020 (cit. on p. 11).

[25] Carlo Caini and Rosario Firrincieli. "TCP Hybla: a TCP Enhancement for Heterogeneous Networks". In: *Int. J. Satell. Commun. Netw.* 22.5 (2004) (cit. on pp. 23, 27).

[26] Neal Cardwell et al. "BBR: Congestion-Based Congestion Control". In: *ACM Queue* 14 (2016) (cit. on pp. 18, 23, 26).

[27] Nicolas Chan, Elizabeth Polgreen, and Sanjit A. Seshia. "Gradient Descent over Metagrammars for Syntax-Guided Synthesis". In: *CoRR* abs/2007.06677 (2020). arXiv: 2007.06677. URL: https://arxiv.org/abs/2007.06677 (cit. on p. 32).

[28] Haoxian Chen, Anduo Wang, and Boon Thau Loo. "Towards Example-Guided Network Synthesis". In: *APNet*. 2018 (cit. on p. 15).

[29] Mark Chen et al. "Evaluating Large Language Models Trained on Code". In: *CoRR* abs/2107.03374 (2021). arXiv: 2107.03374. URL: https://arxiv.org/abs/2107.03374 (cit. on pp. 5, 15).

[30] Qiaochu Chen et al. "Multi-Modal Synthesis of Regular Expressions". In: *PLDI*. 2020 (cit. on p. 15).

[31] Dah-Ming Chiu and Raj Jain. "Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks". In: *Comput. Networks* 17 (1989) (cit. on p. 11).

[32] Ravi Chugh et al. “Programmatic and direct manipulation, together at last”. In: *PLDI*. Santa Barbara, CA, USA: ACM, 2016, pp. 341–354. DOI: 10.1145/2908080.2908103 (cit. on p. 57).

[33] Stephen A. Cook. “The Complexity of Theorem-Proving Procedures”. In: *STOC*. ACM, 1971, pp. 151–158 (cit. on p. 8).

[34] Allen Cypher et al., eds. *Watch what I do: programming by demonstration*. Cambridge, MA, USA: MIT Press, 1993. ISBN: 0262032139 (cit. on p. 2).

[35] Qionghai Dai and Yue Gao. “Mathematical Foundations of Hypergraph”. en. In: *Hypergraph Computation*. Ed. by Qionghai Dai and Yue Gao. Singapore: Springer Nature, 2023, pp. 19–40. ISBN: 978-981-9901-85-2. DOI: 10.1007/978-981-99-0185-2_2. URL: https://doi.org/10.1007/978-981-99-0185-2_2 (visited on 10/01/2024) (cit. on p. 64).

[36] Amogh Dhamdhere et al. “Inferring Persistent Interdomain Congestion”. In: *SIGCOMM*. 2018 (cit. on p. 14).

[37] Rui Dong et al. “WebRobot: web robotic process automation using interactive programming-by-demonstration”. In: *PLDI*. San Diego, CA, USA: ACM, 2022, pp. 152–167. DOI: 10.1145/3519939.3523711 (cit. on p. 57).

[38] Min Du et al. “DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning”. In: *CCS*. ACM, 2017, pp. 1285–1298 (cit. on p. 71).

[39] M. Dumas et al. *Fundamentals of Business Process Management*. Springer, 2018 (cit. on p. 73).

[40] Amir Farzad and T. Aaron Gulliver. “Unsupervised log message anomaly detection”. In: *ICT Express* 6.3 (2020), pp. 229–237 (cit. on p. 71).

[41] Yu Feng et al. “Component-based synthesis for complex APIs”. In: *POPL*. Paris, France: ACM, 2017, pp. 599–612. DOI: 10.1145/3009837.3009851 (cit. on pp. 20, 57).

[42] Yu Feng et al. “Component-Based Synthesis of Table Consolidation and Transformation Tasks From Examples”. In: *PLDI*. 2017 (cit. on pp. 15, 32).

[43] Yu Feng et al. “Program Synthesis Using Conflict-Driven Learning”. In: *PLDI*. 2018 (cit. on p. 15).

[44] Margarida Ferreira et al. “Counterfeiting Congestion Control Algorithms”. In: *HotNets*. ACM, 2021, pp. 132–139 (cit. on p. 2).

[45] Margarida Ferreira et al. “Counterfeiting Congestion Control Algorithms”. In: *HotNets*. 2021 (cit. on pp. 13, 15, 16, 33).

[46] Margarida Ferreira et al. “FOREST: An Interactive Multi-tree Synthesizer for Regular Expressions”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Vol. 12651. Lecture Notes in Computer Science. Springer, 2021, pp. 152–169. DOI: 10.1007/978-3-030-72016-2_9. URL: https://doi.org/10.1007/978-3-030-72016-2%5C_9 (cit. on pp. 1, 7).

[47] Margarida Ferreira et al. “Program Synthesis from Partial Traces”. In: *Under submission*. 2025 (cit. on pp. 2, 45, 53, 55).

[48] Margarida Ferreira et al. *Program Synthesis from Partial Traces*. 2025. DOI: 10.48550/arXiv.2504.14480. arXiv: 2504.14480 [cs.PL]. URL: https://arxiv.org/abs/2504.14480 (cit. on p. 2).

[49] Margarida Ferreira et al. “Reverse-Engineering Congestion Control Algorithm Behavior”. In: *IMC*. ACM, 2024, pp. 401–414 (cit. on p. 2).

[50] Margarida Ferreira et al. “Reverse-Engineering Congestion Control Algorithm Behavior”. In: *IMC*. Madrid, Spain: ACM, 2024, pp. 401–414. DOI: 10.1145/3646547.3688443 (cit. on p. 50).

[51] John K. Feser, Swarat Chaudhuri, and Isil Dillig. “Synthesizing Data Structure Transformations from Input-Output Examples”. In: *PLDI*. 2015 (cit. on pp. 14, 32).

[52] Sally Floyd. *HighSpeed TCP for Large Congestion Windows*. https://www.ietf.org/rfc/rfc3649.txt. 2003 (cit. on p. 29).

[53] Sadjad Fouladi et al. “Salsify: Low-Latency Network Video through Tighter Integration between a Video Codec and a Transport Protocol”. In: *NSDI*. 2018 (cit. on p. 14).

[54] Jeff Friesen. “Extracting JSON Values with JsonPath: Document Processing for Java SE”. In: *Java XML and JSON: Document Processing for Java SE*. Berkeley, CA, USA: Apress Berkeley, CA, Jan. 2019, pp. 299–322. ISBN: 978-1-4842-4329-9. DOI: 10.1007/978-1-4842-4330-5_10 (cit. on p. 49).

[55] Cheng Peng Fu and Soung C. Liew. “TCP Veno: TCP Enhancement for Transmission over Wireless Access Networks”. In: *IEEE J. Sel. Areas Commun.* 21.2 (2003) (cit. on p. 23).

[56] Sishuai Gong, Usama Naseer, and Theophilus Benson. “Inspector Gadget: A Framework for Inferring TCP Congestion Control Algorithms and Protocol Configurations”. In: *TMA*. IFIP, 2020 (cit. on pp. 14, 16).

[57] C. Cordell Green. “Application of Theorem Proving to Problem Solving”. In: *IJCAI*. William Kaufmann, 1969, pp. 219–240 (cit. on p. 1).

[58] Sumit Gulwani. “Automating string processing in spreadsheets using input-output examples”. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. Ed. by Thomas Ball and Mooly Sagiv. ACM, 2011, pp. 317–330. DOI: 10.1145/1926385.1926423. URL: https://doi.org/10.1145/1926385.1926423 (cit. on p. 5).

[59] Sumit Gulwani. “Programming by Examples - and its applications in Data Wrangling”. In: *Dependable Software Systems Engineering*. Ed. by Javier Esparza, Orna Grumberg, and Salomon Sickert. Vol. 45. 2016 (cit. on p. 32).

[60] Sumit Gulwani, William R. Harris, and Rishabh Singh. “Spreadsheet Data Manipulation Using Examples”. In: *CACM* 55 (2012) (cit. on pp. 1, 14, 32).

[61] Zheng Guo et al. “Program synthesis by type-guided abstraction refinement”. In: *POPL*. New Orleans, LA, United States: ACM, 2020, 12:1–12:28. DOI: 10.1145/3371080. URL: https://doi.org/10.1145/3371080 (cit. on p. 57).

[62] Zheng Guo et al. “Type-Directed Program Synthesis for RESTful APIs”. In: *PLDI*. 2022 (cit. on p. 20).

[63] Zheng Guo et al. “Type-directed program synthesis for RESTful APIs”. In: *PLDI*. San Diego, CA, USA: ACM, 2022, pp. 122–136. DOI: 10.1145/3519939.3523450. URL: https://doi.org/10.1145/3519939.3523450 (cit. on pp. 36, 53, 56, 57).

[64] Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. “RbSyn: Type- and Effect-Guided Program Synthesis”. In: *PLDI*. Virtual, Canada: ACM, 2021, pp. 344–358. ISBN: 9781450383912. DOI: 10.1145/3453483.3454048. URL: https://doi.org/10.1145/3453483.3454048 (cit. on p. 57).

[65] Sangtae Ha, Injong Rhee, and Lisong Xu. “CUBIC: a New TCP-Friendly High-Speed TCP Variant”. In: *ACM SIGOPS Oper. Syst. Rev.* 42.5 (2008) (cit. on p. 23).

[66] Shivam Handa and Martin C. Rinard. “Inductive Program Synthesis Over Noisy Data”. In: *ESEC/SIGSOFT FSE*. 2020 (cit. on p. 32).

[67] David A. Hayes and Grenville J. Armitage. “Revisiting TCP Congestion Control Using Delay Gradients”. In: *Networking (2)*. Springer, 2011 (cit. on pp. 23, 29).

[68] Shilin He et al. “A Survey on Automated Log Analysis for Reliability Engineering”. In: *ACM Comput. Surv.* 54.6 (July 2021), 130:1–130:37. ISSN: 0360-0300. DOI: 10.1145/3460345. URL: https://dl.acm.org/doi/10.1145/3460345 (visited on 09/08/2024) (cit. on p. 59).

[69] Shilin He et al. “A Survey on Automated Log Analysis for Reliability Engineering”. en. In: *ACM Computing Surveys* 54.6 (July 2022), pp. 1–37. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3460345. URL: https://dl.acm.org/doi/10.1145/3460345 (visited on 09/08/2024) (cit. on p. 59).

[70] Dan Hendrycks et al. “Measuring Coding Challenge Competence With APPS”. In: *NeurIPS Datasets and Benchmarks*. 2021 (cit. on p. 1).

[71] Kate Highnam et al. *BETH Dataset Real Cybersecurity Data for Anomaly Detection Research*. https://www.kaggle.com/datasets/katehighnam/beth-dataset. Accessed: 2025-05-15 (cit. on p. 71).

[72] Kate Highnam et al. “Beth dataset: Real cybersecurity data for unsupervised anomaly detection research”. In: *CEUR Workshop Proc.* Vol. 3095. 2021, pp. 1–12 (cit. on p. 71).

[73] Janey C. Hoe. “Improving the Start-Up Behavior of a Congestion Control Scheme for TCP”. In: *SIGCOMM*. 1996 (cit. on p. 23).

[74] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007 (cit. on p. 67).

[75] Yintong Huo et al. “SemParser: A Semantic Parser for Log Analytics”. In: *Proceedings of the 45th International Conference on Software Engineering*. ICSE ’23. Melbourne, Victoria, Australia: IEEE Press, July 2023, pp. 881–893. ISBN: 978-1-66545-701-9. DOI: 10.1109/ICSE48619.2023.00082. URL: https://dl.acm.org/doi/10.1109/ICSE48619.2023.00082 (visited on 08/05/2024) (cit. on p. 60).

[76] Van Jacobson et al. *Bottleneck Bandwidth and RTT (BBR) Congestion Control*. https://elixir.bootlin.com/linux/v4.14/source/net/ipv4/tcp_bbr.c (cit. on p. 23).

[77] Glen Jeh and Jennifer Widom. "SimRank: a measure of structural-context similarity". In: *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. KDD '02. New York, NY, USA: Association for Computing Machinery, July 2002, pp. 538–543. ISBN: 978-1-58113-567-1. DOI: 10.1145/775047.775126. URL: https://dl.acm.org/doi/10.1145/775047.775126 (visited on 10/18/2024) (cit. on p. 69).

[78] Natasha Yogananda Jeppu et al. "Learning Concise Models from Long Execution Traces". In: *DAC*. San Francisco, CA, USA: IEEE, 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218613. URL: https://doi.org/10.1109/DAC18072.2020.9218613 (cit. on p. 2).

[79] Susmit Jha et al. "Oracle-Guided Component-Based Program Synthesis". In: *ICSE*. ACM, 2010 (cit. on pp. 5, 15).

[80] Juyong Jiang et al. "A Survey on Large Language Models for Code Generation". In: *CoRR* abs/2406.00515 (2024) (cit. on p. 1).

[81] Rajeev Joshi, Greg Nelson, and Keith Randall. "Denali: A Goal-Directed Superoptimizer". In: *PLDI*. Berlin, Germany: ACM, 2002, pp. 304–314. ISBN: 1581134630. DOI: 10.1145/512529.512566. URL: https://doi.org/10.1145/512529.512566 (cit. on pp. 36, 58).

[82] Ashwin Kalyan et al. "Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples". In: *ICLR (Poster)*. 2018 (cit. on p. 15).

[83] Tom Kelly. "Scalable TCP: Improving Performance in Highspeed Wide Area Networks". In: *SIGCOMM CCR* 33 (2003) (cit. on p. 23).

[84] Joel Kuepper et al. "CryptOpt: Verified Compilation with Randomized Program Search for Cryptographic Primitives". In: *PLDI*. Orlando, FL, United States: ACM, 2023, pp. 1268–1292. DOI: 10.1145/3591272. URL: https://doi.org/10.1145/3591272 (cit. on pp. 36, 58).

[85] Aleksandar Kuzmanovic and Edward W. Knightly. "TCP-LP: Low-Priority Service via Endpoint Congestion Control". In: *IEEE/ACM Trans. Netw.* 14.4 (2006) (cit. on p. 23).

[86] Adam Langley et al. "The QUIC Transport Protocol: Design and Internet-Scale Deployment". In: *SIGCOMM*. 2017 (cit. on p. 11).

[87] Woosuk Lee et al. "Accelerating Search-Based Program Synthesis Using Learned Probabilistic Models". In: *PLDI*. 2018 (cit. on p. 15).

[88] DJ Leith and R. Shorten. "H-TCP Protocol for High-Speed Long Distance Networks". In: *PFLDNet*. 2004 (cit. on pp. 18, 23, 27).

[89] Toby Jia-Jun Li et al. "PUMICE: A Multi-Modal Agent that Learns Concepts and Conditionals from Natural Language and Demonstrations". In: *UIST*. New Orleans, LA, USA: ACM, 2019, pp. 577–589. DOI: 10.1145/3332165.3347899. URL: https://doi.org/10.1145/3332165.3347899 (cit. on p. 57).

[90] Xiang Li et al. "Efficient Bottom-Up Synthesis for Programs with Local Variables". In: *POPL*. London, UK: ACM, 2024, pp. 1540–1568. DOI: 10.1145/3632894. URL: https://doi.org/10.1145/3632894 (cit. on p. 57).

[91] Shao Liu, Tamer Basar, and R. Srikant. "TCP-Illinois: A Loss- and Delay-Based Congestion Control Algorithm for High-Speed Networks". In: *Perform. Evaluation* 65 (2008) (cit. on pp. 23, 27).

[92] Zohar Manna and Richard J. Waldinger. "Toward Automatic Program Synthesis". In: *Commun. ACM* 14.3 (1971), pp. 151–165 (cit. on pp. 1, 5).

[93] Xavier Marchal et al. "An Analysis of Cloud Gaming Platforms Behaviour Under Synthetic Network Constraints and Real Cellular Networks Conditions". In: *J. Netw. Syst. Manag.* 31.2 (2023) (cit. on p. 13).

[94] Benjamin Mariano et al. "Control-Flow Deobfuscation using Trace-Informed Compositional Program Synthesis". In: *OOPSLA*. Pasadena, CA, USA: ACM, 2024, pp. 2211–2241. URL: https://doi.org/10.1145/3689789 (cit. on p. 57).

[95] Ruben Martins et al. "Trinity: An Extensible Synthesis Framework for Data Science". In: *VLDB* 12.12 (2019) (cit. on pp. 15, 20).

[96] S. Mascolo et al. "TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links". In: *MobiCom*. 2001 (cit. on p. 23).

[97] Weibin Meng et al. "LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs". en. In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*. Macao, China: International Joint Conferences on Artificial Intelligence Organization, Aug. 2019, pp. 4739–4745. ISBN: 978-0-9992411-4-1. DOI: 10.24963/ijcai.2019/658. URL: https://www.ijcai.org/proceedings/2019/658 (visited on 08/01/2024) (cit. on p. 71).

[98] Aaron Meurer et al. "SymPy: symbolic computing in Python". In: *PeerJ Comput. Sci.* 3 (2017) (cit. on p. 20).

[99] Tova Milo, Amit Somech, and Brit Youngmann. "Boosting SimRank with Semantics". en. In: *Proceedings of the 22nd International Conference on Extending Database Technology (EDBT)*. 2019. DOI: 10.5441/002/edbt.2019.05 (cit. on p. 69).

[100] Ayush Mishra et al. "The Great Internet TCP Congestion Control Census". In: *SIGMETRICS*. 2020 (cit. on pp. 11, 13, 14, 16, 25).

[101] Philipp Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications". In: *OSDI*. 2018 (cit. on p. 24).

[102] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. "Z3: An Efficient SMT Solver". In: *TACAS*. 2008 (cit. on p. 24).

[103] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. "Z3: An Efficient SMT Solver". In: *TACAS*. Vol. 4963. Lecture Notes in Computer Science. Budapest, Hungary: Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24 (cit. on p. 53).

[104] Chandrakana Nandi et al. "Synthesizing structured CAD models with equality saturation and inverse transformations". In: *PLDI*. London, UK: ACM, 2020, pp. 31–44. DOI: 10.1145/3385412.3386012. URL: https://doi.org/10.1145/3385412.3386012 (cit. on p. 58).

[105] Akshay Narayan et al. "Restructuring Endpoint Congestion Control". In: *SIGCOMM*. 2018 (cit. on pp. 11, 17).

[106] Vikram Nathan et al. "End-to-End Transport for Video QoE Fairness". In: *SIGCOMM*. 2019 (cit. on p. 14).

[107] Sasho Nedelkoski et al. "Self-attentive classification-based anomaly detection in unstructured logs". In: *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE. 2020, pp. 1196–1201 (cit. on p. 71).

[108] Saswat Padhi et al. *The SyGuS Language Standard Version 2.1*. 2023. DOI: 10.48550/arXiv.2312.06001. arXiv: 2312.06001 [cs.PL]. URL: https://arxiv.org/abs/2312.06001 (cit. on pp. 8, 50).

[109] Jitendra Padhye and Sally Floyd. "On Inferring TCP Behavior". In: *SIGCOMM*. 2001 (cit. on p. 14).

[110] Adithya Abraham Philip et al. "Prudentia: Findings of an Internet Fairness Watchdog". In: *SIGCOMM*. 2024 (cit. on p. 14).

[111] Amir Pnueli and Roni Rosner. "On the Synthesis of a Reactive Module". In: *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 1989, pp. 179–190. DOI: 10.1145/75277.75293. URL: https://doi.org/10.1145/75277.75293 (cit. on p. 5).

[112] Cosmin Radoi et al. "Translating imperative code to MapReduce". In: *OOPSLA*. Portland, OR, USA: ACM, 2014, pp. 909–927. DOI: 10.1145/2714064.2660228. URL: https://doi.org/10.1145/2714064.2660228 (cit. on p. 58).

[113] Daniel Ramos et al. *Are Large Language Models Memorizing Bug Benchmarks?* 2025. arXiv: 2411.13323 [cs.SE]. URL: https://arxiv.org/abs/2411.13323 (cit. on p. 6).

[114] Daniel Ramos et al. "UnchartIt: An Interactive Framework for Program Recovery from Charts". In: *ASE*. 2020 (cit. on pp. 1, 7).

[115] Devdeep Ray. "Integrating Video Codec Design and Network Transport for Emerging Internet Video Streaming Applications". PhD thesis. Carnegie Mellon University, 2022 (cit. on p. 14).

[116] Devdeep Ray and Srinivasan Seshan. "CC-fuzz: Genetic Algorithm-Based Fuzzing for Stress Testing Congestion Control Algorithms". In: *HotNets*. 2022 (cit. on p. 17).

[117] Jan Rüth, Ike Kunze, and Oliver Hohlfeld. "An Empirical View on Content Provider Fairness". In: *TMA*. 2019 (cit. on p. 14).

[118] Constantin Sander et al. "DeePCCI: Deep Learning-based Passive Congestion Control Identification". In: *NetAI@SIGCOMM*. 2019 (cit. on p. 14).

[119] Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stochastic superoptimization". In: *ASPLOS*. Houston, Texas, USA: ACM, 2013, pp. 305–316. DOI: 10.1145/2451116.2451150. URL: https://doi.org/10.1145/2451116.2451150 (cit. on pp. 36, 58).

[120] Jiasi Shen and Martin C. Rinard. "Using active learning to synthesize models of applications that access databases". In: *PLDI*. Phoenix, AZ, USA: ACM, 2019, pp. 269–285. DOI: 10.1145/3314221.3314591. URL: https://doi.org/10.1145/3314221.3314591 (cit. on pp. 57, 58).

[121] Lei Shi et al. "Network Traffic Classification by Program Synthesis". In: *TACAS*. 2021 (cit. on p. 15).

[122] Richard Shin, Illia Polosukhin, and Dawn Song. "Improving Neural Program Synthesis with Inferred Execution Traces". In: *NeurIPS*. Montréal, Canada: Curran Associates, Inc., 2018, pp. 8931–8940. URL: https://proceedings.neurips.cc/paper/2018/hash/7776e88b0c189539098176589250bcba-Abstract.html (cit. on p. 2).

[123] David Canfield Smith. "PYGMALION: A Creative Programming Environment". Other thesis. June 1975. URL: http://worrydream.com/refs/Smith%20-%20Pygmalion.pdf (cit. on p. 5).

[124] Armando Solar-Lezama. *6.S981 Introduction to Program Synthesis Fall 2023 Lecture Notes*. 2023. URL: https://people.csail.mit.edu/asolar/SynthesisCourse/index.htm (cit. on p. 6).

[125] Armando Solar-Lezama et al. "Programming by Sketching for Bit-Streaming Programs". In: *PLDI*. 2005 (cit. on p. 15).

[126] Phillip D. Summers. "A Methodology for Lisp Program Construction from Examples". In: *POPL*. ACM Press, 1976, pp. 68–76 (cit. on p. 5).

[127] Emina Torlak and Rastislav Bodík. "A lightweight symbolic virtual machine for solver-aided host languages". In: *PLDI*. Edinburgh, UK: ACM, 2014, pp. 530–541. DOI: 10.1145/2594291.2594340. URL: https://doi.org/10.1145/2594291.2594340 (cit. on pp. 50, 53).

[128] Eelco Visser. "A Survey of Rewriting Strategies in Program Transformation Systems". In: *Electronic Notes in Theoretical Computer Science* 57 (2001), pp. 109–143. ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(04)00270-1. URL: https://www.sciencedirect.com/science/article/pii/S1571066104002701 (cit. on p. 58).

[129] Richard J. Waldinger and Richard C. T. Lee. "PROW: A Step Toward Automatic Program Writing". In: *IJCAI*. William Kaufmann, 1969, pp. 241–252 (cit. on pp. 1, 5).

[130] Ranysha Ware et al. "CCAnalyzer: An Efficient and Nearly-Passive Congestion Control Classifier". In: *SIGCOMM*. 2024 (cit. on pp. 14, 16, 25, 26, 32).

[131] Ranysha Ware et al. "Modeling BBR's Interactions with Loss-Based Congestion Control". In: *IMC*. 2019 (cit. on pp. 11, 17, 26).

[132] D.X. Wei et al. "FAST TCP: Motivation, Architecture, Algorithms, Performance". In: *IEEE/ACM Trans. on Networking* 14.6 (2006) (cit. on p. 11).

[133] Keith Winstein and Hari Balakrishnan. "TCP ex Machina: Computer-Generated Congestion Control". In: *SIGCOMM*. 2013 (cit. on p. 17).

[134] Yang Wu et al. "LSimRank: Node Similarity in a Labeled Graph". en. In: *Web and Big Data*. Ed. by Xin Wang et al. Cham: Springer International Publishing, 2020, pp. 127–144. ISBN: 978-3-030-60259-8. DOI: 10.1007/978-3-030-60259-8_10 (cit. on p. 69).

[135] Lisong Xu, Khaled Harfoush, and Injong Rhee. "Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks". In: *INFOCOM*. 2004 (cit. on pp. 23, 28).

[136] Qiongwen Xu et al. "Synthesizing safe and efficient kernel extensions for packet processing". In: *SIGCOMM*. Virtal (online): ACM, 2021, pp. 50–64. DOI: 10.1145/3452296.3472929. URL: https://doi.org/10.1145/3452296.3472929 (cit. on p. 36).

[137] Kuat Yessenov, Ivan Kuraj, and Armando Solar-Lezama. "DemoMatch: API discovery from demonstrations". In: *PLDI*. Barcelona, Spain: ACM, 2017, pp. 64–78. DOI: 10.1145/3062341.3062386. URL: https://doi.org/10.1145/3062341.3062386 (cit. on p. 57).

[138] Yifei Yuan et al. "Scenario-Based Programming for SDN Policies". In: *CoNEXT*. 2015 (cit. on p. 15).

[139] Jieming Zhu et al. "Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics". English. In: IEEE Computer Society, Oct. 2023, pp. 355–366. ISBN: 9798350315943. DOI: 10.1109/ISSRE59848.2023.00071. URL: https://www.computer.org/csdl/proceedings-article/issre/2023/159400a355/1RKjmueSehW (visited on 08/19/2024) (cit. on p. 59).