Diffy: Data-Driven Bug Finding for Configurations

SIVA KESAVA REDDY KAKARLA, Microsoft, USA
FRANCIS Y. YAN, Microsoft, USA
RYAN BECKETT, Microsoft, USA

Configuration errors remain a major cause of system failures and service outages. One promising approach to identify configuration errors automatically is to learn common usage patterns (and anti-patterns) using data-driven methods. However, existing data-driven learning approaches analyze only simple configurations (*e.g.*, those with no hierarchical structure), identify only simple types of issues (*e.g.*, type errors), or require extensive domain-specific tuning. In this paper, we present Diffy, the first push-button configuration analyzer that detects likely bugs in structured configurations. From example configurations, Diffy learns a common template, with "holes" that capture their variation. It then applies unsupervised learning to identify anomalous template parameters as likely bugs. We evaluate Diffy on a large cloud provider's wide-area network, an operational 5G network testbed, and MySQL configurations, demonstrating its versatility, performance, and accuracy. During Diffy's development, it caught and prevented a bug in a configuration timer value that had previously caused an outage for the cloud provider.

CCS Concepts: • **Software and its engineering** → *Software configuration management and version control systems*; • **Networks** → *Network reliability*.

Additional Key Words and Phrases: configuration bug finding, template synthesis, anomaly detection

ACM Reference Format:
Siva Kesava Reddy Kakarla, Francis Y. Yan, and Ryan Beckett. 2024. Diffy: Data-Driven Bug Finding for Configurations. *Proc. ACM Program. Lang.* 8, PLDI, Article 155 (June 2024), 30 pages. https://doi.org/10.1145/3656385

1 INTRODUCTION

Ensuring the correct operation of software systems is critical to avoid downtime and performance degradation. While many techniques exist to find bugs in software, increasingly systems are built by assembling existing software artifacts whose behavior is customized through specialized *configuration* languages [9, 10, 18, 27, 29, 31, 50, 51, 61]. For instance, routing protocols that control how packets move across the Internet [50], application orchestration frameworks [18, 27] that manage critical infrastructure, and databases backing popular web services [10] are all managed through extensive configuration.

The increasing use of configuration to program systems has led to a commensurate rise in misconfiguration-related outages [14, 42, 47, 49, 55, 56, 60]. Misconfigurations have taken every major cloud provider offline [42, 55, 56], and are to blame for service outages across numerous industries, including airlines, financial institutions, streaming and e-commerce services [60], and social media [47]. In general, studies have frequently reported misconfigurations as a major source of outages for large production systems [8, 22, 38, 43, 48, 57, 59].

Authors' addresses: Siva Kesava Reddy Kakarla, sivakakarla@microsoft.com, Microsoft, Redmond, USA; Francis Y. Yan, francisy@microsoft.com, Microsoft, Redmond, USA; Ryan Beckett, rybecket@microsoft.com, Microsoft, Redmond, USA.

ACM 2475-1421/2024/6-ART155
https://doi.org/10.1145/3656385

To combat misconfiguration, one line of research has explored the use of data-driven learning methods to identify possible errors as *anomalies* or deviations from normal usage based on a training corpus of example configurations [12, 51, 52, 61]. This approach is appealing because it requires no effort from users to provide specifications of correctness, which may be difficult to obtain or even unknown to the system operators [31].

While the prospect of automatically identifying configuration bugs without specifications is alluring, current incarnations of this approach have several major limitations. First, a majority of them [51, 52, 61] apply only to "flat" configurations whose format comprises a bag of key-value string pairs. While useful in some simple settings, many configurations include richer structures such as sets, lists, tables, maps, objects, or even specialized domain-specific languages (DSLs) [2, 31]. Second, prior work requires that users incorporate extensive domain-specific knowledge into their tools. For instance, parts of configurations are frequently represented using *ad hoc* string data formats. Existing tools typically identify bugs by checking if these strings conform to a set of user-defined types (*e.g.*, a file path, an IP address [31, 61] or that `"ON"` and `"OFF"` are boolean [51]). As a result, they either learn little from data that does not conform to predefined types or require that users arduously add such domain-specific knowledge.

In this paper, we present Diffy[1], the first push-button approach to automatically find likely bugs in arbitrary JSON configurations. We target the JSON [7] format due to its pervasive use in practice, its ability to represent complex structure (*e.g.*, nested lists and objects), and its ease of conversion from other configuration formats (*e.g.*, YAML, XML). While the focus of the paper is JSON, the techniques we present are general and can apply to other hierarchically structured data formats. Diffy requires no user specifications; its input includes only (1) a set of configuration files and (2) an optional set of regular expression tokens to help guide its analysis. From these inputs Diffy synthesizes a template, which is another JSON file that captures the similarities and differences in the configurations. The template contains "holes" that, when filled in with specific values, will recreate the original configurations. Using the synthesized template, Diffy then employs a state-of-the-art unsupervised anomaly detection algorithm known as *isolation forest* [37] to identify likely bugs in configurations as those with anomalous parameters for these "holes".

At the core of Diffy's approach is a novel template synthesis algorithm that attempts to extract similarities between JSON configuration elements. For a given set of configurations, many valid templates exist. Efficiently finding a "good" template, *i.e.*, one that is neither too complex nor too general, is challenging. We define a cost metric to approximate how well a template matches a set of configurations, and then formulate the synthesis problem as a dynamic programming algorithm. The algorithm is related to the idea of string edit distance, but instead finds the minimal *regular-expression-aware* edit distance between strings. Diffy finds a template for other data structures, *e.g.*, lists of lists of strings, by recursively evaluating element template costs and selecting the most suitable template type, such as ordered, unordered, or repetitive.

We assess Diffy against a variety of real configurations, including those from a large cloud provider's wide-area-network (WAN), an operational 5G radio access network (RAN), and MySQL database configurations mined from GitHub. Our results show that Diffy: (1) Generalizes across domains as the first automatic bug finder applicable to arbitrary JSON configurations. (2) Scales well, analyzing thousands of configurations comprising millions of lines within seconds to a few minutes, exhibits a near-linear scaling trend, and outperforms existing data mining tools by 2–3 orders of magnitude. (3) Achieves high precision (up to 97% on the WAN and RAN). And (4) identifies issues comparable to domain-specialized tools.

[1]Diffy finds *iffy* differences in configurations.

During Diffy's development, it automatically identified a bug in a protocol timer configuration value that had previously caused a large outage for the cloud provider's WAN and that was silently reintroduced by faulty automation.

Contributions: In summary, this paper's contributions are to:

- Define the problem of statistical bug finding for configurations in terms of synthesizing a low-cost template to fit the configuration data.
- Solve the template synthesis problem with a novel dynamic programming formulation based on the minimal regular-expression-aware string edit distance and then generalize this approach to richer configuration structures.
- Employ unsupervised learning to identify the most likely bugs from configuration differences based on the synthesized template.
- Implement Diffy based on these ideas, the first tool to automatically detect likely bugs in arbitrary JSON configurations.
- Demonstrate the versatility, performance, and precision of Diffy on three diverse configuration datasets (wide-area network, radio access network, and database).

2 OVERVIEW OF DIFFY

Consider the example configurations used to define networking policies for a wide-area network (WAN) shown in Figure 1. For this example, we picked 48 different configuration files, one from each router to provide to Diffy. These configurations are parsed from vendor-specific formats (*e.g.*, Cisco [9], Juniper [29]), and define low-level settings for various routing protocols, access control policies, queueing and performance, and more. In practice, these configurations are often several tens of thousands of lines long, and thus not easily amenable to human review. For the sake of exposition, we have highly simplified these configurations for the example.

Challenges for Diffy. Analyzing configuration files like for this WAN poses several challenges. First, they make use of complex and domain-specific structure. Route policies (*e.g.*, `"Policy"`), for example, are defined using a domain-specific policy language for matching and modifying layer 3 protocol routes. These policies consist of nested JSON structures whose particular order, size, and elements may or may not be semantically relevant depending on the context. For instance, the order of the declared route length matches (`"LenMatch"`) does not matter, while the order of the `"Actions"` applied to the routes is important. Similarly, the policies are defined using nested objects with different keys, some of which may be missing or intentionally different across configurations. Existing data-driven methods cannot analyze such structure (see §7).

Second, an abundance of data exists in *ad hoc* data formats. For example, route policy actions like autonomous system (AS) number prepending to routes (*e.g.*, `"prepend 65000 65000"`) and route length filtering (*e.g.*, `"/12-/24"` and `"le 8"`) employ unconventional data formats that hinder existing configuration bug finding tools. These tools will learn little to nothing from strings that do not conform to a predefined set of *types* such as an integer, boolean, or a file path [36, 51, 52, 61]. In theory, it is possible to enhance these tools by incorporating specialized parsers, types, and rules tailored to specific domains. This is the approach taken by SelfStarter [31], which models domain-specific concepts such as IP address octets and table reordering rules. However, incorporating such information requires substantial human effort and engineering. Router vendors, as one example, support thousands of distinct configuration parameters, presented in a variety of formats [9, 29]. Furthermore, this process needs to be repeated for each new configuration type that one aims to analyze, thereby restricting the tool's broad applicability.

Synthesizing templates. Diffy takes the configurations in Figure 1 as input, along with regular expression tokens of interest. For the example, we include a single token describing a number

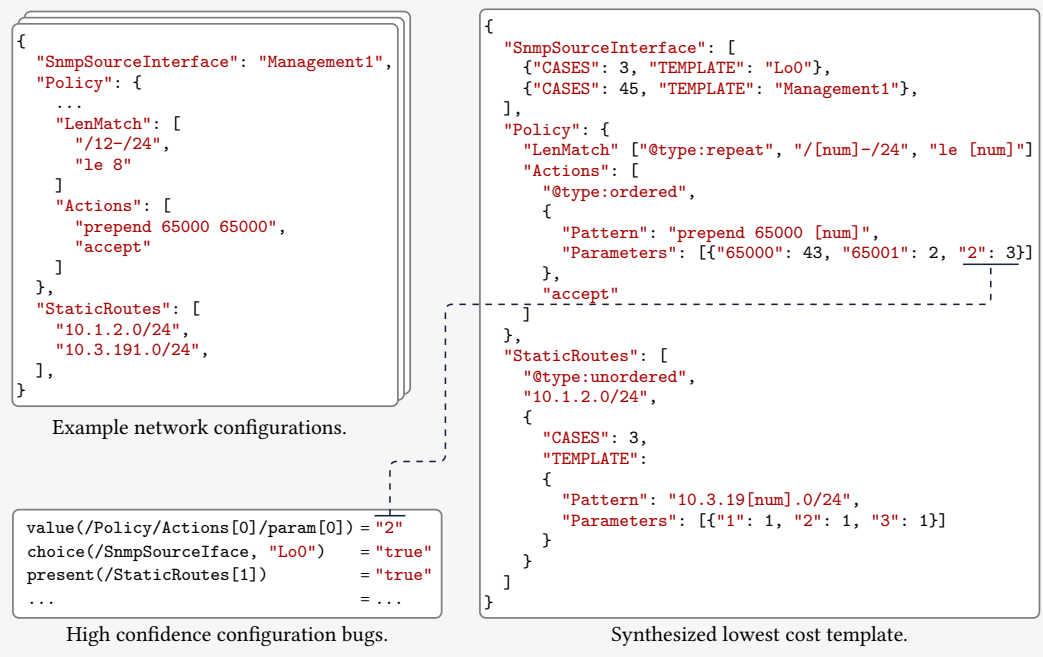


Fig. 1. Example of Diffy in use for anonymized and simplified configurations inspired by a wide-area network (48 configurations in total). Top left shows example configuration snippets for WAN routers. Right shows the synthesized template produced by Diffy. Bottom left shows the high confidence bugs flagged by Diffy.

`[num]` = `"[0-9]+"`. Diffy synthesizes a template (right) based on these inputs. The synthesized template is also a JSON file, but with "holes" that contain values that vary across the configurations.

For the SNMP source interface (`"SnmpSourceIface"`) configuration segment, Diffy identifies a "choice" template indicating that the value is either for a management (`"Management1"`) or loopback (`"Lo0"`) interface, with the former appearing in 45 out of 48 files. As there is no simple pattern to combine these choices, they are left separate.

For the route length filters list (`"LenMatch"`), Diffy learns a `"@type:repeat"` template where all the elements of the list are well summarized with one of two patterns: `"/[num]-24"` or `"le [num]"`. On the other hand, for the `"StaticRoutes"`, Diffy identifies that the better template for these values is an unordered multiset (`"@type:unordered"`). One element of the multiset is always `"10.1.2.0/24"` while a second element is optional and only appears in three configurations. In those three, Diffy has learned that the value of the second list element is `"10.3.19[num].0/24"` and that the value for the "hole" is different for each configuration.

For the route policy actions (`"Actions"`), Diffy synthesizes an ordered sequence template (`"@type:ordered"`) consisting of a prepend action followed by an `"accept"` action that permits the route. Diffy also correctly learns the ad hoc data format for the action that prepends autonomous system (AS) numbers to Border Gateway Protocol (BGP) routes as `"prepend 65000 [num]"`, and understands the values captured by each `[num]` token as well as their frequencies. In the example, the value `"65000"` appears in 43 configurations, `"65001"` in two and `"2"` in three. Despite lacking prior knowledge of Autonomous System (AS) path prepending, Diffy successfully extracts a helpful pattern from the data to detect anomalies as we will see next.

Finding bugs with unsupervised learning. After synthesizing a configuration template, Diffy employs unsupervised learning [37] to identify template parameter *anomalies*, which are shown

for the example in the bottom left of Figure 1. The first anomaly captures that the value of the 0^{th} parameter of the 0^{th} list element in the `"Actions"` list template being `"2"` is a likely bug since most configurations use value `"65000"` or `"65001"`. Indeed this is a misconfiguration likely due to different configuration vendors implementing different AS path prepending semantics. An operator might have expected the second argument of the `"prepend"` action to mean how many times to prepend the AS number 65000 (twice) to the BGP AS path, when instead the configuration expects a space-separated *list* of AS numbers to prepend to the path. Such a mistake can cause other networks to erroneously filter these routes, reducing Internet connectivity. In this case Diffy correctly marks the more frequent `"2"` as an anomaly and not `"65001"` (another private AS number) because `"2"` is more distant from other values. Diffy leverages a state-of-the-art anomaly detection algorithm (§4) that is able to identify such anomalies and achieve consistently accurate detection results.

The second anomaly states that `"SnmpSourceIface"` has pattern `"Lo0"` instead of `"Management1"`[2], and the last anomaly indicates that only 3 out of the 48 configurations have a second static route. Diffy reports all the anomalies with an anomaly likelihood score above a user-provided threshold.

Why it works and limitations. Diffy works on the principle of "bugs as deviant behavior" [13] where bugs are often easily identified as behavior that deviates from the normal. Template synthesis and unsupervised learning go hand-in-hand to identify only the most likely bugs and thereby reduce false positives. Synthesized templates, even when imperfect, concisely summarize all the differences across the configurations and unsupervised learning effectively sifts through these differences to find the most likely bugs while filtering noise. For instance, in the three configurations that have a second static route, the pattern learned for the route is `"10.3.19[num].0/24"`. However, since the value for the "hole" is different in each configuration and the evidence (3 examples) is low, the unsupervised learning algorithm marks these differences as unlikely to be bugs and filters them out. In this way, Diffy often avoids reporting intentional differences as bugs.

Diffy requires the existence of configurations with at least some overlap to be effective as configurations with no similarity provide little opportunity for learning. Diffy also currently limits its analysis to bugs that are identified as deviations in single template parameters. Finding multi-parameter relationships, *e.g.*, if setting 1 is `"true"` then setting 2 must not be `"0"`, is outside the scope of our approach, and we leave it to future work to efficiently learn such relationships.

3 TEMPLATE SYNTHESIS

In this section, we describe the format for configurations and templates (§3.1), explain how Diffy defines costs for templates, show how it efficiently synthesizes low-cost string templates (§3.4), and lift this idea to JSON lists and objects (§3.5, §3.6).

3.1 Configuration and Template Syntax

Before we can explain how Diffy computes a template like that in Figure 1, we must first define the input format and template syntax. A summary of the JSON, template, and anomaly syntax are illustrated in Figure 2. Diffy processes configurations in JSON format [7], defined as a JSON node n. Specifically, a JSON node is either a string s, a list of JSON nodes $[n_1, \ldots, n_k]$, or an object of unordered key-value pairs with string keys and JSON node values $\{s_1 : n_1, \ldots, s_k : n_k\}$. Although JSON has other types of values such as booleans, integers,

Table 1. Example regular expressions.

Example Tokens		
`[any]`	=	`".*"`
`[float]`	=	`"[0-9]+\.[0-9]+"`
`[num]`	=	`"[0-9]+"`
`[alpha]`	=	`"[a-zA-Z]+"`
`[path]`	=	`".*/.*"`
`[upper]`	=	`"[A-Z]+"`
`[hex]`	=	`"[a-fA-F0-9]+"`

[2]In this example, network operators confirmed that indeed a few SNMP source interfaces were mistakenly set as loopback interfaces in the configurations and they should not be originating SNMP messages.

$s \in \Sigma^*, r \in \mathcal{R}, \ell \in$ Label		*definitions*	τ	::= ρ_ℓ	*labeled template*
n	::= s	*json string*	ρ	::= s	*const string*
	$\mid$ $[n_1, \ldots, n_k]$	*json list*		$\mid$ r	*regex token*
	$\mid$ $\{s_1 : n_1, \ldots, s_k : n_k\}$	*json object*		$\mid$ τ^*	*repeat template*
e	::= `value(`ℓ`)`	*identity expr*		$\mid$ τ?	*optional template*
	$\mid$ `integer(`ℓ`)`	*is integer expr*		$\mid$ $\tau_1 \cdot \ldots \cdot \tau_k$	*concatenation*
	$\mid$ `length(`ℓ`)`	*length expr*		$\mid$ $[\tau_1, \ldots, \tau_k]$	*ordered seq*
	$\mid$ `pow2(`ℓ`)`	*power of 2 expr*		$\mid$ $\{\tau_1, \ldots, \tau_k\}$	*unordered set*
	$\mid$ `present(`ℓ`)`	*is present expr*		$\mid$ $(\tau_1 \mid \ldots \mid \tau_k)$	*choice template*
	$\mid$ `choice(`ℓ`,`τ`)`	*choice expr*		$\mid$ $\{s_1 : \tau_1, \ldots, s_k : \tau_k\}$	*object template*

Fig. 2. Configuration JSON and template abstract syntax.

and a special null value, we convert every base value to a string, *e.g.*, `1.2` becomes `"1.2"`, to simplify the algorithm. Despite this simplification, Diffy still finds bugs in numeric values (see §6).

A labeled template $\tau = \rho_\ell$ consists of a template ρ as well as a unique label ℓ that we use to reference ρ. The template may be a constant string s, a regular expression token r such as those in Table 1, or a concatenation of sub-templates ($\tau_1 \cdot \ldots \cdot \tau_k$). For example, `"/[num]-/24"` from the example is a concatenation of `"/"` · `[num]` · `"-/24"`. Templates for lists may be unordered multisets $\{\tau_1, \ldots, \tau_k\}$, ordered sequences $[\tau_1, \ldots, \tau_k]$ or repetitions of a sub-template τ^*. An optional template τ? describes element that may be present or absent. Object templates consist of key-value pairs $\{s_1 : \tau_1, \ldots, s_k : \tau_k\}$. A choice template `(`$\tau_1$ `|` $\ldots$ `|` τ_k`)` signifies a case split, where a configuration will match any of τ_1 through τ_k, useful for representing sets of configuration that correspond to distinct patterns. Finally, e defines expressions over configuration values captured by template labels (*e.g.*, ℓ) that are used to identify anomalies.

Example templates: Going back to Figure 1, we can write the template presented in JSON more formally. The following synthesized template is for the `"LenMatch"` field from Figure 1:

$$(((\texttt{"/"}_a \cdot \texttt{[num]}_b \cdot \texttt{"-/24"}_c)_d \mid (\texttt{"le "}_e \cdot \texttt{[num]}_f)_g)_h)^*_i$$

That is, the field represents a sequence (list) of zero or more elements, where each element is described by a choice of either of the two patterns. The labels `a` through `i` identify the sub-templates.

As another example, consider the `"Actions"` field for the route policy. The synthesized template consists of an ordered sequence of two sub-templates:

$$[(\texttt{"prepend 100 "}_a \cdot \texttt{[num]}_b)_c, \texttt{"accept"}_d]_e$$

Each subtemplate in Diffy has a unique label to allow for writing expressions over configuration values by referencing one or more labels. The expression in Figure 1 states that the value captured by `b` is unlikely to be `"2"`, or `(value(b) = "2")` is an anomaly. This approach is general and permits expressions involving multiple parameters, such as `length(d) < value(b)`, or even the entire string with `c`. Expressions transform configuration values to reveal anomalies. For example, the values `"1"`, `"2"`, `"3"`, `"Z"` for some label `x` may not appear anomalous as strings, but when evaluated over the function `integer(x)`, the string `"Z"` becomes anomalous.

As a final example, the `"StaticRoutes"` field is an unordered list template where one of the fields is optional, and is expressed with the following template:

$$\{\texttt{"10.1.2.0/24"}_a, (\texttt{"10.3.19"}_b \cdot \texttt{[num]}_c \cdot \texttt{".0/24"}_d)\texttt{?}_e\}_f$$

The expression `present(e)` from Figure 1 maps the optional value to a boolean representing whether `e` is present in a configuration. Since most configurations do not have the second static route, those with result `"true"` are flagged as potential bugs. We elaborate on this idea in §4.

For the remainder of the paper, we sometimes elide the labels from templates for readability.

3.2 Preliminaries

We write the set of all JSON nodes as NODE. For $n \in$ NODE, we use $|n|$ to refer to the *length* of n. For a string s, $|s|$ is the length of s. For lists and objects $|[n_1, \ldots, n_i]| = |\{s_1 : n_1, \ldots, s_i : n_i\}| = i$. Given a template $\tau = \rho_\ell$, we define $\mathcal{L}(\tau) = \ell$ as the label for the template. For a set S, the notation $\{\{S\}\}$ refers to the set of all multisets over elements of S. For a regex $r \in \mathcal{R}$ and a string $s \in$ NODE, we write $s \sim r$ to mean that s is in the language of r.

To make the synthesis problem more tractable, we do not allow *choice* templates to be used within a concatenation template. For instance, `"abc"` · `([num] | "d")` is not allowed, while `("abc"` · `[num])|"abcd"` is. This restriction reduces the search space without loss of generality. We similarly assume that nested choice templates are flattened, *e.g.* $\tau_1 | (\tau_2 | \tau_3)$ is written as $(\tau_1 | \tau_2 | \tau_3)$.

The template synthesis problem inputs include (1) k JSON nodes (configurations), (2) a set of regular expressions $\mathcal{R}$, and (3) a cost function $C : \mathcal{R} \to \mathbb{R}$ that assigns a cost $C(r)$ to every character of a string matched by $r \in \mathcal{R}$. We assume that $0 \leq C(r) \leq 1$. Typically one should define $C(r)$ proportionally to the sparsity of the regular expression, *i.e.*, regular expressions that match fewer strings should have lower (better) cost since they are more specific. For instance, `[num]` should have lower cost than an identifier token `[a-zA-Z0-9]+` since this would prioritize matching numbers over identifiers when the text is all digits. To ensure a template always exists, we assume that a default regular expression `[any]` $\in \mathcal{R}$ matches all strings, and that $C($`[any]`$) = 1$.

3.3 Template Cost Function

There are many templates that can represent a given set of configurations. The goal is to find a template that balances generality and complexity. For example, several templates could represent the strings `"Eth1"`, `"Eth2"`, and `"Eth3"`. The token `[any]` covers all three strings but is perhaps too broad because it matches all other strings as well. Alternatively, the template `("Eth1" | "Eth2" | "Eth3")` precisely captures only those three strings but is complex and likely overfits the data. In this case, a compromise such as `"Eth"` · `[num]` strikes a balance between conciseness and specificity.

To synthesize desirable templates, we clearly need a way to compare templates. We do so by defining a heuristic template *cost* function that gives a direct measure of the "quality" of a template. This cost function is inspired by the Minimum Description Length Principle [16] where the cost of a template is related to the amount of data that one would need to transmit to recreate a template's configurations. The cost function is designed to return a normalized score between 0 (best) and 1 (worst) and lends itself to a natural and efficient synthesis algorithm based on dynamic programming. In this way, DIFFY avoids an enumerative search over templates, such as those used in prior data mining tools [21, 45, 46], and scales to large real-world configurations (§6.3).

Cost of string templates. We define a heuristic cost function for templates in Figure 3. The cost function C has three arguments: a template instantiation function T, the number of configurations k, and the template itself τ. The cost function C returns a value in $v \in \mathbb{R}$ such that $0 \leq v \leq 1$. The template instantiation function $T : \mathcal{L} \times \mathbb{N} \to \{\{$NODE$\}\}$ takes a pair of a template label $\ell \in \mathcal{L}$ and a configuration index $i \in \mathbb{N}$ such that $1 \leq i \leq k$ and returns a multiset of JSON nodes representing the matches for the template for configuration i. The function T essentially "fills in" the concrete values for each configuration for each sub-template.

For strings, the cost definitions are simple—the normalized cost for a constant string (CONST-COST) is 0 (ideal), and the normalized cost for a regular expression template r (REGEX-COST) is given by the per-character cost $C(r)$. For a concatenation of templates $\tau_1 \cdot \ldots \cdot \tau_j$, we compute the normalized cost (CONCAT-COST) for each τ_i as $C(T, k, \tau_i)$ and then multiply this cost by the length of the strings matched by each template $\mathcal{S}(T, k, \mathcal{L}(\tau_i))$. This value is then re-normalized by dividing by the total length of the matched strings for each template $\sum_{i=1}^{j} \mathcal{S}(T, k, \mathcal{L}(\tau_i))$.

$$
\begin{array}{lcll}
C(T,k,s_\ell) & \triangleq & 0 & \text{(CONST-COST)} \\
C(T,k,r_\ell) & \triangleq & C(r) & \text{(REGEX-COST)} \\
C(T,k,(\tau_1 \cdot \ldots \cdot \tau_j)_\ell) & \triangleq & \frac{\sum_{i=1}^{j} \mathcal{S}(T,k,\mathcal{L}(\tau_i)) \cdot C(T,k,\tau_i)}{\sum_{i=1}^{j} \mathcal{S}(T,k,\mathcal{L}(\tau_i))} & \text{(CONCAT-COST)} \\
C(T,k,(\tau_1 \mid \ldots \mid \tau_j)_\ell) & \triangleq & \vec{C}(T,k,\ell,1,\mathcal{N}(T,k,\ell),\tau_1 \cdots \tau_j) & \text{(CHOICE-COST)} \\
C(T,k,((\tau_1 \mid \ldots \mid \tau_j)_{\ell'})^*_\ell) & \triangleq & \vec{C}(T,k,\ell,1,\mathcal{S}(T,k,\ell_2),\tau_1 \cdots \tau_j) & \text{(REPEAT-COST)} \\
C(T,k,[\tau_1,\ \ldots,\ \tau_j]_\ell) & \triangleq & \vec{C}(T,k,\ell,\mathcal{M}(T,k,\ell),\mathcal{S}(T,k,\ell),\tau_1 \cdots \tau_j) & \text{(ORD-COST)} \\
C(T,k,\{\tau_1,\ \ldots,\ \tau_j\}_\ell) & \triangleq & \vec{C}(T,k,\ell,\mathcal{M}(T,k,\ell),\mathcal{S}(T,k,\ell),\tau_1 \cdots \tau_j) & \text{(UNORD-COST)} \\
C(T,k,\{s_1:\tau_1,\ \ldots,\ s_j:\tau_j\}_\ell) & \triangleq & \vec{C}(T,k,\ell,\mathcal{M}(T,k,\ell),\mathcal{S}(T,k,\ell),\tau_1 \cdots \tau_j) & \text{(OBJECT-COST)} \\
C(T,k,(\tau?)_\ell) & \triangleq & C(T,k,\tau) & \text{(OPTION-COST)} \\
\hline
\vec{C}(T,k,\ell,\lambda,\gamma,\tau_1 \cdots \tau_j) & \triangleq & 1 - \left(\frac{\sum_{i=1}^{j}(1-C(T,k,\tau_i)) \cdot \mathcal{N}(T,k,\mathcal{L}(\tau_i))}{j \cdot \mathcal{N}(T,k,\ell)}\right) \cdot \left(\frac{\gamma-j}{\gamma-\lambda}\right) & \text{(SEQ-COST)} \\
\mathcal{S}(T,k,\ell) & \triangleq & \sum_{i=1}^{k} \sum_{n \in T(\ell,i)} |n| & \text{(TOTAL-LENGTH)} \\
\mathcal{N}(T,k,\ell) & \triangleq & \sum_{i=1}^{k} |T(\ell,i)| & \text{(NODE-COUNT)} \\
\mathcal{M}(T,k,\ell) & \triangleq & \max_{1 \le i \le k} \max_{n \in T(\ell,i)} |n| & \text{(MAX-LENGTH)}
\end{array}
$$

Fig. 3. Definition of a template cost C given k JSON configurations and a template instantiation function T.

Example: Consider the template $\tau = (\texttt{"Eth"}_a \cdot [\texttt{num}]_b)_c$ that matches strings `"Eth12"` and `"Eth8"`. Assume we have $C([num]) = \frac{1}{5}$ (this exact value is used for illustrative purposes only). The number of configurations $k = 2$ and template instantiation:

$$T(a,1) = T(a,2) = \{\texttt{"Eth"}\} \qquad T(b,1) = \{\texttt{"12"}\} \qquad T(b,2) = \{\texttt{"8"}\}$$
$$T(c,1) = \{\texttt{"Eth12"}\} \qquad T(c,2) = \{\texttt{"Eth8"}\}$$

To compute the cost of τ we first compute the cost of $\texttt{"Eth"}_a$ and $[\texttt{num}]_b$, which are 0 and $\frac{1}{5}$ respectively. We then compute the cost of the concatenation as:

$$\frac{0 \cdot (|\texttt{"Eth"}| + |\texttt{"Eth"}|) + \frac{1}{5} \cdot (|\texttt{"12"}| + |\texttt{"8"}|)}{|\texttt{"Eth"}| + |\texttt{"Eth"}| + |\texttt{"12"}| + |\texttt{"8"}|} = \frac{1}{15}$$

Cost of other templates. Intuitively, the cost for lists and object templates should be related to the costs of their respective elements. However, this is not enough. Elements of a list or set may be missing from some configurations (due to optional elements $\tau?$) or the template may require more or fewer cases to describe the content. For instance, the templates `["Eth1", "Eth2"?, "Eth3"?]` and $(\texttt{"Eth"} \cdot [\texttt{num}])^*$ both describe lists, yet the latter may be preferred due to its simplicity.

Figure 3 defines our cost function for ordered (ORD-COST), unordered (UNORD-COST), repeat (REPEAT-COST), choice (CHOICE-COST), and object (OBJECT-COST) templates in a similar manner. All of them evaluate the cost of a sequence of templates $\tau_1 \cdots \tau_j$ using $\vec{C}$. The function $\vec{C}$ takes the label of callee template ℓ as well as a lower bound λ and upper bound γ on the number of template elements possible for the sequence given the configurations. For each template in the sequence τ_i, we compute the inverse cost, or "benefit", $(1 - C(T,k,\tau_i))$. Each element is then scaled by the fraction of configurations with that element present relative to those in scope $\frac{\mathcal{N}(T,k,\mathcal{L}(\tau_i))}{\mathcal{N}(T,k,\ell)}$ and we divide the result by j to calculate the average benefit. To penalize more complex (longer) templates, we scale the result by $\frac{\gamma-j}{\gamma-\lambda}$, which represents the proximity of the template to the minimal possible length compared to the maximum. The final value represents the "benefit" of the template, and 1 minus this value gives the normalized cost.

For ordered, unordered, and object templates, λ is set to $\mathcal{M}(T,k,\ell)$ the maximum concrete list or object size, and the γ is set to $\mathcal{S}(T,k,\ell)$ the total number list or key elements across all configurations (if all elements required a unique template). For a repeat template, we set $\lambda = 1$ since a single pattern could describe all list elements, and set the same value for γ. A choice template has $\lambda = 1$ and $\gamma = \mathcal{N}(T,k,\ell)$ as the number of configurations in scope.

Example: Consider two JSON lists `["Eth1", "Eth2"]` and `["Eth2", "Eth3"]`. Many templates describe these lists. For instance, the ordered template $\texttt{[("Eth1"}_a\texttt{)?}_b\texttt{, "Eth2"}_c\texttt{, ("Eth3"}_d\texttt{)?}_e\texttt{]}_f$. In this case we have the template instantiation function T defined as:

$$T(a,1) = T(b,1) = \{\texttt{"Eth1"}\} \qquad T(a,2) = T(b,2) = \emptyset \qquad T(c,1) = T(c,2) = \{\texttt{"Eth2"}\}$$
$$T(d,1) = T(e,1) = \emptyset \qquad T(d,2) = T(e,2) = \{\texttt{"Eth3"}\}$$
$$T(f,1) = \{\texttt{["Eth1", "Eth2"]}\} \qquad T(f,2) = \{\texttt{["Eth2", "Eth3"]}\}$$

The cost of the template is given by (ORD-COST), which computes $\vec{C}$ with $\lambda = \mathcal{M}(T,k,f) = 2$ (the largest list size) and $\gamma = \mathcal{S}(T,k,f) = 4$ (the total number of elements). Recursively computing each subtemplate (b, c, e) results in cost 0, and now we calculate the cost as:

$$1 - \left(\frac{1\cdot\mathcal{N}(T,k,b)+1\cdot\mathcal{N}(T,k,c)+1\cdot\mathcal{N}(T,k,e)}{3\cdot\mathcal{N}(T,k,f)}\right)\cdot\left(\tfrac{4-3}{4-2}\right) = 1 - \left(\tfrac{1\cdot1+1\cdot2+1\cdot1}{3\cdot4}\right)\cdot\left(\tfrac{1}{2}\right) = \tfrac{5}{6}$$

In this case, the template cost is high because (1) labels b and e only match elements in half of the configurations and (2) the template consists of 3 list elements rather than the minimal 2 possible for the unordered type.

Now consider another template that also matches the same two lists: $\texttt{("Eth"}_a \cdot \texttt{[num]}_b\texttt{)}_c{}^*{}_d$ again with $C(\texttt{[num]}) = \frac{1}{5}$. For this template, we have the instantiation function T is defined as:

$$T(a,1) = T(a,2) = \{\texttt{"Eth"}, \texttt{"Eth"}\} \qquad T(b,1) = \{\texttt{"1"}, \texttt{"2"}\} \qquad T(b,2) = \{\texttt{"2"}, \texttt{"3"}\}$$
$$T(c,1) = \{\texttt{"Eth1"}, \texttt{"Eth2"}\} \qquad T(c,2) = \{\texttt{"Eth2"}, \texttt{"Eth3"}\}$$
$$T(d,1) = \{\texttt{["Eth1", "Eth2"]}\} \qquad T(d,2) = \{\texttt{["Eth2", "Eth3"]}\}$$

This time, we compute the cost according to (REPEAT-COST), which uses $\vec{C}$ with $\lambda = 1$ and $\gamma = \mathcal{S}(T,k,d) = 4$. The cost $C(T,k,\texttt{("Eth"}_a \cdot \texttt{[num]}_b\texttt{)}_c) = \frac{1}{20}$, and now we calculate:

$$1 - \left(\frac{\frac{19}{20}\cdot\mathcal{N}(T,k,c)}{1\cdot\mathcal{N}(T,k,d)}\right)\cdot\left(\tfrac{4-1}{4-1}\right) = 1 - \left(\frac{\frac{19}{20}\cdot4}{4}\right) = \frac{1}{20}$$

In this case, the repeat template has substantially lower cost than the unordered template because a single pattern captures every element of both lists.

3.4 String Template Synthesis

To efficiently synthesize a low-cost template for a collection of JSON nodes, we first solve the simpler problem of synthesizing a template for a collection of strings (*i.e.*, the base case). We then lift this approach to lists and objects in §3.5. To synthesize string templates efficiently, DIFFY uses a dynamic programming approach to identify a low-cost union of concatenations of templates that match the set S. To further simplify the problem, the algorithm takes two string templates as input and produces a new lowest-cost template that combines these templates. We begin by computing the lowest-cost template for the initial two strings in $s1, s2 \in S$ and then iteratively combine the resulting template with subsequent strings in S until none remain.

We assume all string templates are a sequence of either regex tokens r, or constant string templates of size 1 (*e.g.*, `"10"` is converted to $\texttt{"1"} \cdot \texttt{"0"}$). If the first string template is given as the concatenation $(\tau_1^1 \cdots \tau_m^1)$ and the second template is $(\tau_1^2 \cdots \tau_n^2)$, then we can calculate the lowest

cost template τ that combines $(\tau_1^1 \cdots \tau_i^1)$ and $(\tau_1^2 \cdots \tau_j^2)$ through the dynamic programming cost function $\mathcal{D}(T, k, i, j)$ defined through the recurrence:

$$\mathcal{D}(T,k,i,j) = \min \begin{cases} 0.0 & \textbf{if } i,j = 0 \\ \mathcal{D}(T,k,i-1,j-1) + C(T,k,\tau_i^1)\cdot\mathcal{S}(T,k,\mathcal{L}(\tau_i^1)) & \\ \qquad + C(T,k,\tau_j^2)\cdot\mathcal{S}(T,k,\mathcal{L}(\tau_j^2)) & \textbf{if } \tau_i^1 = \tau_j^2 \\ \mathcal{D}(T,k,a-1,j) + C(r)\cdot\sum_{k=a}^{i}\mathcal{S}(T,k,\mathcal{L}(\tau_k^1)) & \textbf{if } r \in \mathcal{R},\ \texttt{""} \sim r,\ a \in \mathcal{I}(r, \tau_1^1 \cdots \tau_i^1) \\ \mathcal{D}(T,k,i,b-1) + C(r)\cdot\sum_{k=b}^{j}\mathcal{S}(T,k,\mathcal{L}(\tau_k^2)) & \textbf{if } r \in \mathcal{R},\ \texttt{""} \sim r,\ b \in \mathcal{I}(r, \tau_1^2 \cdots \tau_j^2) \\ \mathcal{D}(T,k,a-1,b-1) + C(r)\cdot\sum_{k=a}^{i}\mathcal{S}(T,k,\mathcal{L}(\tau_k^1)) & \\ \qquad + C(r)\cdot\sum_{k=b}^{j}\mathcal{S}(T,k,\mathcal{L}(\tau_k^2)) & \textbf{if } r \in \mathcal{R},\ a \in \mathcal{I}(r, \tau_1^1 \cdots \tau_i^1),\ b \in \mathcal{I}(r, \tau_1^2 \cdots \tau_j^2) \end{cases}$$

This definition takes the current template instantiation function T and number of configurations analyzed so far k (initially $k = 2$). The best template has (un-normalized) cost $\mathcal{D}(T, k, m, n)$. The cost is initially 0.0 and is then updated recursively in two main scenarios. When both sequences have the same next template the previous cost $\mathcal{D}(T, k, i-1, j-1)$, is updated by the normalized costs of the matched i^{th} and j^{th} templates times the number of characters matched by each. All other cases involve a "backwards jump" using regex tokens. For a token $r \in \mathcal{R}$, we consider all indices a and b such that r matches the sub-sequences $\tau_a^1 \cdots \tau_i^1$ and $\tau_b^2 \cdots \tau_j^2$. This logic is represented by an index match function $\mathcal{I}$, which is implemented efficiently using automata. For instance, $\mathcal{I}(\texttt{[num]}, \texttt{"1"} \cdot \texttt{[num]}) = \{1, 2\}$. That is, the sub-sequence starting at index 1 ($\texttt{"1"} \cdot \texttt{[num]}$) matches $r = \texttt{[num]}$, since concatenating two numbers remains a number, and from index 2 ($\texttt{[num]}$) also matches $\texttt{[num]}$. We discuss the details of $\mathcal{I}$ later. In this case, there is a "diagonal" jump, and the lowest cost is updated from $\mathcal{D}(T, k, a-1, b-1)$ by adding the regex cost $C(r)$ for each character matched by the regex. If token r matches an empty string ($\texttt{""} \sim r$) then we can also jump backwards only in a horizontal (1st case) or vertical direction (2nd case).

Example: We illustrate the concept with an example in Figure 4, where we determine the least cost template matching strings that represent memory sizes: `"10M"`, `"15M"`, and `"200"` (uses a default size). We use two tokens, `[num]` and a file path `[path]`, both with $C(r) = \frac{1}{5}$. We first compute the minimum cost template for $(\texttt{"1"}_a \cdot \texttt{"0"}_b \cdot \texttt{"M"}_c)$ and $(\texttt{"1"}_d \cdot \texttt{"5"}_e \cdot \texttt{"M"}_f)$. Figure 4's part ② displays the dynamic programming table that memoizes $\mathcal{D}(T, 2, i, j)$ values, where initially:

$$T(a,1) = \{\texttt{"1"}\}\ T(b,1) = \{\texttt{"0"}\}\ T(c,1) = \{\texttt{"M"}\}\ T(d,2) = \{\texttt{"1"}\}\ T(e,2) = \{\texttt{"5"}\}\ T(f,2) = \{\texttt{"M"}\}$$

and is $\emptyset$ otherwise. By retracing the table from the bottom right corner to the top left corner ($i = 0$, $j = 0$) and identifying the lowest cost case at each step, we retrieve the lowest cost pattern. The new template is $\texttt{"1"}_g \cdot \texttt{[num]}_h \cdot \texttt{"M"}_i$ with fresh labels, and we update T according to the match, *i.e.*, $T(h, 1) = \{\texttt{"0"}\}$ and $T(h, 2) = \{\texttt{"5"}\}$ and so on.

We then repeat this process with $(\texttt{"1"} \cdot \texttt{[num]} \cdot \texttt{"M"})$ and $(\texttt{"2"} \cdot \texttt{"0"} \cdot \texttt{"0"})$ and again display the dynamic programming table for $\mathcal{D}(T, 3, i, j)$. Eventually, the algorithm finds that the lowest cost template involves a "backwards jump" from $i = 2,\ j = 3$ to $i = 0,\ j = 0$ using the `[num]` token, which matches both `"200"` and $\texttt{"1"} \cdot \texttt{[num]}$. Lastly, the `[any]` token matches the final `"M"` template without consuming any characters from `"200"`, resulting in the final template $\boxed{\texttt{[num]} \cdot \texttt{[any]}}$, where `[num]` captures the memory size and `[any]` captures the optional memory unit (*e.g.*, megabytes).

Calculating index matches. In Figure 4, Diffy must compute $\mathcal{I}$. The algorithm must know, for instance, that $\texttt{"1"} \cdot \texttt{[num]}$ matches `[num]` since $1 \in \mathcal{I}(\texttt{[num]}, \texttt{"1"} \cdot \texttt{[num]})$. Similarly, that $\mathcal{I}(\texttt{[num]}, \texttt{"1"} \cdot \texttt{[num]} \cdot \texttt{"M"}) = \emptyset$ since any string ending with `"M"` is not a number. To calculate $\mathcal{I}$ one could check for language containment (*e.g.*, the language of $\texttt{"1"} \cdot \texttt{[num]}$ is a subset of the language of `[num]`).

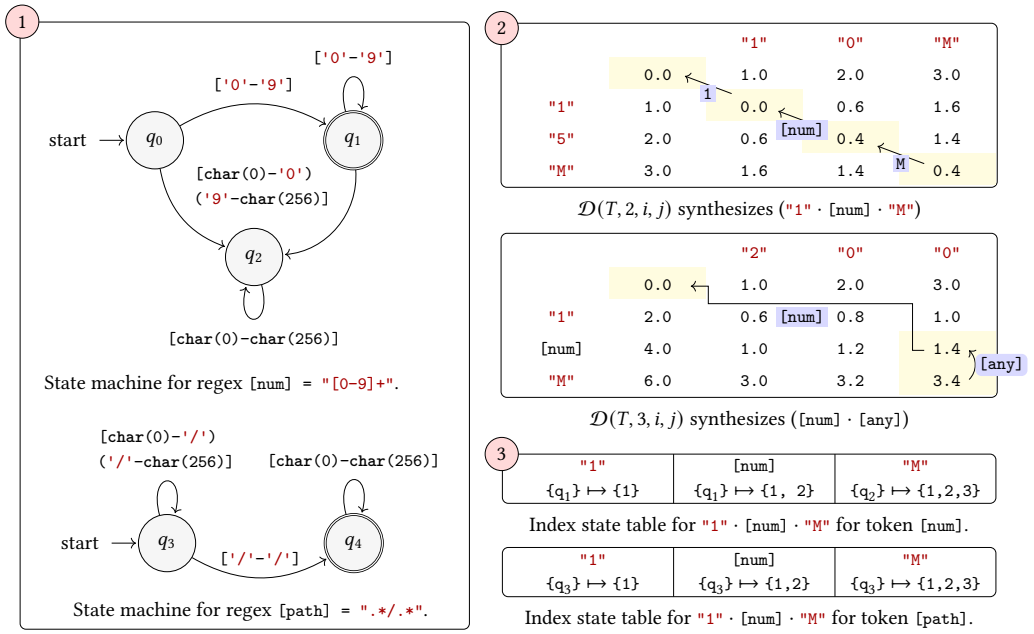


Fig. 4. DIFFY's string template synthesis algorithm for strings "10M", "15M", and "200" using regular expression tokens for numbers [num] = "[0-9]+" and file paths [path] = ".*/.*" with cost $\frac{1}{5}$. The algorithm has three steps: (1) constructing DFAs for each token, (2) synthesizing a template using dynamic programming, and (3) tracking automaton states in index tables. The table entries can either have an exact match with zero cost or jump backward by matching a regex token. An index entry, such as $\{q_3\} \mapsto \{1,2\}$ at [num] for token [path], signifies that the substring starting at index 1 or 2 and ending at [num] leads to $\{q_3\}$ in the FSM of [path]. Since q_3 is not a final state it does not match either substring. The final path through the table results in pattern "1" · [num] · "M" before being adjusting to [num] · [any] for the third string.

However, this approach is far too slow in practice because it calculates $\mathcal{I}$ for every regex r and for every pair of indices, resulting in $\mathcal{O}(m^2 + n^2)$ expensive language containment checks.

Instead, we efficiently compute $\mathcal{I}$ for every substring simultaneously by tracking possible automaton states for each end index, as depicted in Figure 4 (part (3)). For template "1" · [num] · "M", the algorithm begins at state q_0 of the [num] automaton and transitions to state q_1 after processing "1". Since this is an accepting state, "1" matches [num]. Similarly, "1" · [num] matches [num] since transitioning from state q_1 upon seeing a [num], we must remain in state q_1. However, "1" · [num] · "M" will end with "M" (q_2 state), not matching [num]. Simulating a state machine for a token is done using a product automata construction, and we discuss the details for this algorithm in Appendix B for space reasons. This method enables near-linear time token substring matching.

Choice string templates. Whenever the normalized template cost exceeds a threshold (0.5 by default), we create a new template. When evaluating new strings, the algorithm compares them to existing choices and selects the one with the lowest cost. For example, naive synthesis for strings "true", "false", and "True" produces template [any] · "e", with high cost. Instead, because the normalized cost for the template for "true" and "false" exceeds 0.5, we create a choice template. Comparing "True" to both "true" and "false", the former offers a lower-cost match. Thus, the final template is ([any] · "rue" | "false")

While the choice of threshold may seem arbitrary, we show in §6.4 that DIFFY's ability to correctly identify bugs is fairly insensitive to the value of this threshold.

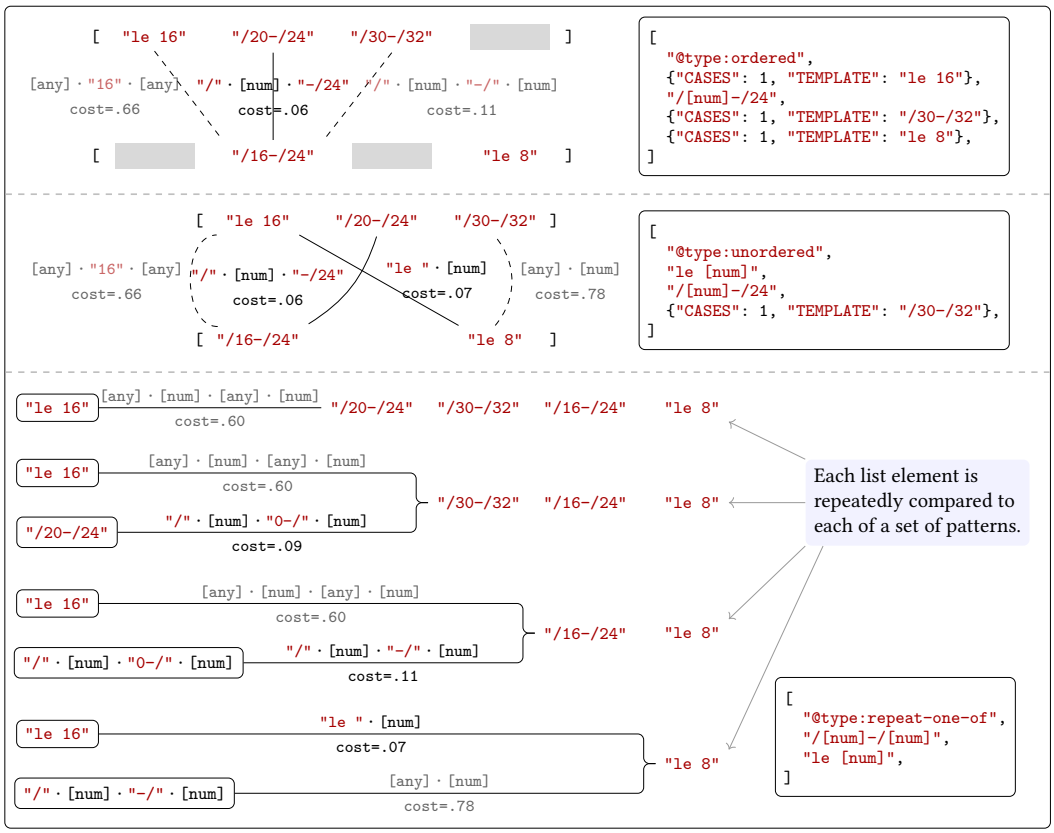


Fig. 5. How Diffy calculates a list template, for input lists `["le 16", "/20-/24", "/30-/32"]` and `["/16-/24", "le 8"]`. The ordered template (top) is determined using sequence alignment [53], where the pairwise element cost is the string template cost. The grey rectangles represent unmatched elements. The first element of the list on the bottom (`"/16-/24"`) is matched with the second element of the list on the top (`"/20-/24"`), and other possible matches not used are greyed out due to their higher costs. The unordered template (middle) is from the lowest-cost matching of elements – *i.e.*, finding the pairs of elements in each list that should be matched together to produce the lowest total cost. The solid lines show the chosen matches, and the dashed lines show some of the matches that were not selected. Finally, Diffy finds the minimum cost repeat template (bottom) by comparing each list element with an accumulated set of patterns.

Complexity. Given two string templates with lengths m and n, computing $\mathcal{D}$ has worst case complexity $O(|\mathcal{R}| \cdot m^2 \cdot n^2)$ in the event that each regular expression $r \in \mathcal{R}$ matches every substring. The quadratic terms exist because, for each element in the table, each regex could potentially match $m \cdot n$ substrings and Diffy must compare every such match to find the one with the lowest cost. In the best-case, the algorithm must do work proportional to $|\mathcal{R}| \cdot m \cdot n$ because it must "fill in" every element of the dynamic programming table similar to the standard string edit distance algorithm. We discuss the optimizations we use to mitigate this worst-case complexity and achieve performance closer to the best case in practice in §5.

3.5 List Template Synthesis

To synthesize a template for two JSON lists, Diffy recursively computes the lowest-cost templates for every pair of list elements. There are three template types: ordered list $[\tau_1, \ldots, \tau_k]$, unordered list $\{\tau_1, \ldots, \tau_k\}$, and repetition τ^*. Diffy calculates the best template for each type and selects the

one with the lowest cost. Figure 5 illustrates the computation of a list template for each type using example lists: `["le 16", "/20-/24", "/30-/32"]` and `["/16-/24", "le 8"]`.

Ordered template. To synthesize the best ordered template, Diffy uses a sequence alignment algorithm to efficiently arrange list elements. Diffy computes pairwise templates for every element in both lists and selects the lowest-cost alignment to form a new template. For unaligned elements, it introduces an optional template. The example's resulting alignment from in Figure 5 is:

`[ "le 16"`	`"/20-/24"`	`"/30-/32"`	`—— ]`
`[ ——`	`"/16-/24"`	`——`	`"le 8" ]`
`[ "le 16"?`	`"/" · [num] · "-/24"`	`"/30-/32"?`	`"le 8"? ]`

Unordered template. To synthesize the optimal unordered template, Diffy calculates the pairwise template cost of list elements and computes the minimum cost matching between the elements, provided their cost falls below the cutoff threshold. The unordered template for the example is: {`"le " · [num]`, `"/" · [num] · "-/24"`, `"/30-/32"?`}, as shown in Figure 5 (middle). This is a better match than the ordered template since the `"le " · [num]` pattern captures multiple matches.

Repeat template. To synthesize a repeat template, Diffy merges elements from each list and applies a greedy clustering algorithm as depicted in Figure 5 (bottom). The algorithm starts by creating a group with the first element (`"le 16"`). Then, it compares the next element (`"/20-/24"`) to the group using a new template. Due to the high cost, `"/20-/24"` forms a separate group. The process continues with the subsequent element `"/30-/32"`, comparing it against both groups. It has a significantly lower cost when combined with `"/20-/24"`, so Diffy updates the second group to be `"/" · [num] · "0-/" · [num]`. The algorithm proceeds until there are no elements are remaining, resulting in the final template of `("le " · [num] | "/" · [num] · "-/" · [num])`*.

3.6 Object Template Synthesis

Handling objects is simpler than lists, as there is only one object template $\{s_1 : \tau_1, \ldots, s_k : \tau_k\}$. To synthesize this template, we identify common keys across the objects recursively compute templates for their values. When keys are present in some configurations but not others, we introduce an optional τ? template for the key. For example, the template of objects {`"addr" : "192.13.4.1"`} and {`"addr" : "192.13.4.2", "len" : "24"`} is {`"addr" : "192.13.4." · [num], "len" : "24"?`}. While this approach typically works well, Diffy's implementation also offers a fuzzy key matching mode that allows object keys to also be templates. Synthesis for these objects resembles unordered template synthesis from § 3.5 for keys. For example, the template for {`"10.0.0.1" : "10"`} and {`"10.0.0.2" : "10"`} would be {`"10.0.0." · [num] : "10"`}.

Handing mismatched types. In certain scenarios, configurations can possess distinct node types. A key might have a string value in one configuration and a list or null value in another. We segregate the configurations based on their types and template them individually. The outcome is presented as a choice template, such as $(\tau_s \mid \tau_l)$ for string and list templates.

4 FINDING BUGS WITH UNSUPERVISED LEARNING

We split Diffy into two orthogonal components — template synthesis and anomaly detection. In this section, we describe the second component that uses a state-of-the-art anomaly detection algorithm known as *isolation forest* [37]. While we describe our approach in detail, we note that it would also be possible to apply any other learning approaches to Diffy's templates.

Isolation forests are a form of unsupervised (or self-supervised) anomaly detection technique. Given a vector of values such as `[1.2, 1.8, 0.9, 2.2, 20.1]`, an isolation forest will assign an "anomaly score" to each input. This score ranges from 0 to 1 where a value with score closer to 1 indicating a higher likelihood of it being an anomaly. In the example, `20.1` would have the highest

anomaly score. Although this example used only single-dimentional inputs, isolation forests also handle multi-dimensional data as well (*i.e.*, a vector of vectors).

Isolation forest is based on the observation that anomalies are both "few and different," *i.e.*, they not only have infrequent occurrences but also consist of values that noticeably differ from the normal, making them easier to separate from other samples. Empirically, this observation leads to more accurate and reliable anomaly detection results compared with other methods that rely solely on frequency, density, or distance [23]. To quantify the observation, isolation forest partitions data randomly and recursively, logging the number of partitions required to isolate each sample. This process is then repeated many times to derive the average number of partitions, which serves as the anomaly score for each sample.

As an unsupervised learning method, isolation forest does not require anomaly labels from users and thus is well suited to our problem where the labels are often not available. Meanwhile, isolation forest is highly scalable and efficient, demonstrating linear time and space complexities.

Embedding configuration data. Given the synthesized template τ from §3, Diffy recursively traverses τ. For each label ℓ in τ, Diffy translates the concrete values $v \in T(\ell, i)$ for each configuration i into a numerical vector that is fed to the isolation forest algorithm. This translation occurs according to the following steps: (1) apply each type of applicable expression (*e.g.*, `present` for an optional template) to get string values (*e.g.*, `"true"`), (2) if the strings from all configurations are numerical, then encode them directly, otherwise sort the strings and then apply a standard ordinal encoding to obtain a numerical representation. Table 2 showcases example expressions that we used, and users can extend Diffy to add additional expressions.

Consider a configuration parameter `x` with values: `"2"`, `"4"`, `"8"`, and `"10"`. The expression `value(x)` just returns `x`, and Diffy translates the values directly as `[2, 4, 8, 10]`, leading to low anomaly scores since all values are close to each other. However, the expression `pow2(x)` maps the inputs to `"true"`, `"true"`, `"true"`, `"false"`, which Diffy then encodes as `[1, 1, 1, 0]`, resulting in `"10"` having a higher anomaly score. Now consider another parameter `y` that captures list values such as `["0x3", "0x0", "0x0"]`, `["0x0", "0x0", "0x0"]`, and `["0x0"]`. The expression `length(y)` computes their lengths as `[3, 3, 1]`, which may identify the last list as a potential bug if most lists are longer.

Table 2. Example expressions for a value with label x that transform the value to another value, with example outputs.

Expression	Description	Examples
`present(x)`	Optional value x has a value.	`"true"`
`value(x)`	The value of x.	`"1.3"`, `"abc"`
`pow2(x)`	Is value x a power of 2.	`"false"`
`integer(x)`	Is value x an integer.	`"true"`
`choice(x,`τ`)`	Does value x match choice τ.	`"false"`
`length(x)`	Length of list x.	`"7"`, `"12"`

Analyzing multiple parameters: In this work, we focus on identifying configuration errors involving only a single configuration parameter. Single parameter errors are by themselves a prevalent category, and encompass many kinds of misconfigurations including typos or "fat finger" mistakes, copy-paste errors, type errors, missing configuration, and more (see §6). We leave learning multi-parameter errors, *i.e.*, correlations across multiple configuration settings, as future work.

5 IMPLEMENTATION OF DIFFY

Diffy is implemented in 4.7K lines of F# and 1.6K lines of C# code. It accepts a file directory with JSON configuration files and an optional set of tokens with associated costs through the command line. It always includes token `[any]` in its analysis. Diffy outputs a template, and a set of potential bugs along with a score for each bug. Diffy accepts many other optional flags to include

or exclude different parts of the configurations from its analysis or filter out different kinds of anomaly expressions from its results. To enhance scalability, Diffy employs several optimizations.

While creating the JSON abstract syntax tree from Figure 2 in memory, Diffy calculates a perfect hash for each node and stores it as part of the node. This hash enables quick approximate equality checks between JSON elements and allows Diffy to efficiently cache calls to its `synthesize` function to prevent redundant work. Furthermore, when synthesizing a template for a set of strings S or lists L, *etc.*, Diffy identifies duplicate JSON nodes between configurations using these hashes, and replaces them with a single copy of the node. This simplification applies recursively and significantly reduces the cost of synthesis.

Recall that the complexity of string synthesis is worst case $O(|\mathcal{R}| \cdot m^2 \cdot n^2)$. We apply several optimizations to avoid this worst-case complexity in practice. First, Diffy's implementation includes an additional rule in the dynamic programming formulation for the `[any]` token that only allows that token to match a single character (*i.e.*, horizontally or vertically). We then post-process the template to coalesce any consecutive `[any]` tokens. Doing so avoids having to consider all backwards jumps. Similarly, we use a greedy regex matching semantics to greatly reduce the number of backwards jumps we must consider (*i.e.*, consider only the maximal match for the regex for each start index).

To speed up the substring matching process, Diffy caches state transitions for index tables from Figure 4. When updating states the cache stores the new set of states as a function of the current state set, token, and new token. This one simple idea removes most of the overhead from tracking automata states, and reduces the index match function $\mathcal{I}$ to linear time in practice, as only a few state sets are typically visited. Finally, Diffy first checks if every string shares a common prefix. If so, it removes this prefix and inserts a constant template at the beginning. It then synthesizes a template for the remaining suffixes.

To improve the performance of synthesizing unordered templates, Diffy uses a greedy matching algorithm that iteratively pairs off the two elements with the lowest cost template. Doing so improves performance and we found does not affect the quality of the final template much.

Lastly, Diffy omits the computation of some list templates if their cost cannot outperform the existing ones. For instance, when an ordered template's cost is low enough that an unordered template could never surpass it, Diffy does not bother synthesizing the best unordered template.

6 EVALUATION

Our primary focus is to examine the versatility, scalability, and accuracy of Diffy. We assess these characteristics through multiple case studies, each representative of a distinct aspect of the network stack. These include the configurations from the wide-area network of a large cloud provider, the radio units of an operational 5G testbed, and a collection of MySQL database configurations mined from GitHub. Due to space limitations, we have included the MySQL analysis in Appendix C. These configurations vary significantly in their structure and scale.

6.1 Case Study 1: Wide-Area Network

We analyzed one of the world's largest backbone networks, a wide area network (WAN) consisting of thousands of routers and millions of configuration lines. Routers in the WAN are categorized by roles, such as edge, border, core, and reflector. Table 3 shows the total number of lines of configuration for each role. For sensitivity reasons, the exact role names and configuration details are anonymized. We apply Diffy to each role since configurations within a role share similar definitions, like prefix lists and community lists, though specifics can vary across routers.

Router's *running* configuration files (*i.e.*, those active on the devices) are stored in a centralized database, providing an accurate snapshot for Diffy. A set of *golden* configurations (*i.e.*, the expected configurations) are also available. The running and golden configurations may vary for

Table 3. Diffy performance on WAN configurations. For each configuration element type, table shows the (**P**)arsing time to read JSON files, the (**S**)ynthesis time and the time for finding (**A**)nomalies. Each entry of the format x / y represents the time taken to template or find anomalies without any tokens (x) and with additional tokens like `[num]` and `[alpha]` (y). All the times are in seconds.

Role	Config line count	Prefix list			Route policy			Other		
		P	S	A	P	S	A	P	S	A
R1	$\mathcal{O}(10^6)$	45	34 / 110	1.20 / 0.60	42	1120 / 1212	3.99 / 2.80	70	43 / 58	4 / 1.8
R2	$\mathcal{O}(10^6)$	34	12 / 24	0.70 / 0.12	27	496 / 537	0.18 / 0.10	26	45 / 115	0.6 / 0.56
R3	$\mathcal{O}(10^6)$	58	29 / 80	1.20 / 0.50	45	1120 / 1284	1.28 / 1.60	85	36 / 55	1.2 / 0.99
R4	$\mathcal{O}(10^5)$	4	3.5 / 8	0.20 / 0.10	4	21 / 34	0.16 / 0.18	5	1.8 / 3.3	0.1 / 0.1
R5	$\mathcal{O}(10^4)$	0.4	1 / 2.4	0.02 / 0.02	0.4	0.81 / 1.1	0.02 / 0.03	0.7	0.23 / 0.27	0.04 / 0.03

a large number of operational reasons. For example, partial or incomplete configuration change rollouts, temporary changes (*e.g.*, QoS settings to alleviate ongoing network congestion), differing maintenance schedules (*e.g.*, operators can only take a limited number of devices offline in a given month for refresh), and many more reasons.

To reduce network outages, the WAN uses a custom "diffing" service that periodically compares the golden and running configurations and reports all current deviations that may need repair. We leverage this service to identify the ground truth (*i.e.*, those differences that are unintentional) and use this data to evaluate the accuracy of Diffy. Because the service reports all differences, it also includes many differences that are ultimately harmless (*e.g.*, the version number for the configurations). For this reason, we focus on evaluating Diffy's precision rather than recall.

Diffy performance. To convert vendor-specific configuration files to JSON, we utilized two parsers: open-source Batfish [17] and a closed-source parser. We employed the latter for prefix lists and route policies, while Batfish handled the rest (*e.g.*, community lists, VRFs, SNMP servers, *etc.*). We used both parsers because Batfish offers broader coverage but also complicates and obscures the original configuration data by expanding and inlining prefix lists and route policies. The internal parser's JSON representations of prefix lists is exemplified in the top section of Figure 1. Using these parsers, we analyzed over 99% of the configuration lines for the WAN routers.

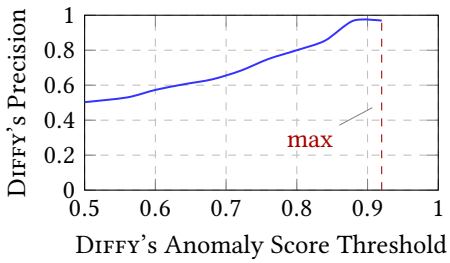


Fig. 6. Precision vs. anomaly score filter for WAN.

Table 3 displays the time Diffy takes to transform the JSONs into internal data structures, synthesize a template, and find anomalies for each role and configuration element type, both with and without tokens. During the learning phase, we had Diffy report all potential anomalies, which were then sorted and filtered by score for operator review. In most instances, Diffy completes within seconds or minutes, even when processing millions of configuration lines. Route policies, being the most complex element, take the longest time. However, even in the worst-case scenario, Diffy completes route policies in approximately 20 minutes.

Accuracy of Diffy. We run Diffy on each role using a minimum anomaly score filter 0.51 to find all issues with some evidence of being an anomaly. For each router and configuration element name we report if Diffy found an anomaly with the element for this router and compare with the "diffing" tool based on golden configurations. From this ground truth, we plot the precision (true vs. false positive ratio) of Diffy as a function of the anomaly score filter in Figure 6. Diffy's anomaly score is closely correlated with its precision against the "diffing" service. For instance, for issues with an anomaly score of 0.8 or higher, roughly 80% of Diffy's findings are confirmed as true positives, and the precision goes up to **97%** true positives with anomaly score 0.88 or higher.

While ground truth exists for route policies and prefix lists, other elements such as SNMP servers are not currently tracked by the diffing tool. For these, we identified a small subset of classes of issues Diffy reported with high score and manually investigated them with network operators. Of these, the operators identified **77%** as true positives, **8%** as false positives, and **15%** as requiring further investigation. Many of the issues represented "cruft" in the configurations from legacy devices and were subsequently addressed by the operators.

Breakdown of anomalies. We show the frequency of each type of anomaly expression from Table 2 across all configuration elements in the WAN in Table 4. Most reported issues for this dataset are related to (1) a configuration element missing or (2) present that should not be, or (3) a parameter value from the template being different.

Template types. We examined the templates for each element type to better understand the use of different list template types described in §3.5. For prefix lists, ordered templates are prevalent, however in some cases where the prefixes differed greatly across routers, Diffy synthesized the general template `([num] · "." · [num] · "." · [num] · "." · [num] · "/" · [num])*`. Route policies mainly use object templates with certain list-type values, for which ordered templates are frequent, as the keys (match, set, or action commands) demand order. Repeat and unordered templates are common for other elements such as communities, AS paths, Virtual Routing and Forwarding (VRFs) and servers as they have set semantics. These observations demonstrate that the different template types are useful in practice for capturing the domain-specific semantics for the various configuration components.

Example Bug: BGP Community Regex Match. Diffy discovered inconsistencies in regular expression filter policies used for matching route community tags. For example, some routers configured a regex starting with a `"^"` character, while most did not. Diffy synthesized the template `[any] · [num] · ":2[1-3]"` and found that the value of `[any]` was typically `""` and only rarely `"^"`. This character, when present, requires a match at the beginning of the BGP community string and may lead to a failed match. The operators acknowledged the problem, standardizing the configurations.

Example Bug: IGP Timer. In another instance, a single router had an IGP routing protocol configuration parameter with a "overload" setting. This setting was supposed to be `"true"` but had previously been erroneously set to a time in seconds (*e.g.*, `"1200"`) on a single router. This switches the value to `"false"` 1200 seconds after a reboot and resulted in the router taking in more traffic than intended. The device did not have the capacity to handle this traffic, and started dropping most of the packets, leading to connectivity issues for many of the cloud's customers. The configuration issue was resolved, but automation later mistakenly reapplied the flawed setting, opening up the potential for another outage. Diffy detected a high-confidence anomaly from template {`"overload" : ("false" | "1200")`} and we were able to alert the operators in time.

Table 4. Percentage of anomalies by expression type for WAN configurations with anomaly threshold 0.8.

Expression	Prefix list	Route policy	Other
`present`	92.0	90.9	38.5
`value`	4.2	7.3	50.6
`pow2`	0.2	1.3	6.2
`integer`	3.6	0.1	4.0
`choice`	0.0	0.4	0.7

6.2 Case Study 2: 5G Radio Access Network

We collected configuration files from a 5G vRAN (virtualized Radio Access Network) testbed developed and operated within a global cloud provider. The testbed represents the state-of-the-art 5G vRAN deployment, consisting of hardware components such as radio units (RUs), vRAN servers, PTP (Precise Time Protocol) grandmaster clocks, network switches, along with 5G software stacks.

Table 5. We compare the string synthesis performance of Diffy and FlashProfile [45]. The "Total calls" column shows the number of string synthesis calls in thousands (§3.4). We also show the mean and median number of strings in each call. The sub-columns show the breakdown by prefix list, route policy, and other.

Role	Total calls (K)			Mean			Median			FlashProfile (s)			Diffy (s)			Speed up (F/D)		
R1	243	871	95	7.5	4.2	13.8	2	3	3	17855	48932	7624	86	53	36	207×	923×	212×
R2	90	395	264	5.2	2.1	5.5	2	2	4	5864	8824	74425	8	11	56	733×	802×	1329×
R3	164	596	100	8.0	2.6	13.5	2	2	2	11855	18356	5985	54	32	25	220×	574×	239×
R4	96	78	4	2.4	2.7	16.8	2	2	12	2419	3015	246	3	1.1	0.7	806×	2740×	351×
R5	47	9	0.2	2.0	2.4	4.0	2	2	4	1139	293	10	0.4	0.2	0.2	2847×	1465×	50×

Our focus in this dataset is on the configurations established for a total of seven RUs in the testbed. These RUs, manufactured by Foxconn for 5G vRAN, use four antennas to transmit and receive radio signals on the n78 band (3.3–3.8 GHz). During operation, each RU loads an XML configuration file that specifies its behavior, *e.g.*, the MAC address of the vRAN server for baseband processing, the IP address of the PTP grandmaster for clock synchronization, as well as many low-level configuration knobs (see Figure 9 for a sample configuration snippet).

Given that the testbed is time-shared among users who frequently experiment with various setups, it is common for multiple variants of the XML configuration to coexist on each RU. To evaluate Diffy, we first acquired 42 RU configurations that were consciously created by previous testbed users, excluding factory defaults. In addition, we recreated several types of bugs commonly encountered during the daily operation of the 5G testbed and we retroactively introduced these bugs back by generating an additional 8 synthetic configurations to simulate the operational experience. Each synthetic RU configuration showcases a distinct category of bug, such as a different firmware version, an old PTP grandmaster IP, or a deprecated VLAN tag for user-plane messages. In total, the final dataset encompasses 50 configurations.

Diffy performance. We converted the 50 XML configurations into JSON and ran Diffy on them to detect anomalies with two tokens: `[num]` and `[hex]`. Diffy completed in 225 ms, with 44 ms spent on parsing, 110 ms on synthesis, and 71 ms on discovering anomalies. Diffy outputs anomalies in descending order by their anomaly scores, and we manually reviewed the 64 flagged issues with an anomaly score of 0.7 or higher. Among these issues, 62 were true positives, resulting in a precision of **96.9%**. Notably, Diffy correctly identified 38 issues in non-synthetic configurations and flagged all retroactively injected bugs with high confidence, resulting in a recall of **100%**. The only two false positives were associated with uncommon yet plausible values for `RRH_DST_MAC_ADDR`, the MAC address of the destination vRAN server, due to Diffy's lack of knowledge of the MAC address that the user intended to allocate to a vRAN server.

Example bugs. Diffy successfully identified the misconfigured RU frequency with the synthesized template `"3"` · `[num]` and value `"3929700"` kHz. This was outside the range of typical values, which all fall into the n78 band. The template is shown in Figure 9. In another case, Diffy flagged a field called `"RRH_TX_ATTENUATION"`, which indicates the transmit power attenuation before the RU's power amplifier. In one configuration, this field has a value of `"[20.0, 20.0, 20.0, 20.0]"`, whereas the typical value is `"[30.0, 30.0, 30.0, 30.0]"`. As a result, 4 anomalies were flagged by Diffy, one for each element. It is likely the configuration was created by a testbed user who experimented with power attenuation, but this deviation should be flagged to the testbed operator.

6.3 Diffy Performance

In this section, We expand on Diffy's performance along several dimensions.

String synthesis performance. We evaluated Diffy's string synthesis algorithm from §3 by comparing it to FlashProfile [45], a leading data profiling tool that also learns regular expression

patterns to represent string sets. We obtained the set of all calls in Diffy to synthesize string patterns for each of the WAN roles R1–R5 in Table 5. We replay this set of calls in both Diffy and in FlashProfile and measure the total time to synthesize string patterns. We measure the performance difference for prefix lists (first column), route policies (second column), and other elements (third column), and show the corresponding speedup (FlashProfile time / Diffy time).

The performance comparison results is shown in Table 5. Diffy takes anywhere from under 1 second, to 86 seconds to synthesize the string patterns. In contrast, FlashProfile takes between 10 seconds and 13.6 hours to synthesize similar patterns. This corresponds to a roughly 2–3 orders of magnitude speedup for Diffy. One of the main reasons for these differences is likely due to the different approaches taken by the tools. FlashProfile has its own notion of pattern cost, but relies on heavyweight program synthesis to enumerate regular expression candidate patterns [45, 46]. In contrast, Diffy takes the cost into account during synthesis as part of its dynamic programming algorithm. By using FlashProfile in Diffy, for instance, the time to template the route policies in role R1 would increase from roughly 20 min to over 13 days.

We also manually compared the quality of patterns generated by Diffy and FlashProfile using FlashProfile's publicly available benchmark dataset [44]. We found that Diffy produced similar sets of patterns as FlashProfile if given the same regular expression tokens, costs, and an appropriate grouping threshold cutoff. Moreover, slight variations in the patterns often do not impact anomaly detection much (see §6.4).

Performance trends. We have seen that Diffy can easily scale to the entire WAN. However, to get a better sense of how the performance varies with the number of configurations, we additionally ran Diffy on smaller subsets of the full set of configurations of the largest role R1 and observe its relative performance. In Figure 7 we show the fraction of the runtime for the subset of configurations compared to that of the full set of configurations relative to the ratio of configurations analyzed. This lets us see the general scaling trend, which appears to be just slightly more than linear.

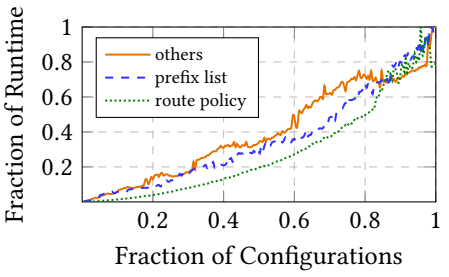


Fig. 7. Diffy performance vs. the fraction of WAN configurations of R1 being analyzed.

To evaluate the performance of Diffy in relation to the number of tokens, we executed Diffy on WAN configurations, altering the number of tokens between 1 and 10. Due to limitations in space, the selected tokens and the particulars of the experiment are elaborated in Appendix D. Our findings (Figure 10) reveal that Diffy demonstrates linear scalability with the number of tokens.

6.4 Diffy Hyperparameter Sensitivity

To evaluate the sensitivity of the grouping threshold used to create choice templates, we vary the threshold from 0.02 to 0.98 and plot the fraction of the maximum possible anomalies reported at each threshold using an anomaly score threshold of 0.8 or higher in Figure 8 for the WAN, RAN, and MySQL configurations (see Appendix C). The plot shows the results for each of the configuration element types in the WAN for all the roles summed together.

We find that beyond a certain point (*e.g.*, 0.2) the particular value of the threshold does not affect the anomalies reported greatly. There may be several reasons for this phenomenon. For the WAN, many of the anomalies are due to missing policies (see Table 4), which remains largely unaffected by the particular groupings of patterns. In other cases, however, the anomaly detection approach

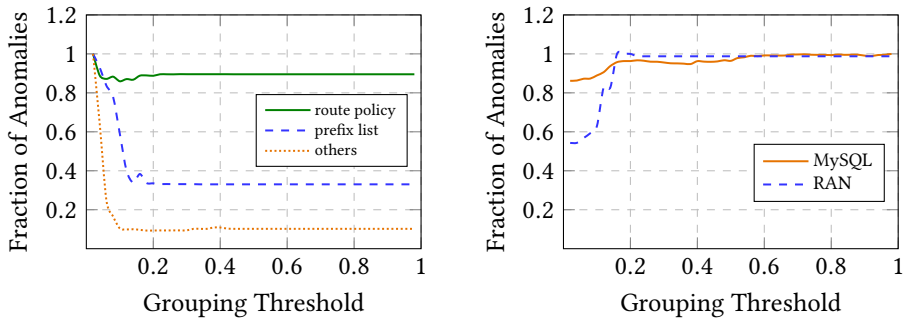


Fig. 8. Fraction of anomalies vs. grouping threshold for the WAN, RAN, and MySQL configurations.

described in §4 is fairly robust to the groupings used. For instance, Diffy may detect an anomaly in the value of a string parameter (*e.g.*, `value`) or due to separate patterns (*e.g.*, `choice`).

6.5 Diffy Generality

Diffy offers extensive coverage of configurations due to its support of complex structure for formats such as JSON, XML, and YAML as well as its ability to learn domain-specific data formats from examples. In Table 6, we showcase this generality by comparing a sample of WAN router configuration element types against existing data-driven configuration analysis tools.

Table 6. Representative subset of Router configuration elements and their support in existing tools.

Element	Diffy	SelfStarter[31]	Others[51, 61]
ACLs	●	●	○
AS Path Lists	●	○	○
Community Lists	●	○	○
DNS Servers	●	○	○
Prefix Lists	●	●	○
Route Policies	●	●	○
Route Process	●	○	●
Static Routes	●	○	○
VRFs	●	○	●

SelfStarter [31] is built specifically to analyze router configurations and supports structured policies like prefix lists and route policies. However, it requires hand-tuning and specific implementation for each element type that one wants to analyze and therefore cannot support many conceptually similar and widely used configuration elements like BGP community lists without nontrivial modification. Even in the domain of router configurations, Diffy analyzes significantly more configuration features than SelfStarter. Meanwhile, other tools [51, 52] find simple kinds of issues for basic strings but fail to analyze complex structured configurations.

7 RELATED WORK

Diffy is related to several lines of prior work:

Configuration anomaly detection. Previous research extensively explored identifying configuration errors using data-driven learning, such as ConfigC [52], ConfigV [51], and Minerals [36]. These approaches view configurations as bags of key-value pairs, learning rules like type or arithmetic rules. EnCore [61] learns configuration invariants while considering the system environment to minimize false positives (*e.g.*, by verifying file paths). However, unlike Diffy, these methods do not apply to hierarchically structured configurations like JSON and do not handle ad hoc data formats, limiting their applicability.

SelfStarter [31] is a closely related work, uniquely capable of analyzing some table structures in configurations. It identifies errors in ACLs, prefix lists, and route policies, but needs domain-specific semantic knowledge (*e.g.*, about IP address octets and ACL reordering). In contrast, Diffy generalizes SelfStarter's approach to manage general JSON structures and ad hoc data formats.

Additionally, as discussed in Table 6, Diffy supports a broader range of templates, such as string, repeat, and choices templates (refer to Figure 1).

Synthesizing data formats. Learning concise patterns from a given set of strings (§3.4) has been explored in prior work [1, 16, 45], including FlashProfile [45] and PADs [16]. FlashProfile relies on expensive program synthesis techniques that involve enumerating all regular expressions matching a set of examples [46], then utilizing cost metrics and clustering methods to choose the most appropriate patterns. However, exhaustive enumeration renders FlashProfile impractical for frequent use in template synthesis as shown in Table 5. In comparison, Diffy employs a dynamic programming approach and various optimizations to efficiently synthesize string templates.

PADs attempts to solve the ambitious problem of learning complex data formats directly from text using a top down statistical approach. Rather than learning from raw text, Diffy instead directly leverages the hierarchical structure that exists in JSON and optimally solves the string synthesis sub-problem using dynamic programming. PADs also uses regular expression tokens but relies on a pre-tokenizing step for input strings that may hinder anomaly detection (*e.g.*, all integers are converted to `[num]` prior to analysis). Diffy lazily tokenizes strings based on the specific configuration values. These two approaches vary greatly in their details, though it may be possible to incorporate ideas from PADs and Diffy.

Network verification. Verification enables users to formally prove the correctness of their configurations by precisely modeling a system and allowing proactive analysis of configuration changes. By comparing behavior to user-provided specifications, verifiers identify bugs before deployment. Verification has been successful in specialized domains like data plane forwarding [3, 26, 28, 34, 35, 39, 40, 58, 62], routing [4–6, 15, 20, 54], and DNS [30]. However, it requires (1) rigorous system modeling, which can be time-consuming or infeasible for many systems, and (2) user specifications, which may be challenging to obtain. Diffy requires no specifications and no system modeling but also provides no guarantees of correctness.

Log file anomaly detection. There is extensive work related to log file anomaly detection [11, 19, 24, 25] that attempts to predict system failures based on anomalous log file entries. These tools typically work by clustering log lines into templates such as `"Command Failed on: *"` and track template counts over time to detect anomalous behavior. For instance, an increase in a template that mentions the word `"Failed"` may predict a future system crash. Compared to Diffy, these tools are highly specialized to the domain of log files and neither analyze the hierarchical structure of JSON nor find bugs in template parameters that are common in configurations.

8 CONCLUSION

In this paper, we presented Diffy, the first tool that can identify likely bugs in complex configurations with ad hoc data formats. Given JSON configurations, Diffy first efficiently synthesizes a low-cost template that succinctly captures similarities as well as differences across these configurations. This template approach leverages a novel dynamic programming algorithm to efficiently learn a set of patterns summarizing string data, and Diffy then lifts this algorithm recursively to synthesize templates for structured data. Finally, by applying unsupervised learning, Diffy is able to identify likely configuration errors as deviating from the normal behavior. Evaluating Diffy on a variety of network configurations, we demonstrated its versatility, scalability, and accuracy.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback. We would also like to thank David Walker, Todd Millstein, and Victor Bahl for providing feedback on an early draft of the paper. Finally, we would like to thank Karthick Jayaraman and Mark Kasten for their helpful feedback helping in the analysis of the WAN configurations.

AVAILABILITY

Diffy is publicly available on Github [32] as well as Zenodo [33]. These sites contain the code to run Diffy as well as the MySQL and RAN configuration datasets used in the paper. The WAN configuration dataset is confidential and not included as part of the release.

A EXAMPLE: OPERATIONAL 5G TESTBED

We show another example of Diffy in use for a simplified and anonymized configuration snipped from an operational 5G testbed. Shown in Figure 9, the configuration is for a set of radio units (RUs), each with around 30 different configuration parameters. These parameters control different aspects of the RU such as the grandmaster PTP IP address, the frequency, the attenuation, and so on. Some of the configuration parameters such as `"RRH_DST_MAC_ADDR"` consist of hex values. An example template from Diffy is shown on the right, and the generated anomalous expressions are shown in the bottom left. As an example, Diffy has identified that the frequency is potentially incorrectly configured for two of the configurations and that the MAC address is potentially wrong for another two.

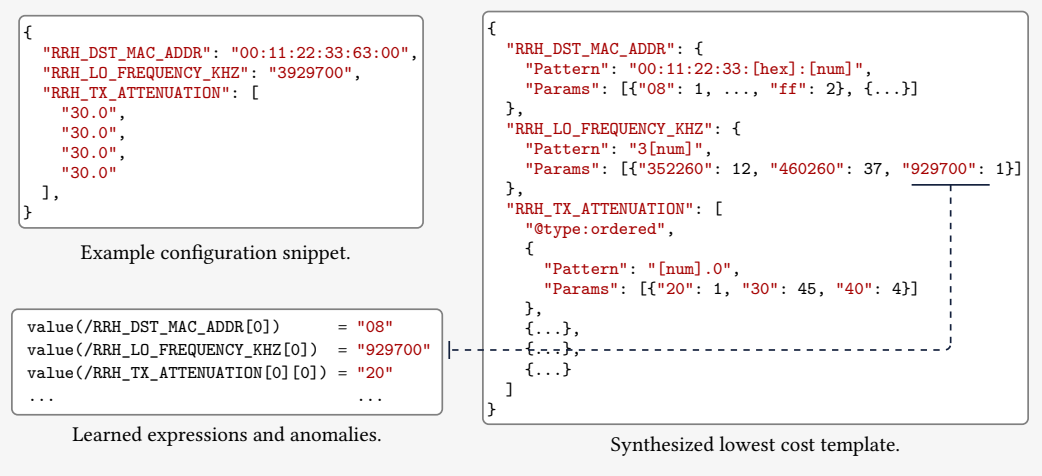


Fig. 9. Example of Diffy for anonymized and simplified configurations from a 5G virtual radio access network.

B AUTOMATA INDEX TRACKING

In this section, we expand on the algorithm Diffy uses to efficiently compute the set of matching indices for regular expression tokens against another sequence of characters and tokens. Recall the example from Figure 4, where the input sequence is `"1"` · `[num]` · `"M"` and the goal is to find all substrings such that the `[num]` token necessarily matches that substring. For instance, `"1"` necessarily matches `[num]`, as does `"1"` · `[num]` as well as `[num]` by itself. However, `"1"` · `[num]` · `"M"`, `[num]` · `"M"`, and `"M"` are not necessarily numbers.

B.1 Tracking Automata States

Recall that we take the sequence `"1"` · `[num]` · `"M"` and start by mapping each starting index to a set of possible states. Initially, there is only a single possible starting index 1, we start in the initial state for the automata for `[num]`, result in $\{q_0\} \mapsto \{1\}$. Now to find what states we may be in after consuming the first element of the sequence (`"1"`), we simply apply the automaton transition for

"1" for each state, resulting in {q_1} ↦ {1}, which is then stored in the index table. Since q_1 is an accepting state, this means that the substring from 1 to 1 (*i.e.*, "1") matches token [num].

Continuing the process, we now add {q_0} ↦ {2} alongside the existing {q_1} ↦ {1} to the index table to represent matches starting from the second index as well. At this point, we come across a [num] token in the template sequence. To determine which states we can reach by consuming a string that also matches this token, we compute the cross product of the token automata with the automata of the [num] token. The starting states are (q_0, q_1) for the old entry ({q_1} ↦ {1}) since we have already consumed "1". We need to find all states that would be final in the [num] automaton corresponding to a matching string. We find that starting either at index 1 or 2 (same process repeated but with (q_0, q_0) for entry {q_0} ↦ {2}) necessarily results in being in state q_1 only, which means we have a valid regex match. As a result, the index table for the second entry is updated to {q_1} ↦ {1, 2}.

Finally, after the last template sequence value "M", the automaton will be be in state q_2. Once again, we add the entry {q_0} ↦ {3} to the index table before applying the "M" to update the automaton states. For the entry {q_1} ↦ {1,2}, reading "M" results in being in state q_2. Similarly, for the new entry {q_0} ↦ {3}, we end up in state q_2. Thus, the index table is updated to {q_2} ↦ {1,2,3}. This is not a final state, so there is no match for [num] token ending at "M".

Note that, with this approach, we only do work at each step proportional to the number of unique sets of automaton states that are reached, which tends to be just a small constant number. Diffy also caches the results of the product construction and thus, scales nearly linearly with the size of the input strings in practice. The complete algorithm for computing the indices $\mathcal{I}$ is shown in Algorithm 1.

B.2 Fast Algorithm for Token Matching

The FindMatchingStartIndicies (Line 1) takes a regex r and a string template τ as input. The string template ($\tau = \tau_1 \cdot \ \ldots \ \cdot \tau_k$) is assumed to be a concatenation of single character const strings and tokens from $\mathcal{R}$. The procedure returns for each index ($1 \leq i \leq k$), the set of smaller start indices from which the token r matches the sub-template ($\mathcal{I}[i] = \{j \mid j \leq i \wedge \textsc{TokenMatch}(r,\ \tau_j \cdot \ \ldots \ \cdot \tau_i)\}$). As described below, the procedure computes these indices using a single linear pass by tracking automaton states. For simplicity, we earlier described $\mathcal{I}$ to be returning the last set of indices, $\mathcal{I}[k]$, instead of start indices for each i. However, in practice, we first compute the starting indices for each ending index for the entire template $\tau_1 \cdot \ \ldots \ \cdot \tau_k$ and cache the result. We use this cache during our dynamic programming step to simply return $\mathcal{I}[i]$, when $\mathcal{I}(r, \tau_1 \cdot \ \ldots \ \cdot \tau_i)$ is called.

The procedure first gets a finite state machine ($\mathcal{A}_r$) from the regular expression input of the token r using a standard GetFSM (Line 2) procedure. We take the alphabet (Σ) of an automaton to be ASCII {char(0), char(1), ... ,char(256)} for simplicity, though the actual implementation of Diffy uses Unicode. As we consider FSMs, each state in the FSM will have a transition from each symbol in the alphabet. For simplicity, we have shown that the transitions are stored on a per symbol, but the actual implementation stores character ranges as shown in ① in Figure 4.

The matching start indices are maintained using a list of k sets, $\mathcal{I}$ (Line 3) and T (Line 3) is the index table described earlier, which is a map from the set of automaton states to the set of start indices. The procedure consumes one element τ_i at a time and updates $\mathcal{I}$ and T (Lines 4 to 9). After the end of the i^{th} iteration, an element $\langle S,\ V\rangle \in T$ means that the sub-template from any start indices in V to i^{th} index will lead the token automaton to end up in one of the S states. If all the elements until i are const strings, then S will be a singleton since there is always a deterministic path for a const string. However, if some of the elements were tokens, then depending on the token and the transitions in the current automaton $\mathcal{A}_r$, we could have a set of states.

Line 5 accounts for the case where if we were to start at the i^{th} index and finish at i^{th} index, then we would be in the automaton start state before consuming τ_i. The procedure calls out UpdateIndexTable procedure (Line 6) to consume τ_i and update the index table. The UpdateIndexTable procedure returns a new index table T', where for each entry $\langle S,\ V\rangle \in T$, there is an entry $\langle S',\ V\rangle \in T'$ which is the result of making a transition on τ_i from each state in S (Line 14).

The MoveStates procedure (Line 17) is split into two cases based on the input τ_i. If τ_i is a character, then for each state in the input, we get the next set of states by looking up the transitions in $\mathcal{A}_r$ (Lines 37 to 38). If τ_i is a token, we need to find all states that could happen from the input set of states if we consider a string accepted by τ_i. We simultaneously make moves in both the automata to calculate the new states. In the $\mathcal{A}_i$, we begin in the start state, and in $\mathcal{A}_r$ we begin in the input set of states (Line 23). We, then, make a transition in both automata for each symbol in the alphabet c. We can think of this as if the input to the automata are single character strings (Line 26), and we want to find out what states that could lead to. If in $\mathcal{A}_i$, that leads to an accepting state, then the single character string is accepted by τ_i and we have the set of new states that could happen in $\mathcal{A}_r$ (Lines 31 to 32). To simulate the two character strings, we use the state combination from the single character string and continue from there (Lines 33 to 35). We continue this process until all the combinations are processed, and this process is guaranteed to terminate as each automaton has a finite number of states.

C CASE STUDY 3: MYSQL

In our final case study, we evaluate Diffy on MySQL configurations, a common target for anomaly-based bug finding tools due to its simple key-value format [51, 52, 61]. We compared Diffy to ConfigV [51] using the same MySQL configuration dataset as in the original paper. ConfigV learned invariants (rules) from a training set of 256 configurations and tested them on a 973-configurations test set, randomly selected from GitHub. Both datasets are publicly available online[3].

ConfigV is designed to detect various errors, particularly those related to MySQL. It identifies Type errors, which infer the expected type (string, bool, int, path, size (integer followed by 'K', 'M', 'G')) of each keyword from the training set and flag discrepancies in the test set. As these errors are broadly applicable beyond MySQL, we focus on them. ConfigV employs association rule learning to create Type rules with specified support and confidence thresholds. Only rules with a minimum of 2% support and 71% confidence are considered.

Upon executing ConfigV, we found that it lacks a confidence threshold and has an absolute support threshold. We set the support to 5 (equivalent to 2% of the 256 training configurations) and eliminated inferred rules below 71% confidence. While the ConfigV paper claims 94 Type rules, we got 101. Testing ConfigV yielded 3200 errors, far more than the reported 345. Many were erroneous (flagged keywords were int but marked as not int), potentially due to a bug in the tool. We instead created a script to infer each flagged keyword's correct type using the paper's type inference rules and compared it to the expected type from the learned rules. After this adjustment, we found 229 violations. A significant portion of the remaining violations were false positives, either due to considering only Unix path syntax (path) and not Windows, or resulting from placeholder variables such as `{{MYSQL_PORT}}` or `"@mysql-pid"` [41]. After excluding these instances, 168 violations remained.

[3] https://github.com/ConfigV/ConfigV

Algorithm 1: Calculate Matching Indices I

Input: Token r, and string template $\tau = \tau_1 \cdot \ \ldots \ \cdot \tau_k$
Output: Indices (i, j) where r matches $(\tau_i \cdot \ \ldots \ \cdot \tau_j)$

Procedure *FindMatchingStartIndices*(r, τ)
  $\mathcal{A}_r \leftarrow$ GetFSM(r)
  $T \leftarrow \emptyset,\ \mathcal{I} \leftarrow [\emptyset_1, \emptyset_2, \cdots, \emptyset_k]$
  for i = 1 ***to*** k **do**
    $T[\{\mathcal{A}_\text{r}.\text{startState}\}] \leftarrow T[\{\mathcal{A}_\text{r}.\text{startState}\}] \cup \{\text{i}\}$
    $T \leftarrow$ UpdateIndexTable$(\mathcal{A}_r,\ \tau_\text{i},\ T)$
    for $\langle S, V\rangle$ ***in*** T **do**
      if $S \subseteq \mathcal{A}_r$.finalStates **then**
        $\mathcal{I}[i] \leftarrow \mathcal{I}[i] \cup V$
  return $\mathcal{I}$
Procedure *UpdateIndexTable*$(\mathcal{A}_r, \tau_i, T)$
  $T' \leftarrow \emptyset$
  for $\langle S, V\rangle$ ***in*** T **do**
    newStates $\leftarrow$ MoveStates$(\mathcal{A}_r,\ \tau_i,\ S)$
    T'[newStates] $\leftarrow T'$[newStates] $\cup\ V$
  return T'
Procedure *MoveStates*$(\mathcal{A}_r, \tau_i,$ states$)$
  states$' \leftarrow \emptyset$
  if $\tau_i \in \mathcal{R}$ **then**
    $\mathcal{A}_i \leftarrow$ GetFSM(τ_i)
    seen $\leftarrow \emptyset,$ queue $\leftarrow$ [ ]
    seen $\leftarrow$ seen $\cup\ \{\langle \mathcal{A}_i.\text{startState},\ \text{states}\rangle\}$
    queue.enqueue$(\langle \mathcal{A}_i.\text{startState},\ \text{states}\rangle)$
    while queue **do**
      $\langle$s, S$\rangle \leftarrow$ queue.Dequeue()
      for c ***in*** Σ **do**
        S$' \leftarrow \emptyset$
        for s_r ***in*** S **do**
          S$' \leftarrow$ S$' \cup \{\mathcal{A}_r.\text{trans}[s_r, c]\}$
        s$' \leftarrow \mathcal{A}_i.\text{trans}[s, c]$
        if s$' \in \mathcal{A}_i$.finalStates **then**
          states$' \leftarrow$ states$' \cup$ S$'$
        if $\langle$s$',$ S$'\rangle \notin$ seen **then**
          seen $\leftarrow$ seen $\cup\ \{\langle$s$',$ S$'\rangle\}$
          queue.enqueue$(\langle$s$',$ S$'\rangle)$
  else
    for s ***in*** states **do**
      states$' \leftarrow$ states$' \cup\ \{\mathcal{A}_r.\text{trans}[s, \tau_i]\}$
  return states$'$

Table 7. Regular expression tokens costs used to assess Diffy's performance for different token subsets.

Id	Token			Cost
1	`[path]`	=	`".*/.*"`	0.25
2	`[num]`	=	`"[0-9]+"`	0.35
3	`[float]`	=	`"[0-9]+\.[0-9]+"`	0.30
4	`[alpha]`	=	`"[a-zA-Z]+"`	0.30
5	`[lower]`	=	`"[a-z]+"`	0.20
6	`[upper]`	=	`"[A-Z]+"`	0.20
7	`[ip]`	=	`"[0-9]+\.[0-9]+\.[0-9]+\.[0-9]"`	0.25
8	`[prefix]`	=	`"[0-9]+\.[0-9]+\.[0-9]+\.[0-9]/[0-9]+"`	0.20
9	`[hex]`	=	`"[a-fA-F0-9]+"`	0.40
10	`[community]`	=	`"[0-9]+:[0-9]+"`	0.20

Each violation consists of the test configuration file name, the keyword, its expected type and the actual value from the configuration. We grouped the 168 violations by their three-tuple of keyword, expected type, and value, as each violation with the same three-tuple has the same underlying reason. This allowed us to identify 54 unique kinds of violations.

Because Diffy lacks distinct training and testing phases, we evaluated ConfigV results by running Diffy on the combined configuration sets (training and testing), using a minimum anomaly score threshold of 0.7, and filtering for anomalies specific to the testing set while excluding placeholder variables. We then cross-referenced ConfigV-reported issues with Diffy-detected anomalies to identify similarities. Of the **54** error cases that ConfigV reported, Diffy flagged **40** as anomalies and did not report the other **14**.

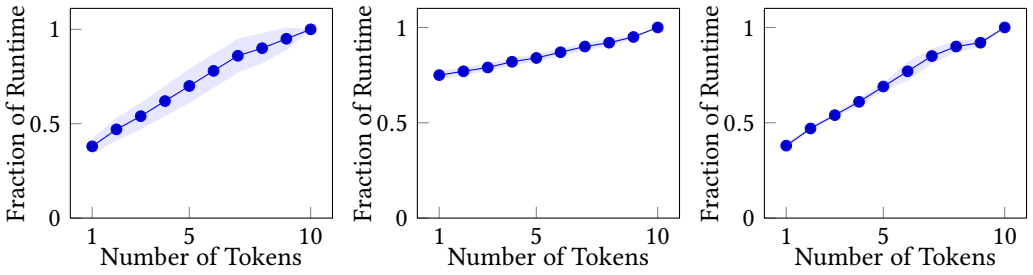
Of the 14 unflagged cases, several involved ConfigV detecting a type error when a string value was expected, but an empty string was provided. Diffy did not report these cases as empty strings were common values for the keys, likely representing default behaviors. In another instance, ConfigV inferred a size type for the `net-buffer-length` field, reporting a violation when an integer was given. Diffy also reported these cases but with lower anomaly scores (*e.g.*, 0.65). ConfigV flagged multiple configurations with keyword `binlog-cache-size` due to inferring as size type from the training set. However, most testing set values were integers, resulting in false error reports. This could be attributed to separate testing and training phases. Diffy, recognizing the flipped case, did not report these issues. Finally, in one case Diffy learned a sub-optimal pattern due to the order in which it processed the configurations and thus, missed a valid issue.

Interestingly, without knowledge about MySQL types, Diffy found similar cases. For example, Diffy identified fields with patterns like `[num]` · `[any]` from Figure 4, where `[num]` captures the value and `[any]` captures units (*e.g.*, `"M"`, `"G"`, `"K"`). If users used invalid units like `"KB"`, Diffy detected these cases due to their rarity. In another instance, ConfigV identified a type issue where a file path was expected, but the user provided `"OFF"`. ConfigV had built-in domain knowledge, recognizing `"OFF"` as a boolean value and flagging it as an issue. Diffy also found this issue since it learned the template `([path]|"OFF")`, with the latter appearing once and the former every other time.

Overall, Diffy reported many of the same violations as ConfigV but without requiring any domain-specific tuning and engineering. For the 14 cases Diffy did not report, many appear to be either false positives or of unclear value without further investigation.

D PERFORMANCE OF DIFFY WITH NUMBER OF TOKENS

To accurately assess Diffy's efficiency with varying regular expression token combinations, we utilized 10 specific tokens, listed in Table 7, running Diffy on WAN configurations using different subsets of these tokens. We picked these tokens as it has a combination of domain-specific tokens



(a) Prefix list (b) Route policy (c) Others

Fig. 10. DIFFY performance (fraction of runtime) vs. the number of tokens for the WAN dataset. For each size of tokens, we run DIFFY with all possible subset combinations, and show the mean and standard deviation.

and generic tokens. As mentioned earlier, we assigned the cost based on the sparsity of the regular expression. Tokens with regular expressions that match fewer strings like `[ip]` are given lower cost compared to `[num]`, as they are more specific. We processed DIFFY with all possible token combinations for each size, and subsequently calculated the mean and standard deviation for the results obtained. To illustrate, when considering the number of tokens as 2, there are 45 possible combinations $\binom{10}{2}$ from the 10 tokens, necessitating 45 separate runs of DIFFY, each with a unique input combination. When the number of tokens is 10, there is only one combination, and we take standard deviation as zero. For every combination, we run DIFFY on each role and each element type, such as prefix list and route policy. We then aggregated the time DIFFY consumed across all roles for every distinct element type. The results depicting DIFFY 's performance, in terms of fraction of runtime in relation to the increase in the number of tokens, is illustrated in Figure 10. In route policies and others, the standard deviation is around 0.02, which is why the shaded part is not big. From the plots, it is evident that as we increase the number of tokens, we see a relatively linear increase in the runtime.

REFERENCES

[1] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2015. Profiling relational data: a survey. *The VLDB Journal* 24 (2015), 557–581.

[2] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. 2018. Semantic-based automated reasoning for AWS access policies using SMT. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, Austin, Texas, USA, 1–9.

[3] Ryan Beckett and Aarti Gupta. 2022. Katra: Realtime Verification for Multilayer Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 617–634. https://www.usenix.org/conference/nsdi22/presentation/beckett

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. ACM, New York, NY, USA, 155–168. https://doi.org/10.1145/3098822.3098834

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2018. Control Plane Compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 476–489. https://doi.org/10.1145/3230543.3230583

[6] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2019. Abstract Interpretation of Distributed Network Control Planes. *Proc. ACM Program. Lang.* 4, POPL, Article 42 (dec 2019), 27 pages. https://doi.org/10.1145/3371110

[7] Tim Bray. 2017. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259. https://doi.org/10.17487/RFC8259

[8] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. 2020. Understanding and discovering software configuration dependencies in cloud and datacenter systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 362–374. https://doi.org/10.1145/

3368089.3409727
[9] Cisco. 2023. Basic Router Configuration. https://www.cisco.com/c/en/us/td/docs/routers/access/800M/software/800MSCG/routconf.html. [Online; accessed 30-March-2023].
[10] Oracle Corporation. 2023. MySQL. https://www.mysql.com/. Accessed: 2023-11-01.
[11] Min Du and Feifei Li. 2016. Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 859–864.
[12] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada) *(SOSP '01)*. Association for Computing Machinery, New York, NY, USA, 57–72. https://doi.org/10.1145/502034.502041
[13] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review* 35, 5 (2001), 57–72.
[14] Evolven. 2022. Downtime, Outages and Failures - Understanding Their True Costs. https://www.evolven.com/blog/downtime-outages-and-failures-understanding-their-true-costs.html. Accessed: March 26, 2023.
[15] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 217–232. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/fayaz
[16] Kathleen Fisher, David Walker, Kenny Q Zhu, and Peter White. 2008. From dirt to shovels: fully automatic tool generation from ad hoc data. *Acm sigplan notices* 43, 1 (2008), 421–434.
[17] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 469–483. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/fogel
[18] Cloud Native Computing Foundation. 2023. Kubernetes Documentation. https://kubernetes.io/docs/home/. Accessed: 2023-11-01.
[19] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 ninth IEEE international conference on data mining*. IEEE, 149–158.
[20] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. ACM, New York, NY, USA, 300–313. https://doi.org/10.1145/2934872.2934876
[21] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.
[22] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-Anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. 2014. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM symposium on cloud computing*. 1–14.
[23] Songqiao Han, Xiyang Hu, Hailiang Huang, Mingqi Jiang, and Yue Zhao. 2022. ADBench: Anomaly Detection Benchmark. In *Neural Information Processing Systems (NeurIPS)*.
[24] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE international conference on web services (ICWS)*. IEEE, 33–40.
[25] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. 2016. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*. IEEE, 207–218.
[26] Alex Horn, Ali Kheradmand, and Mukul Prasad. 2017. Delta-net: Real-time Network Verification Using Atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 735–749. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/horn-alex
[27] Istio 2023. *Istio Configuration*. Retrieved Sep 19, 2023 from https://istio.io/latest/docs/ops/configuration/
[28] Karthick Jayaraman, Nikolaj Bjorner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. 2019. Validating Datacenters at Scale. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. ACM, New York, NY, USA, 200–213. https://doi.org/10.1145/3341302.3342094
[29] Juniper Networks. 2023. CLI User Guide for Junos OS. https://www.juniper.net/documentation/us/en/software/junos/cli/index.html. Accessed: April 2, 2023.
[30] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. 2020. GRoot: Proactive Verification of DNS Configurations. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 310–328. https:

//doi.org/10.1145/3387514.3405871

[31] Siva Kesava Reddy Kakarla, Ryan Beckett, Todd Millstein, and George Varghese. 2022. SCALE: Automatically Finding RFC Compliance Bugs in DNS Nameservers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 307–323. https://www.usenix.org/conference/nsdi22/presentation/kakarla

[32] Siva Kesava Reddy Kakarla, Francis Y. Yan, and Ryan Beckett. 2024. Diffy: Data-Driven Bug Finding for Configurations. https://github.com/microsoft/DiffyConfigAnalyzer. Accessed: April 5, 2024.

[33] Siva Kesava Reddy Kakarla, Francis Y. Yan, and Ryan Beckett. 2024. Diffy: Data-Driven Bug Finding for Configurations. https://doi.org/10.5281/zenodo.10740687. Accessed: April 5, 2024.

[34] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 113–126. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian

[35] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 15–27. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/khurshid

[36] Franck Le, Sihyung Lee, Tina Wong, Hyong S. Kim, and Darrell Newcomb. 2006. Minerals: Using Data Mining to Detect Router Misconfigurations. In *Proceedings of the 2006 SIGCOMM Workshop on Mining Network Data* (Pisa, Italy) *(MineNet '06)*. Association for Computing Machinery, New York, NY, USA, 293–298. https://doi.org/10.1145/1162678.1162681

[37] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation forest. In *2008 eighth ieee international conference on data mining*. IEEE, IEEE, Pisa, Italy, 413–422.

[38] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. 2017. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 599–613.

[39] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking Beliefs in Dynamic Networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) *(NSDI'15)*. USENIX Association, USA, 499–512.

[40] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. *SIGCOMM Comput. Commun. Rev.* 41, 4 (aug 2011), 290–301. https://doi.org/10.1145/2043164.2018470

[41] Qbolec Mark Santolucito. 2017. Potential Misconfiguration. https://github.com/qbolec/ants/issues/1. Accessed: April 17, 2023.

[42] Nextgov. 2021. Commercial Cloud Outages Are a Wake-Up Call. https://www.nextgov.com/ideas/2021/03/commercial-cloud-outages-are-wake-call/172731/. Accessed: 2023-11-01.

[43] David Oppenheimer, Archana Ganapathi, and David A Patterson. 2003. Why do Internet services fail, and what can be done about it?. In *4th Usenix Symposium on Internet Technologies and Systems (USITS 03)*.

[44] Saswat Padhi. 2018. FlashProfileDemo: A C# application that demonstrates the capabilities of FlashProfile. https://github.com/SaswatPadhi/FlashProfileDemo/tree/master/tests. Accessed: March 25, 2023.

[45] Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd D. Millstein. 2018. FlashProfile: A Framework for Synthesizing Data Profiles. *PACMPL* 2, OOPSLA (2018), 150:1–150:28. https://doi.org/10.1145/3276520

[46] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. *SIGPLAN Not.* 50, 10 (oct 2015), 107–126. https://doi.org/10.1145/2858965.2814310

[47] Raymond Pompon. 2021. BGP, DNS, and the fragility of our critical systems. https://www.f5.com/labs/articles/cisotoc iso/bgp-dns-and-the-fragility-of-our-critical-systems. Accessed: 2023-11-01.

[48] Ariel Rabkin and Randy Howard Katz. 2012. How hadoop clusters break. *IEEE software* 30, 4 (2012), 88–94.

[49] Teri Radichel. 2023. About the 5-hour Microsoft Outage. https://medium.com/cloud-security/about-the-5-hour-microsoft-outage-18d47543769d. Accessed: 2023-11-01.

[50] Yakov Rekhter, Susan Hares, and Tony Li. 2006. A Border Gateway Protocol 4 (BGP-4). RFC 4271. https://doi.org/10.17487/RFC4271

[51] Mark Santolucito, Ennan Zhai, Rahul Dhodapkar, Aaron Shim, and Ruzica Piskac. 2017. Synthesizing configuration file specifications with association rule learning. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–20.

[52] Mark Santolucito, Ennan Zhai, and Ruzica Piskac. 2016. Probabilistic automated language learning for configuration files. In *Computer Aided Verification: 28th International Conference, CAV 2016, Proceedings, Part II 28*. Springer, Springer, Cham, Toronto, ON, Canada, 80–87.

[53] Temple F Smith and Michael S Waterman. 1981. Identification of common molecular subsequences. *Journal of molecular biology* 147, 1 (1981), 195–197.

[54] Alan Tang, Siva Kesava Reddy Kakarla, Ryan Beckett, Ennan Zhai, Matt Brown, Todd Millstein, Yuval Tamir, and George Varghese. 2021. Campion: Debugging Router Configuration Differences. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 748–761. https://doi.org/10.1145/3452296.3472925

[55] Liam Tung. 2019. Azure global outage: Our DNS update mangled domain records, says Microsoft. https://www.zdnet.com/article/azure-global-outage-our-dns-update-mangled-domain-records-says-microsoft/. Accessed: 2023-11-01.

[56] Kurt Wise. 2017. High Number of AWS Misconfigurations Leaves Huge Security Holes. https://virtualizationreview.com/articles/2017/04/19/aws-misconfigurations-leaves-huge-security-holes.aspx.

[57] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 244–259.

[58] Hongkun Yang and Simon S. Lam. 2016. Real-time Verification of Network Properties Using Atomic Predicates. *IEEE/ACM Trans. Netw.* 24, 2 (April 2016), 887–900. https://doi.org/10.1109/TNET.2015.2398197

[59] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. 2011. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 159–172.

[60] Iris Zarecki. 2019. 19 of the worst IT outages in 2019 – A Recap of Being Let Down. https://www.continuitysoftware.com/blog/19-of-the-worst-it-outages-in-2019-a-recap-of-being-let-down/.

[61] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 687–700. https://doi.org/10.1145/2541940.2541983

[62] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. 2020. APKeep: Realtime Verification for Real Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 241–255. https://www.usenix.org/conference/nsdi20/presentation/zhang-peng

Received 2023-11-16; accepted 2024-03-31