ConfExp: Root-Cause Analysis of Service Misconfigurations in Enterprise Systems

NEGAR MOHAMMADI KOUSHKI, Computer and Information Sciences, Temple University, USA
IBRAHIM EL-SHEKEIL, Computer Science and Cybersecurity, Metro State University, USA
KRISHNA KANT, Computer and Information Sciences, Temple University, USA

Misconfiguration is a known and increasingly serious problem in enterprise systems due to frequent code updates and retuning of the configuration parameters. Diagnosing complex, residual misconfiguration problems that lead to inaccessible services or failed transactions often starts with either a user complaint or observation by administrators, followed by a largely manual process of deciding what tests to run and how to proceed with further testing based on the test results. The goal of this paper is to automate this process and thereby make root-cause analysis of accessibility related misconfigurations much speedier and much more effective. We explore an extensible domain-knowledge-driven methodology, called ConfExp using a network emulator that runs real enterprise networking protocols. Thus, by using commonly used tests, we show that the root-cause can be determined in all cases where discriminative tests exist. The methodology also highlights areas where more discriminative tests are needed to pinpoint the precise configuration variables at fault.

CCS Concepts: • **Networks** → **Network services**; **Network performance evaluation**.

ACM Reference Format:
Negar Mohammadi Koushki, Ibrahim El-Shekeil, and Krishna Kant. 2024. ConfExp: Root-Cause Analysis of Service Misconfigurations in Enterprise Systems. *J. ACM* 000, 000, Article 000 (2024), 21 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Configuration Variables (CVs), or *run-time parameters*, control and adjust software and hardware system functionalities, providing flexibility to meet varying provider and user requirements without code changes. Nearly every hardware and software module has numerous CVs, yet understanding their settings, interactions, and failure impacts often eludes service creators and administrators. This problem is exacerbated by outdated or incomplete documentation of CVs and their interactions.

Misconfigurations have become a common source of operational issues and are routinely exploited by attackers to gain access or disrupt systems. As reliance on IT services grows, ensuring continuous operation becomes essential but challenging due to complex dependencies and numerous configuration parameters. The complexity is further increased by the adoption of Service-Oriented Architecture (SOA) and microservices, which aim to reduce dependencies between service packages for easier development and scalability [1]. DevOps practices, particularly Continuous

This research was supported by NSF grant CNS-2011252.
Authors' addresses: Negar Mohammadi Koushki, koushki@temple.edu, Computer and Information Sciences, Temple University, 1925 N 12th St, Philadelphia, PA, USA, 19122; Ibrahim El-Shekeil, Computer Science and Cybersecurity, Metro State University, 700 East Seventh Street, Saint Paul, MN, USA, 55106-5000, ibrahim.el-shekeil@metrostate.edu; Krishna Kant, kkant@temple.edu, Computer and Information Sciences, Temple University, 1925 N 12th St, Philadelphia, PA, USA, 19122.

Manuscript submitted to ACM

Integration/Continuous Development (CI/CD), further complicate configurations due to constant changes in code and run-time settings [2]. Lightweight containerized deployments enable flexibility, but the increase in CVs raises the risk of failures and lengthens downtime as diagnosing issues grows more complex.

Misconfigurations can vary in impact, from minor issues to severe outages affecting critical infrastructure services like DNS. Historical examples include Microsoft's 2001 DNS outage [3], Sweden's `.se` domain outage in 2009 [4], and Apple's DNS-related service outage in 2015 [5]. Unforeseen infrastructure deficiencies can also lead to unexpected failures, as seen in the 2016 Delta and Southwest Airlines outages, where backup systems failed to activate [6, 7]. While configuration management tools (e.g., Ansible, Chef) facilitate large-scale configuration updates, they also risk deploying problematic configurations across many nodes.

This paper addresses challenges with identifying root causes of misconfigurations that render services inoperational, focusing on enterprise *production* systems where non-standard tests are usually restricted. We aim to identify misconfigured CVs or sets of CVs affecting service operability. Unlike proactive diagnostics, which are impractical in large production systems, we rely on sequential testing initiated by reported issues, with each test informing subsequent actions. Our goal is to minimize testing through intelligent test selection.

Our approach leverages the SEED emulator to mirror real data center operations by running unmodified service code within containers, applying real commands like `ping` and `traceroute`. Unlike many studies relying on abstract tests, our method applies directly to live systems, providing a functional digital twin for in-depth root-cause analysis.

Existing tools like SNMP, NETCONF, and Nagios monitor system health but don't diagnose misconfigurations within CVs and their dependencies. Our approach uniquely addresses this need, as detailed in Section 6. Thus the main contributions of our work are as follows:

- A comprehensive SEED-based emulator of enterprise networks, running real open-source service code for authentic diagnostics [8].
- A CV diagnosis mechanism with a novel test-selection algorithm to identify misconfigurations affecting service operability.
- An automated process to determine optimal test sequences, minimizing tests while enhancing diagnostic precision.
- Empirical validation against expert diagnosis, showing our method often surpasses manual troubleshooting in accuracy and efficiency.
- Direct applicability of our method to real deployments, enabling real-time diagnostics without custom test modifications.

We propose ConfExp as a practical tool for enterprise network diagnostics that is compatible with real networks or SEED-based "digital twins." We plan to open-source our code to facilitate broader use and adaptation.

The rest of this paper is organized as follows. Section 2 describes our diagnostic methodology, foundational concepts, and notations. Section 3 outlines our misconfiguration diagnosis approach, detailing test selection strategies. Section 4 presents our network testbed setup using SEED. Section 5 discusses experimental results, threshold tuning, and categorization of outcomes. Section 6 reviews related work in network fault diagnosis. Section 7 concludes with our findings and future research directions.

2 PROPOSED CONFEXP DIAGNOSIS METHODOLOGY

In this section, we discuss the basics of ConfExp diagnosis methodology, starting with a discussion of CVs and faults in enterprise systems.

2.1 Configuration Faults and their Handling

An enterprise system typically runs a large number of services, which includes both the basic system services such as DNS, Dynamic Host Configuration Protocol (DHCP), routing, firewall, etc. and application services such as web-browsing, payment, Database (DB), shopping cart, etc. Each service may involve many CVs of different types. With services such as DNS, routing, firewall, Active Directory, etc. that maintain a repository of specific items (i.e., DNS entries, routing entries, firewall rules, user access privileges, etc.), the configuration includes two types of CVs: (a) *Basic CVs*, that relate to the service as a whole (e.g., IP address of a name server, Virtual Local Area Network (VLAN) setup, etc.), and (b) *Specific CVs*, that relate to individual service entries (e.g., DB entries, DNS entries, routing entries, firewall rules, etc.). The misconfiguration of basic CVs would affect the entire service, whereas the misconfiguration of specific CVs would affect only certain clients or transactions.

Configuration problems can be examined both proactively and reactively. A proactive method may run some standard tests following an update or periodically collect detailed logs for offline analysis. While this can be useful, a lack of focus for the tests or log analysis is unlikely to discover many problems. Furthermore, a proactively detected problem would still need to trigger a diagnosis procedure to root-cause the problem. The diagnosis may also be triggered by a user complaint or the administrator noticing some unexpected or unsatisfactory behavior. Although the goal of our diagnosis is finding the root-cause of the misconfiguration, this is not always possible due to the limitations of the generally available tests that we use. In such cases, the goal is simply to narrow down the problem to a specific service or group of CVs, which can be investigated further either manually or using more application specific tools.

2.2 Assumptions of the Diagnosis Framework

For simplicity, we make the following assumptions in this paper:

(1) We assume that each diagnosis iteration targets a single fault affecting one or more services. While multiple faults could exist simultaneously, their effective identification is beyond the scope of this paper.
(2) We assume that any relevant fault in a service or network component is detectable by the Local Management Consoles (LMCs), and faults not shown by the LMC are outside the scope of this approach.
(3) For network-wide awareness, we assume a virtual Global Management Console (GMC), which provides aggregated status information across subnets. The implementation of GMC and dealing with distributed systems issues such as unpredictible delays is beyond this paper's scope.
(4) We assume that the underlying fault, once present, is persistent and does not resolve or change unpredictably. This assumption simplifies the diagnosis, focusing on faults that remain consistently problematic until addressed.
(5) We assume that the LMCs operate independently within robust management networks that are unaffected by issues in the main network. This setup is typical in enterprise environments and ensures reliable fault detection at the subnet level.

2.3 Diagnosis Architecture

The complexity of diagnosing problems in IT systems arises from the intricate interdependencies among services and their CVs. Typically, the diagnosis process begins with a user-reported problem or an observation by an administrator. This initial report/observation is necessarily imprecise, and may speak of application slowdown, query failure, or node/service inaccessibility. Our diagnosis framework employs the concepts of LMCs, Test Agents (TAs), and a GMC, as illustrated in Fig. 1. To effectively narrow down the scope of the problem, we operate under the assumption that the

underlying fault is permanent, although it might only become apparent under specific conditions such as heavy loads, resource pool exhaustion, or certain types of queries.

We further assume that the network is composed of multiple subnets, potentially including VLANs. Each subnet can be thought of as a local cluster comprising a number of servers and/or clients. Within each subnet, an LMC provides real-time updates on service statuses and serves as the initial point of fault detection. Each LMC is a crucial component of the local management network for its respective subnet and functions independently of the main network's state. This setup of having an independent local management network is commonplace in enterprise environments and reflects real-world scenarios accurately. For simplicity, we assume that these management networks are highly robust and are not the focus of our diagnosis procedures.

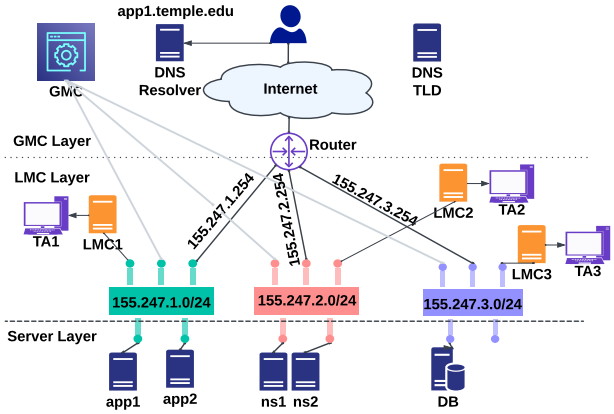


Fig. 1. Diagnosis Architecture.

TAs can be thought of as software programs running in each subnet and are responsible for initiating local tests and displaying results via the LMCs. TAs are also used to initiate non-local tests for diagnosis purposes and may additionally collect data, analyze it, and log it for future examination.

Although fault diagnosis is generally limited to within each subnet, in some cases, network-wide awareness of service status is required. To accommodate this need, we assume a virtual GMC abstraction, where status information from various LMCs can be aggregated and accessed as needed. This GMC concept allows for the emulation of a centralized fault-diagnosis mechanism, facilitating broader fault isolation across subnets. Implementing a full GMC setup introduces certain complexities, such as managing distributed data consistency and addressing any communication delays. However, these aspects are beyond the scope of this work and are suggested as future considerations.

2.4 Testing Framework Basics

Consider an enterprise system with n services denoted as $S = \{s_1, s_2, \ldots, s_n\}$. Each service s_i involves m_i CVs denoted by the set $CV_i = \{cv_{i1}, \ldots, cv_{im}\}$. We also define a severity measure π_i for the impact or criticality of service s_i. This severity index is essential for prioritizing services for testing. We assume $\pi_i \in [0..1]$ where 0 indicates no impact or low criticality, and 1 signifies maximum impact or high criticality. The assignment of severity values is based on domain knowledge, historical data, and the perceived consequences of service failures. We also define a permutation P of the service indices that arranges them in the decreasing order of severity, i.e., $\pi_{P(i)} \geq \pi_{P(j)}$ for all $j > i$.

Let $T = \{t_1, t_2, \ldots, t_l\}$ denote the complete set of tests, and let us define three functions to connect them to services and their CVs. Let $T_R(s_i)$ be the set of tests relevant to service s_i, and $CV_{t_{s_i}}$ be the set of CVs involved in the test t for service s_i. A test t is considered relevant to service s_i if and only if it involves one or more of the CVs associated with s_i (i.e., $CV_t \cap CV_i \neq \emptyset$). In other words, a test is relevant to a service if it has the potential to detect issues or validate the functionality of the CVs within that service. We also assume that each test either passes or fails. We denote this using the function Outcome(t) with values 1 (pass) and 0 (fail).

2.4.1 Defining Diagnosis Tests and Their Categories. A test in our system validates specific aspects of functionality either explicitly or implicitly through the returned response. For instance, a test ensuring communication with a remote DB implies operational network connection, routing, DNS entry, firewall rules, DB connector service, and the DB itself. However, it doesn't confirm that the network or the service is configured properly from a performance perspective. Additional tests focused on examining the CVs of various services are needed to establish the proper configuration of the CVs.

A test type comprises of tests using the same command but different inputs from various services. For example, t_i = nmap app1.temple.edu and t_j = nmap ns1.temple.edu which would have different outputs, 10.10.1.1, tcp/80 and 10.10.2.1, udp/53, respectively.

Testing of remote functionality involves three distinct steps: name resolution of the service, accessibility via the network, and service functionality itself. This motivates us to define the following hierarchy (or sequence) for testing.

Name Resolution (DNS queries): Placed first as the foundational layer of network communication, it establishes a stable reference point by addressing the reliability and consistency of DNS name resolution. Key assumptions include the uniformity of domain names across clients, DNS resolvers consistently reaching authoritative root servers, consideration of client and resolver variations, and simplified assessment through domain administrator configurations.

Reachability (ping, traceroute, etc.): After name resolution, we need to evaluate the network's ability to reach destinations, considering obstacles like firewalls and routing faults. Assumptions involve both general and specific reachability, the nuanced interpretation of failed pings, the necessity of additional tests like traceroute, and the implication of a successful ping indicating necessary route information.

Application (telnet, HTTP get, etc.): The final category examines specific network services and applications, logically following name resolution and reachability confirmation. Assumptions encompass generic tests like telnet for Transmission Control Protocol (TCP) services, the significance of responsive applications indicating operational services, non-responsive applications indicating potential faults, and the disruptive role of firewalls in application functionality.

2.4.2 Relevance of the Test. Different tests have different levels of effectiveness. In much of the work in the literature, a test is implicitly considered to be more effective if it involves more "nodes" or resources in the system. In the misconfiguration context, a test involving many nodes will likely check some overall relationship between them, rather than doing a more specific check, and is thus likely to be weaker. We attempt to capture the test effectiveness through the notion of the *relevance of a test* relative to a fault. We first extract common CVs from fault descriptions. For instance, consider the fault "Network Connection Issues in App server," which involves problems related to basic connectivity between devices on the network. This fault can result from incorrect IP address, disabled or misconfigured interface, incorrect subnet mask, incorrect port, incorrect gateway, wrong MTU settings, incorrect broadcast address, or misconfigured firewall rules. Therefore, the relevant CVs extracted for this fault are [IP address, interface, subnet mask, port, gateway, MTU, broadcast, firewall rules]. Similarly, for the fault "Firewall Blocking in App server," which pertains to traffic being blocked by firewall rules. The extracted CVs include [firewall rules, IP address, interface, port].

After extracting the CVs, we create a 2D matrix to map these CVs to specific diagnosis tests and calculate a relevance score to reflect how well a test can diagnose a given fault. We estimate the relevance score R of test T for fault F as follows:

$$R(T, F) = |\{\text{CV's present in } T\} \cap \{\text{CV's for } F\}|/|\{\text{CV's for } F\}| \quad (1)$$

However, by considering most of the commonly observed faults in F, we can obtain a measure $R(T)$ that is useful in general. Another point to note is that if the initial reporting of the problem is more specific (e.g., network vs. service problem), one could use a more granular estimate of the relevance.

For diagnosis, we obviously do not know the fault F; therefore, all we can do is to find the overall relevance of a test T for fault diagnosis. We call this as the *relevance* of the test T and denote it as $\mathcal{R}(T)$. We estimate it simply as the average over the entire fault-set F considered in the construction, i.e., $\mathcal{R}(T) = \frac{1}{|\mathbb{F}|} \sum_{F \in \mathbb{F}} R(T, F)$ Note that the actual faults that occur may not be contained in the set $\mathbb{F}$; however, by considering most of the commonly observed faults in $\mathbb{F}$, we can obtain a measure $\mathcal{R}(T)$ that is useful in general. Another point to note is that if the initial reporting of the problem is more specific (e.g., network vs. service problem), one could use a more granular estimate of the relevance.

Table 1 shows some sample tests and their relevance scores for different types faults that affect App Server accessibility. These include Network Connection Issues, Firewall and Security Blocking, and DNS Resolution Failures. The relevance scores reflect how well each test can diagnose the corresponding fault based on the common CVs identified.

Table 1. A Few Entries from Test Relevance Matrix for Application Server Faults

Test	Network Connection Failures	Firewall and Security Blocking	DNS Resolution Failures
`host <DNS record>`	0.5	0.5	1
`ping -c 5 <ip addr>`	1	1	1
`traceroute <ip addr>`	1	1	1
`ip -4 addr show dev <interface>`	1	0.25	0
`ifconfig <interface>`	1	0.25	0.2

2.4.3 *The Concept of Confidence Level for Configuration Variables.*

Since our tests are focused on root-causing the reported problem down to the specific CVs of the services, we need a measure of which CVs are likely to be misconfigured and which are likely to be fine. We do this by associating a *Confidence Level* (CL) with each CV in the range [-1,+1], where -1 means that we strongly believe that the CV is set to a value outside of its acceptable or sensible range, and +1 means the opposite (i.e., we believe that the CV is set properly and not contributing to the observed problem). The middle value of 0 indicates that we have no information about its relevance to the fault. At the start of the diagnosis, for every CV m of every service i, we initialize its CL, denoted $Q(cv_{im})$ either to 0 (if there is no prior knowledge about the relevance of this CV to the observed problem) or to some small positive/negative value (if there is prior knowledge).

The next issue is how much we change $Q(cv_{im})$ following a test, henceforth denoted as $\Delta Q_{im}(t)$. We estimate it using the relevance of the test and the number of CVs involved in the test. That is, $\Delta Q_{im}(t) = \frac{\mathcal{R}(T)}{|CV_t|}$, where $\mathcal{R}(T)$ is the test's relevance and $|CV_t|$ is the number of CVs involved in the test t. If Outcome$(t) = 1$ (test passes), all CVs in CV_i have their confidence increased, i.e., $Q(cv_{im})+ = \Delta Q_{im}(t)$. If Outcome$(t) = 0$ (test fails), all CVs in CV_i have their confidence decreased, i.e., $Q(cv_{im})- = \Delta Q_{im}(t)$.

3 PROPOSED APPROACH FOR FAULT DIAGNOSIS OF MISCONFIGURATIONS

This section outlines ConfExp for diagnosing service misconfigurations in interconnected enterprise IT systems. Our approach utilizes a systematic domain-knowledge driven algorithm for test selection. The algorithm aims to satisfy two competing goals: minimize the number of tests while maximizing the likelihood of accurately uncovering misconfigurations. In the following we will introduced some more notations and terms, all of which have been collected into Table 2 for ease of reference.

3.1 Inputs and Initialization

The algorithm starts with a problematic services list (S_p), which is a set of services flagged by the LMC as potentially misconfigured, and a Misconfiguration Threshold (MT). The MT is a threshold value for the CL of the CVs. Specifically, if the CL of a CV drops below a lower threshold (MT_l), it indicates a misconfiguration. Our experiments have determined that $MT_l = -0.6$ is optimal for balancing the number of tests and minimizing false positives. During the initialization phase, the algorithm gathers all tests relevant to the services in S_p, along with the CVs linked to these tests, establishing the necessary data structures for the subsequent diagnosis process (Lines 2-6 in Algorithm 1).

Algorithm 1: Service Fault Diagnosis

1 **Function** serviceFaultDiagnosis(S_p, MT)
2 Initialize the test hierarchy $\mathcal{H}$;
3 Initialize $\mathcal{T}_R, \mathcal{CV}_t, \mathcal{CV}, CV_m \leftarrow \emptyset$;
4 **for** *each* $s_i \in S_p$ **do**
5 $\quad \mathcal{T}_R = \mathcal{T}_R \cup T_R(s_i)$;
6 $\quad \mathcal{CV}_t = \mathcal{CV}_t \cup CV_{t_{s_i}}$;
7 $\quad \mathcal{CV} = \mathcal{CV} \cup CV_i$;
8 **for** *each level* L *in* $\mathcal{H}$ **do**
9 $\quad$ **while** $\forall Q(cv) > MT, cv \in \mathcal{CV}$ **do**
10 $\quad\quad$ Call `TestSelectionExecution`($\mathcal{T}_R, \mathcal{CV}_t, \mathcal{CV}$);
11 $\quad\quad$ **for** *each* $s_i \in S_p$ **do**
12 $\quad\quad\quad$ **if** $\nexists t \in T_R(s_i)$ ***and*** $\nexists Q(cv) < MT$ **then**
13 $\quad\quad\quad\quad$ $cv^{(s_i)}_{\text{misconfig}} = \operatorname{argmin}_{cv \in \bigcup_{t \in T_R(s_i)} CV_{t_{s_i}}} (\min(Q(cv)))$;
14 $\quad\quad\quad\quad$ $CV_m = CV_m \cup \{cv^{(s_i)}_{\text{misconfig}}\}$;
15 $\quad$ **if** CV_m *is not empty* **then**
16 $\quad\quad$ **break**;
17 **return** Set of **misconfigurations** CV_m;

3.2 Iterative Diagnosis Process

The algorithm operates in an iterative manner, continuously selecting and executing tests to identify misconfigurations. The key steps are as follows:

Test Selection. Each CV is assigned a CL indicating the likelihood it is correctly configured. These levels are initialized based on prior knowledge or set to a neutral value. The algorithm prioritizes tests expected to provide the most

Table 2. Summary of Notations and Abbreviations

Notation/Term	Definition	Notation/Term	Definition
n	Number of services	Outcome(t)	Outcome of test t
S	Set of services	F	Set of potential faults
m_i	Number of CVs for s_i	VRRP	Virtual Router Redundancy Protocol
CV_i	Set of CVs for s_i	S_p	Problematic services
π_i	Severity measure for s_i	D_{ij}	Dependency matrix
P	Descending permutation	MT	Misconfiguration threshold
T	Complete set of tests	ZC	Threshold of CVs with zero CL
$T_R(s_i)$	Tests relevant for s_i	RT	Relevance of the test threshold
CV_m	Set of misconfigured CVs	UDP	User Datagram Protocol
$CV_{t_{s_i}}$	CVs in test t for s_i	SEED	Security Education
L	Level in the test hierarchy	DB	Database
$\mathcal{H}$	Test hierarchy	BGP	Border Gateway Protocol
$\mathcal{R}(T)$	Relevance of test t	CL	Confidence Level
$\mathcal{T}_{\text{selected}}$	Set of selected tests	AS	Autonomous System
$Z(t)$	Count of CVs with CL zero in t	DNS	Domain Name System
$Q(cv_{im})$	CL associated with cv_{im}	LMC	Local Management Console
$\Delta Q_{im}(t)$	Change in CL of cv_{im} following t	GMC	Global Management Console
t_{optimal}	Optimal test selected based on criteria	TA	Test Agent
s_{picked}	Service associated with t_{optimal}	TCP	Transmission Control Protocol
CV_{picked}	Set of CVs associated with s_{picked}	VLAN	Virtual Local Area Network

informative results, based on the number of CVs with zero CLs associated with each test and the relevance of the tests, reflecting their diagnosis effectiveness (Algorithm 2).

Test Execution. The selected test is executed, and its outcome (pass or fail) is recorded. Based on the outcome, the CLs of the associated CVs are updated; a successful test increases the confidence level, whereas a failed test decreases it (Lines 7-8 in Algorithm 1).

Misconfiguration Detection. After updating the CLs, the algorithm checks if any CV's CL has fallen below the MT. If such a CV is found, it is flagged as a misconfiguration (Lines 9-12 in Algorithm 1).

Algorithm 2: Test Selection and Execution

1 **Function** TestSelectionExecution($\mathcal{T}_R$, $\mathcal{CV}_t$, $\mathcal{CV}$)
2 $Z(t) = \sum_{cv \in \mathcal{CV}_t} \delta(Q(cv) = 0), \quad \forall t \in \mathcal{T}_R$;
3 **if** $\exists t\ Z(t)$ **then**
4 $\quad \mathcal{T}_{\text{selected}} = \{t \mid \mathcal{R}(T) \geq RT$ **and** $Z(t) \geq ZC\}$;
5 $\quad$ **if** $\mathcal{T}_{selected} \neq \emptyset$ **then**
6 $\quad\quad t_{\text{optimal}} = \text{argmax}_{t \in \mathcal{T}_{\text{selected}}} \mathcal{R}(T)$;
7 $\quad$ **else**
8 $\quad\quad t_{\text{optimal}} = \text{argmax}_{t \in \mathcal{T}_R} Z(t)$;
9 **else**
10 $\quad \mathcal{T}_{\text{selected}} = \{t \mid \min_{cv_j \in \mathcal{CV}_t} Q(cv_j) = \min_{t_y \in \mathcal{T}_R} \min_{cv_l \in CV_{t_y}} Q(cv_l), t \in \mathcal{T}_R\}$;
11 $\quad t_{\text{optimal}} = \text{argmax}_{t \in \mathcal{T}_{\text{selected}}} |CV_t|$;
12 $\quad s_{\text{picked}}$ = Service associated with t_{optimal};
13 $\quad CV_{\text{picked}} = CV_{t_{\text{optimal}}}$;
14 Outcome(t_{optimal}) = Execute(t_{optimal});
15 Call `UpdateConfidenceLevels`(s_{picked}, CV_{picked}, t_{optimal});

Algorithm 3: Update Confidence Levels

1 **Function** updateConfidenceLevels(s_{picked}, CV_{picked}, t_{optimal})
2 **for** *each* $cv \in CV_{picked}$ **do**
3 $\quad$ **if** $Outcome(t_{optimal}) = 1$ **then**
4 $\quad\quad Q(cv) = Q(cv) + \Delta Q(cv, t_{\text{optimal}})$;
5 $\quad$ **else**
6 $\quad\quad Q(cv) = Q(cv) - \Delta Q(cv, t_{\text{optimal}})$;
7 $\quad$ **if** $Q(cv) < MT$ **then**
8 $\quad\quad$ **return** cv as **misconfiguration** for service s_{picked};

Iteration Termination. The iterative process continues until either all CVs have CLs above the MT or a misconfiguration is identified.

3.2.1 Key Components and Algorithms. The diagnosis process is underpinned by several core components:

- **Initialization**: Establishing the initial sets of tests and CVs (Algorithm 1).
- **Test Selection and Execution**: Choosing and executing the most informative tests based on specific criteria (Algorithm 2).
- **Updating CLs**: Adjusting CV CLs according to test outcomes (Algorithm 3).

3.3 Strategic Test Selection

The algorithm selects tests based on their relevance and the number of CVs with zero CLs (Lines 3-8 in Algorithm 2). This strategy balances identification of misconfigurations in fewest tests against the diagnosis accuracy (Lines 9-14 in Algorithm 2).

As an example, consider an enterprise system with multiple services exhibiting issues. The algorithm begins by compiling relevant tests for these services. In each iteration, it selects the most informative tests. For instance, if a

DNS-related test fails, the algorithm reduces the CL of associated DNS CVs. If any CV's CL drops below the *MT*, that CV is flagged for further investigation or correction (Algorithm 3).

4 DESIGNING THE NETWORK TESTBED FOR MISCONFIGURATION ANALYSIS

In studying misconfiguration diagnosis, it is essential to set up a reasonably sized enterprise network that supports not only the basic packet routing but also all important services, including those that are network-related (e.g., Firewall, DNS, Network Address Translation (NAT), etc.) and those that support client queries (e.g., web-service, DB service, File Transfer Protocol (FTP), etc.) Simulation is not a viable option for such a setup because of the lack of comprehensive simulators; thus, the only options are testbeds and emulators. Several testbeds are available with different strengths as surveyed in [9] – the better-known ones being EmuLab and GENI. Although either of these could, in principle, be used, setting up a complete environment using these testbeds takes quite a bit of effort. Network emulators provide similar capabilities by providing the ability to spin up Virtual Machines (VMs) or containers on a local machine to implement various functionalities. For example, open-source routing implementations such as Bird can be run in the VMs/containers to emulate routing protocols, BIND for DNS service, MariaDB for DB service, Apache for Web service, etc. Such a network will have all of the same capabilities, commands, and configuration parameters as a real implementation, and thus the diagnosis techniques can be used unaltered in real networks. Furthermore, unlike testbeds, it is possible to introduce additional network delays easily.

A well-known network emulator is CORE (Common Open Research Emulator) which we have experimented with extensively in the past [10]. It provides the capability to run popular data-center network routing protocols such as Open Shortest Path First (OSPF) and can support nodes running arbitrary code, which are set up as lightweight VMs. Unfortunately, we were unable to automate the job of running commands inside the VMs and returning results back to our control node, which is essential to conveniently manage a large network. Instead, we used a recently developed emulator called SEED, which is an open-source Python library designed to emulate the Internet for educational purposes [11]. In the following, we provide a brief description of SEED, its limitations, and the extensions that we made.

4.1 Utilizing and Extending SEED for Network Misconfiguration Studies

SEED Emulator, with its comprehensive Python classes, mirrors key components of the autonomous systems (ASes), networks, hosts, and Border Gateway Protocol (BGP) routers, along with services like Web servers, DNS, and various cybersecurity scenarios (e.g., Botnets, Darknets), enables the construction of a mini-Internet for realistic emulation. These emulations encapsulated within Docker containers, facilitate diverse cybersecurity and networking experiments. The extensibility of SEED is a notable feature, allowing for the development of new classes to simulate complex configurations, such as an Ethereum blockchain. We illustrate two prototypical networks in SEED as described next.

4.2 Case Study: Diagnosing Misconfigurations in a Compact Network Setup

This is a compact network infrastructure and includes a router, an internet connection, and multiple servers. Owned by a single entity, this setup has unrestricted visibility and testing across all resources. For clarity, our focus is on an Ethernet switch with multiple VLANs. VLANs typically segment workloads to minimize broadcast domains and enhance security. If a server within a VLAN is compromised, its impact remains contained. A router facilitates inter-VLAN routing. In our example, we assume the router solely provides routing services and public IP addresses are in use, eliminating the need for network address translation (NAT). This network is depicted in Fig. 2.

Within this network, we have two application servers, a DB server, two name servers, three VLANs, and an LMC/TA for each VLAN. External users connect to the internet and use a public DNS resolver. All servers operate under the

Table 3. IP Configuration with Highlighted Misconfigurations.

Host	IP Address	Subnet mask	Gateway
App1	155.247.1.1	**255.255.0.0**	155.247.1.254
App2	155.247.1.2	255.255.255.0	155.247.1.254
ns1	155.247.2.1	255.255.255.0	155.247.2.254
ns2	155.247.2.2	255.255.255.0	155.247.2.254
DB	155.247.3.1	**255.255.0.0**	155.247.3.254
LMC1	155.247.1.10	255.255.255.0	155.247.1.254
LMC2	155.247.2.10	255.255.255.0	155.247.2.254
LMC3	155.247.3.10	255.255.255.0	155.247.3.254

Table 4. Diagnosis Test Results for the App1 Access Misconfiguration Scenario.

Source	Dest.	Result
LMC1	App1	Fail
LMC1	App2	Pass
LMC1	ns1	Pass
LMC1	ns2	Pass
LMC1	DB	Pass
LMC3	DB	Pass

Table 5. Diagnosis Test Results for the DB Misconfiguration Scenario.

Source	Dest.	Result
LMC1	App1	Fail
LMC1	App2	Fail
LMC1	ns1	Pass
LMC1	ns2	Pass
LMC1	DB	Fail
LMC3	DB	Pass

“temple.edu" domain. When users query the DNS resolver for server names, the DNS Top Level Domain (TLD) server directs them to the network’s name servers: ns1.temple.edu and ns2.temple.edu. These servers are the authoritative name servers for the “temple.edu” domain.

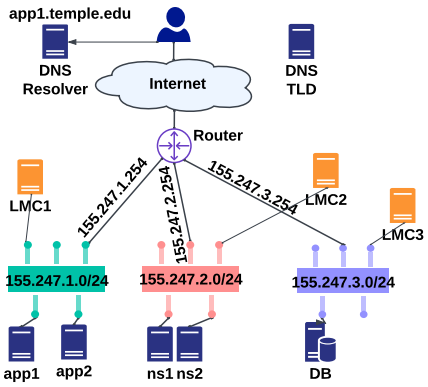


Fig. 2. Compact Network.

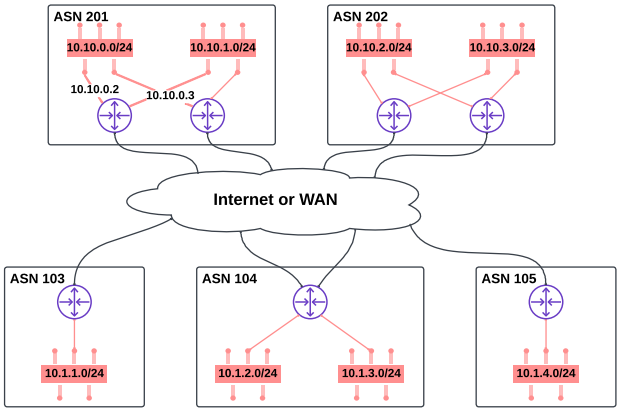


Fig. 3. Complex Network setups.

Our network includes TAs that serve as regular users and diagnosis tools capable of normal network access and specialized diagnosis tasks. The network hosts two main application servers, App1 and App2, as shown in Fig. 2. TA LMC1 is connected with App1 and App2 on the same Ethernet switch in VLAN A. Name servers ns1 and ns2, along with TA LMC2, are in VLAN B, managing domain name resolutions. The DB server and TA LMC3 are located in VLAN C. Each device’s IP configuration is detailed in Table 3. App1 and App2’s functionality critically relies on the DB server’s availability. However, due to the limitations of the SEED emulator, each VLAN is represented as a separate network in our configuration.

Table 3 presents the IP configurations for the network depicted in Fig. 2. In this setup, two misconfigurations are notable: App1’s subnet mask is incorrectly set to 255.255.0.0, and the DB server’s subnet mask is also misconfigured as 255.255.0.0. This configuration causes the DB server to mistakenly assume that App1 and App2, with IP addresses 155.247.1.1/16 and 155.247.1.2/24, respectively, are within its local subnet (155.247.3.1/16). As a result, the DB server incorrectly routes responses meant for App1 and App2 to its local network instead of the intended router, leading to communication failures. These misconfigurations and their subsequent network faults are further elucidated in Tables 5 and 4, which display the diagnosis results of these faults.

Table 6. Misconfigured IP Address of APP Server

(a) Elapsed Time (sec)

Number of tests	Test	Time
t_1	`host <DNS record>`	2.95
t_2	`nmap <ip addr>`	1.85
t_3	`ping -c 5 <ip addr>`	14.54
Total	—	**19.34**

(b) Assessing CL of CVs

CVs	t_1	t_2	t_3
IP Address	-0.22	-0.48	**-1**
DNS record	-0.22	-0.22	-0.22
Port Num.	0	-0.26	-0.26
MTU	0	0	0
Subnet mask	0	0	0
Gateway	0	0	0
Interface	0	0	0
Broadcast	0	0	0

4.3 Exploring Advanced Misconfiguration Scenarios in Complex Network Environments

In large enterprise environments, the intricacy of network configurations significantly heightens the risk and impact of misconfigurations. Unlike more straightforward setups, these complex networks often involve multiple layers of firewalls, routers, and redundancy protocols like VRRP, each adding numerous CVs. This multitude of CVs not only increases the potential for misconfiguration but also makes the fault diagnosis more challenging.

One common problem in such environments is asymmetric routing, particularly in locations with multiple gateways. Traffic might exit the network via one gateway but return through another. Inline firewalls, which expect symmetric traffic flow, may block this returning traffic, leading to disruptions often seen in routing and firewall misconfigurations.

Additionally, implementing high-availability configurations like VRRP, while beneficial for network resilience, further complicates the landscape. These setups introduce more CVs, increasing the chances of misconfigurations and making diagnosis more intricate. For instance, our network model, as shown in Fig. 3, simulates a typical enterprise core network. It comprises interconnected routers and diverse location setups, each varying in complexity from single-router connections to dual-router configurations with VRRP. In such network structures, the interplay between various network elements — from routers to firewalls — and their respective configurations necessitates a comprehensive analysis for effective troubleshooting.

In the next section, we present experimental results for the diagnosis of a variety of faults injected into both the compact and complex networks introduced here.

5 EXPERIMENTAL RESULTS

To validate the robustness of our approach, we examine a variety of scenarios involving service failures, considering cases where issues arise with a single service s_i or multiple services $\{s_1, \ldots, s_j\}$. This targeted approach enables timely identification and prioritization of affected services, providing a structured response to diverse diagnostic challenges. We specifically assess the impact of misconfigurations in APP servers, DB servers, and DNS servers by introducing controlled faults across different network environments, from small clusters to larger subnets. This analysis considers strategies for effectively detecting and resolving issues in various server configurations and network scales.

5.1 Misconfigured IP Address for App Server

Here we focus on the complex network size, which consists of 17 servers distributed across 5 AS'es. The algorithm is executed in several stages, including test selection, DNS record verification, IP address inspection, nmap scanning, and connectivity checks. The entire procedure takes 19.34 seconds, detailed in Table 6a. Notably, after the first 3 tests, the algorithm detects a misconfigured IP address with a CL of -1, indicating a strong likelihood of service failure due to misconfiguration (see Fig. 4a and Table 6b). The figure illustrates how CLs for various CVs change with each test.

Table 7. Misconfigured Interface on DB Server

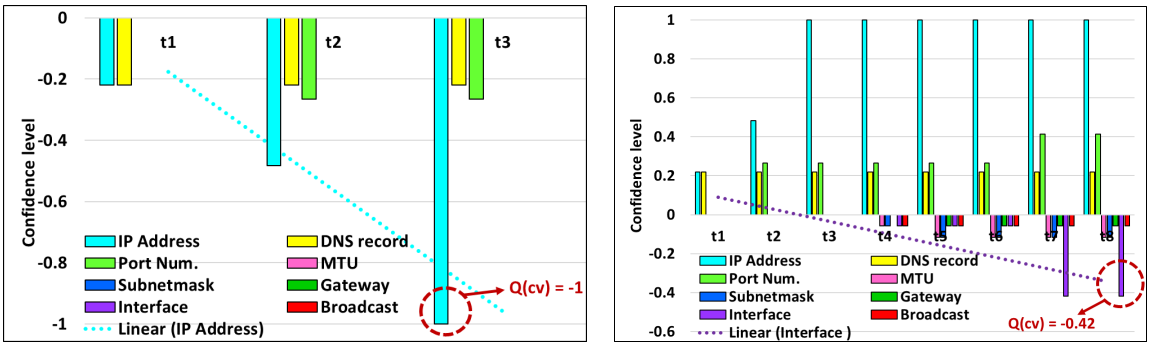
(a) Elapsed Time (sec)

Test		Time
t_1	`host <DNS record>`	0.41
t_3	`nmap <ip addr>`	0.74
t_5	`ping -c 5 <ip>`	4.69
t_2	`ip -4 addr show dev <intf>`	0.22
t_4	`ifconfig <intf>`	0.40
t_6	`traceroute <ip>`	0.34
t_7	`netstat -I <intf>`	0.22
t_8	`telnet <ip><port>`	0.25
Total		**7.27**

(b) Assessing CLs of CVs

CVs	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
IP Address	0.22	0.48	1.00	1.00	1.00	1.00	1.00	1.00
DNS record	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.22
Port Num.	0.00	0.26	0.26	0.26	0.26	0.26	0.26	0.41
MTU	0.00	0.00	0.00	-0.06	-0.12	-0.12	-0.12	-0.12
Subnet mask	0.00	0.00	0.00	-0.06	-0.12	-0.12	-0.12	-0.12
Gateway	0.00	0.00	0.00	0.00	-0.06	-0.06	-0.06	-0.06
Interface	0.00	0.00	0.00	-0.06	-0.06	-0.06	-0.42	**-0.42**
Broadcast	0.00	0.00	0.00	-0.06	-0.06	-0.06	-0.06	-0.06

For instance, after the first test (t_1 : `host <DNS record>`), the CL for IP address and DNS record decreases to -0.2. Subsequently, after t_2 : `nmap <ip addr>`, it further drops by -0.26. Finally, during t_3 : `ping -c 5 <ip addr>`, the CL for the IP address hits -1, leading the algorithm to stop as it identifies the misconfiguration.



(a) Misconfigured IP Address on the App Server. (b) Misconfigured Interface on the DB Server.

Fig. 4. Illustrating the Changes in CLs of CVs during Incremental Testing for Identifying Misconfiguration.

5.2 Misconfigured Interface for DB Server

In this network configuration scenario, we investigate the presence of a misconfigured interface within the DB server across different network sizes, including various numbers of servers distributed across AS. Here our algorithm identifies the misconfiguration after executing a fixed set of 8 tests in all scenarios of the complex network size with 17 servers distributed across 5 AS. The execution of the algorithm encompasses several phases, each contributing to the precise identification of the misconfigured DB server interface. The total elapsed time for the entire procedure is 7.27 seconds, with the crucial aspect being the detection of the misconfigured interface upon completion of the initial 8 tests, as shown in Table 7a. While all CVs are above the confidence threshold of -0.6, the algorithm pinpoints the DB server's interface as the most concerning misconfiguration, with a CL of -0.42 (refer to Fig. 4b and Table 7b). This highlights the algorithm's ability to identify the most significant misconfiguration, even when individual variables are still above the set threshold.

Table 8. Misconfigured Subnet Mask on DNS Server

(a) Elapsed Time (sec)

Test		Time
t_1	`host <DNS record>`	0.40
t_3	`nmap <ip addr>`	0.83
t_5	`ping -c 5 <ip>`	4.65
t_2	`ip -4 addr show dev <intf>`	0.16
t_4	`ifconfig <intf>`	0.34
t_6	`traceroute <ip>`	0.41
t_7	`netstat -I <intf>`	0.55
t_8	`telnet <ip><port>`	0.46
Total		**7.76**

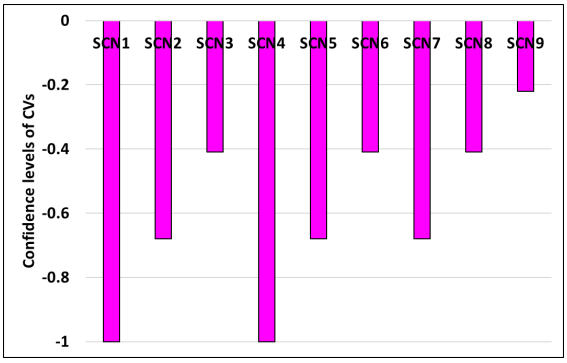
(b) Assessing CLs of CVs

CVs	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
IP Address	0.22	0.48	1.00	1.00	1.00	1.00	1.00	1.00
DNS record	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.22
Port Num.	0.00	0.26	0.26	0.26	0.26	0.26	0.26	0.41
MTU	0.00	0.00	0.00	0.06	0.12	0.12	0.12	0.12
Subnet mask	0.00	0.00	0.00	-0.06	-0.12	-0.12	-0.12	**-0.12**
Gateway	0.00	0.00	0.00	0.00	0.06	0.06	0.06	0.06
Interface	0.00	0.00	0.00	0.06	0.06	0.06	0.42	0.68
Broadcast	0.00	0.00	0.00	0.06	0.06	0.06	0.06	0.06

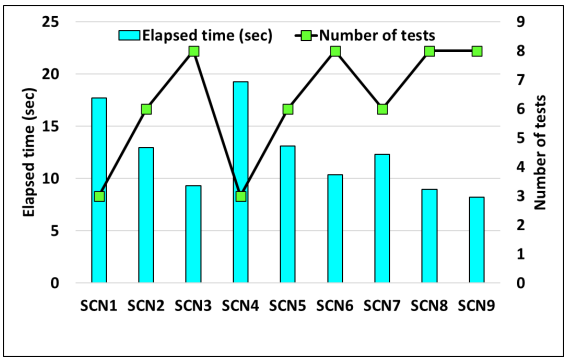
5.3 Misconfigured Subnet Mask for DNS Server

In this scenario, we want to identify a misconfigured subnet mask within the DNS server across different network sizes. Our algorithm detects this in 8 tests in all scenarios. We highlight the scenario in the complex network size. The algorithm's execution is divided into several phases, each contributing to the precise identification of the misconfigured subnet mask within the DNS server. The total elapsed time for the entire process is 7.76 seconds (refer to Table 8a), and the critical point is the early detection of the misconfigured subnet mask upon completion of the initial eight tests. Importantly, the CL associated with the subnet mask is -0.12 (refer to Fig. 6 and Table 8b).

This scenario sheds light on a crucial aspect of the algorithm's operation: when there are several tests dedicated to checking the misconfiguration (e.g., for the IP address), the algorithm can achieve a high CL for identifying failures. However, when tests involve many CVs or are not sufficiently discriminative, the CL of the true may not have a very low negative value but still has the lowest negative value among other variables. In particular, there are no specific tests for checking the mask; instead, those are typically bundled with checking the IP address. In other words, if we can design a test that focuses on the subnet mask, the algorithm will likely be able to determine its misconfiguration with higher confidence.



(a) CLs of Misconfigurations in Each Scenario.



(b) Total Elapsed Time (sec) and Total Test Counts.

Fig. 5. Overview of scenarios involving misconfigurations in one server.

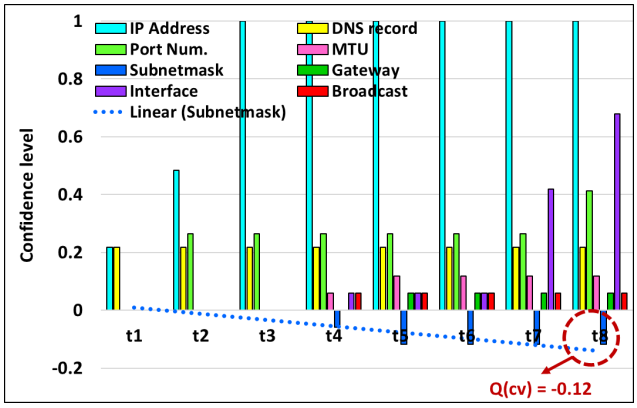


Fig. 6. Changes in CLs of CVs during Incremental Testing for Identifying Incorrect SubnetMask on the DNS Server.

5.4 Misconfigurations in a Single Server

Fig. 5 provides a visual representation of the misconfigurations across various scenarios. The scenarios include misconfigurations in the IP address of the App server (SCN1), port number of the App server (SCN2), the interface of the App server (SCN3), the IP address of the DB server (SCN4), port number of the DB server (SCN5), the interface of the DB server (SCN6), port number of the DNS server (SCN7), the interface of the DNS server (SCN8), and DNS entry of the App server (SCN9). Fig.5a illustrates the CLs associated with each misconfiguration. Notably, scenarios involving IP address misconfigurations in both SCN1 and SCN4 exhibit the most negative CLs, both at -1. Additionally, misconfigurations related to port numbers, whether in SCN2 or SCN5, consistently show CLs of -0.68. Fig.5b represents the elapsed time and the number of tests for each misconfiguration scenario. For instance, scenarios with IP address misconfigurations in both SCN1 and SCN4 show relatively longer elapsed times (17.69s and 19.23s), due to ping unreachability, but require only three tests. Conversely, scenarios like port number misconfigurations in SCN2 and SCN5 strike a balance with moderately low elapsed times (12.98s and 13.09s) and a moderate number of tests (six).

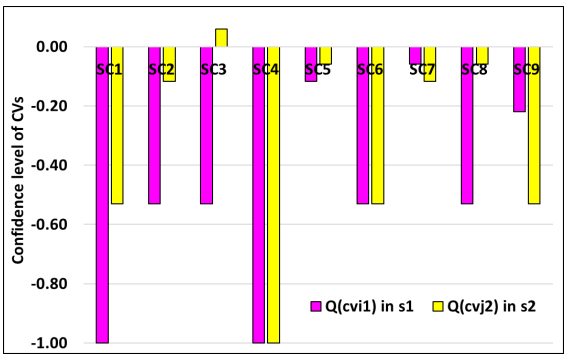
5.5 Misconfigurations in Multiple Servers

In our analysis of scenarios, we employ a systematic approach to replicate diverse misconfigurations across network components as shown in Table 9. This table shows 9 scenarios where two services, s_1 and s_2, encounter problems. cv_{i1} and cv_{j2} represent misconfigured CVs in services s_1 and s_2, respectively. In scenario SC1, the App server, named APP_num6, with IP address 10.10.3.30 is supposed to connect to DNS server, named DNS_num50, on port $53/tcp$. However, both the App server and DNS server have different IP addresses and port numbers. This results in a failure between the App server and the DNS server, but the specific misconfigurations causing the fault are unknown. Our approach successfully identifies the problem in both IP address and port number with CLs of $Q(cv_{i1}) = -1$ sand $Q(cv_{j2}) = -0.53$. This information is illustrated in Fig. 7a. Since cv_2 has the most negative CL among the misconfigurations, we prioritize it as the likely cause of the problem.

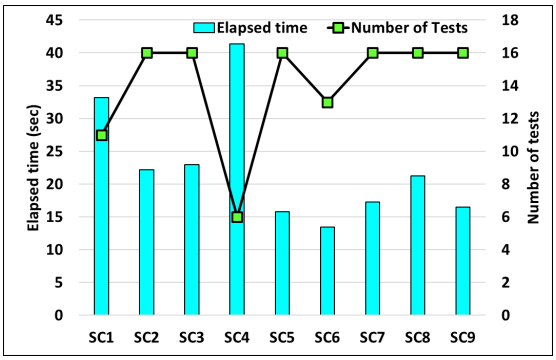
In scenario SC2, the DNS server, named DNS_num70, and the App server, named APP_num4, are set up with port $53/tcp$ and subnet mask 10.10.3.0/24, respectively. However, both are misconfigured, causing communication problems between them since they expect different port configurations. Our approach shows CLs ($Q(cvi1) = -0.53$ and $Q(cv_j2) = -0.12$), suggesting a high likelihood of misconfiguration in both the DNS server's port and the App server's subnet mask.

Table 9. Total Elapsed Time and Total Number of Tests for Diagnosing Two Misconfigurations in Two Different Servers.

Scenarios	s_1	cv_{i1}	s_2	cv_{j2}	$Q(cv_{i1})$	$Q(cv_{j2})$	Num. of Tests	Elapsed time (sec)
SC1	APP_num6	IP Address	DNS_num50	Port Num.	-1	-0.53	11	33.15
SC2	DNS_num70	Port Num.	APP_num4	Subnet mask	-0.53	-0.12	16	22.19
SC3	APP_num1	Port Num.	DB_num1	Gateway	-0.53	0.06	16	22.94
SC4	APP_num6	IP Address	DB_num5	IP Address	-1	-1	6	41.33
SC5	DNS_num50	Subnet mask	DB_num3	Gateway	-0.12	-0.06	18	15.82
SC6	DNS_num60	Port Num.	APP_num8	Port Num.	-0.53	-0.53	13	13.45
SC7	APP_num6	Gateway	DB_num5	Subnet mask	-0.06	-0.12	16	17.23
SC8	APP_num8	Port Num.	DB_num3	Subnet mask	-0.53	-0.06	16	21.23
SC9	DNS_num60	DNS record	APP_num3	Port Num.	-0.22	-0.53	16	16.45



(a) CLs of both Misconfigurations in Each Scenario.



(b) Total Elapsed Time (sec) and Total Test Counts.

Fig. 7. Overview of scenarios involving misconfigurations in two services.

In scenario SC3, the App server APP_num1 is supposed to use port $80/tcp$, and the DB server's gateway (DB_num1) should be set to 10.10.1.2. However, there is a mistake in their configurations, which can cause a problem in the connection between the App server and the DB server. Our approach indicates CLs ($Q(cvi1) = -0.53$ and $Q(cv_{j2}) = -0.06$). These CVs have the most negative CLs compared to other configurations. In SC4, the App server (APP_num6) should use the IP address 10.10.3.30, and the DB server's IP address (DB_num1) is expected to be 10.10.3.60. However, there's an error in their configurations. If the two servers are not intended to communicate directly, this misconfiguration can result in unintended network traffic. Our method reveals high CLs ($Q(cv_{i1}) = -1$ and $Q(cv_{j2}) = -1$) for both misconfigurations.

We also explore scenario SC5 involving conflicting subnet mask and gateway configurations between the DNS server DNS_num50 (10.1.1.0/24) and the DB server DB_num3 (10.10.1.2), leading to disrupting network routing between the two servers. Our algorithm assigns the negative CLs ($Q(cv_{i1}) = -0.13$ and $Q(cv_{j2}) = -0.058$) for both the DNS server's subnet mask and the DB server's gateway.

In scenario SC6, the DNS server DNS_num60 is expected to use port number $53/tcp$, and the App server APP_num8 should also be configured with the port number $53/tcp$. LMC shows that there is a mistake in the servers, indicating communication failures, as the two servers are using different ports for communication. The CLs are $Q(cv_{i1}) = -0.53$ for the DNS server's port configuration and $Q(cv_{j2}) = -0.53$ for the App server's port configuration and show misconfigurations correctly.

For scenario SC7, the App server's gateway (APP_num6) is set to 10.10.3.2, and the DB server's subnet mask is configured as 10.10.3.0/24; however, they are misconfigured. These conflicting configurations can disrupt network routing between the App server and the DB server. Our assessment indicates misconfigurations with CLs ($Q(cv_{i1}) = -0.06$ and $Q(cv_{j2}) = -0.12$) that are the most negative CLs among other CVs.

Table 10. Comparison of Troubleshooting Effectiveness Across Different Expert Domains.

Approach	Scenario 1	Scenario 2	Scenario 3	Scenario 4	Scenario 5	Total
Expert 1	2	9	15	8	8	42
Expert 2	2	9	10	8	5	34
Expert 3	2	11	9	6	8	36
Expert 4	2	9	8	12	4	35
Expert 5	2	9	8	12	4	35
ConfExp	2	8	4	11	5	30

In SC8, the App server (APP_num8) is expected to use port $80/tcp$, and the DB server (DB_num3) should be configured with a subnet mask $10.10.1.0/24$. However, they are set up wrong, causing problems in how the App server and the DB server connect. our analysis reveals misconfigurations with CLs ($Q(cv_{i1}) = -0.53$ and $Q(cv_{j2}) = -0.06$).

Finally, scenario SC9 involves the DNS server (DNS_num60) with a DNS record ($app3.temple.edu$) and the App server (APP_num3) configured with a port ($80/tcp$). As they are set up wrong, it can cause communication faults as the expected ports do not align. Our approach indicates CLs ($Q(cv_{i1}) = -0.22$ and $Q(cv_{j2}) = -0.53$) for misconfigurations in the DNS server's DNS record and the App server's port number.

Fig. 7b presents the total elapsed time and the number of tests for each scenario. Scenarios like SC1 and SC4, despite longer elapsed times (33.15s and 41.33s), require fewer tests (11 and 6) to achieve high CLs (-1.00 CL), primarily due to ping unreachability. Conversely, scenarios like SC6 show low elapsed time, appropriate CLs, and a low number of tests, indicating rapid identification of misconfigurations.

5.6 Validation of Automated Diagnosis Procedure

As discussed in section 6, *the prior work on diagnosis largely deals with rather abstract models where a test is specified by the set of "nodes" or resources that it involves, along with the assumption that any fault in these nodes will lead to test failure. The prior work also does not concern CVs of a resource directly.* Because of these limitations, a comparison of our work against prior art is not meaningful. Instead, because of very realistic nature of our test target (i.e., enterprise network running real protocols used in practice) and tests (e.g., tests like traceroute that administrators actually use), we decided to compare our approach against the prevalent manual diagnosis by experts. We used the inputs from 5 experts in form of a flow-chart to diagnose an issue reported in the same way as for our diagnosis procedure. The reported issue may be caused by one or more of several possible underlying faults, which the expert did not know. We followed the flow-chart to determine the tests performed and success in the diagnosis for specific underlying faults.

Most experts have somewhat different skill set and perspective, which is reflected in how they approach the diagnosis or how successful they are.

Table 10 illustrates the number of tests conducted by each expert to identify specific misconfigurations. The table outlines the results of troubleshooting various scenarios. In Scenario 1, where a client reported an issue accessing the application server due to a misconfigured IP address, all experts, including our methodology, ConfExp, required only two tests to identify the fault.

In Scenario 2, involving a misconfigured interface in the database server, all experts and our approach successfully identified the misconfiguration, with ConfExp requiring the fewest tests (8). Experts 1, 2, 4, and 5 required 9 tests, while Expert 3 needed 11 tests.

For Scenario 3, which dealt with misconfigured firewall rules blocking specific traffic types, Experts 4 and 5 required the fewest tests (8), followed closely by Expert 3 (9 tests), Expert 2 (10 tests), and Expert 1 (15 tests). ConfExp was the

most efficient, needing only 4 tests. Scenario 4, involving misconfigured IP addresses in the application server and port numbers in the DNS server. The number of tests varied, with Expert 3 requiring the fewest tests (6) and Experts 4/5 the most (12 tests each). ConfExp did not do so well here (11 tests) but remains competitive. In Scenario 5, where clients experienced connectivity issues due to a misconfigured routing table, Experts 4 and 5 required the fewest tests (4), followed by Expert 2 and our approach (5 tests each), and Experts 1 and 3 (8 tests each).

The last column lists the overall results – the total number of tests across all scenarios. ConfExp required the least number of total tests (30), even though it did not score the best in all scenarios individually. Experts 2, 4, and 5 performed similarly with 34 and 35 tests, respectively, while Experts 1 and 3 required 42 and 36 tests, respectively. This demonstrates that while all approaches were successful in identifying misconfigurations, ConfExp was the most efficient overall. Furthermore, our approach is fully automated and thus much more convenient to use.

This comparison highlights the advantages of our approach as a comprehensive and automated solution. Unlike manual diagnosis, which relies heavily on the individual expertise, our method is fully automated, approaches the diagnosis in a standard way, and overall performs at least as well as the experts in diagnosing the problem.

6 RELATED WORK

The topic of fault detection and localization is fundamental to both hardware and software, and extensive prior research exists. However, misconfiguration related faults are unique in at least three ways: (a) A resource or "node" usually has several CV's that may be "faulty", (b) The "fault" in a CV is often not a matter of it being faulty/okay; instead, the fault manifests itself because of desired relationships/dependencies between them not being satisfied, and (c) As a result of (b), a single test typically cannot ascertain a CV as being faulty/okay, since the tests mostly check for the desired relationships being satisfied. As a result, the plethora of abstract graph based techniques that that determine if a node is faulty/okay via coverage by individual tests are not applicable. In fact, to the best of our knowledge, there is no automated technique in the literature that we can compare our results with, as demonstrated by our review below.

Traditional network management and monitoring tools like SNMP [12], NETCONF [13], and Nagios [14] provide essential services for centralized network health monitoring, fault alerting, and event logging. These systems rely on mechanisms such as SNMP polling, syslog data, and ICMP probes to capture high-level device states and log events, enabling administrators to monitor network availability and performance. For example, SNMP and NETCONF allow administrators to query or configure device parameters remotely. Nagios uses both central and remote pollers to check device status and alert on predefined conditions. However, while effective for real-time status monitoring, these tools do not address detailed misconfiguration diagnostics at the CV level, which involves assessing dependencies and configurations of services and nodes in a way that accounts for specific operational relationships among CVs. Unlike general network monitoring, our work focuses on detecting misconfigurations in the relationships among CVs themselves. This aspect has been traditionally outside the scope of network management architectures, which prioritize connectivity and availability over in-depth configuration analysis.

Lamraoui et al. [15] advocated for a formula-based approach to automatic fault localization in multi-fault programs, emphasizing the importance of localizing root-causes to go beyond mere fault detection. Similarly, Chen et al. [16] introduced the BALANCE method, employing Bayesian linear attribution specifically for root-cause localization, thus recognizing the need for a nuanced understanding of fault origins. Wong et al. [17] offered an insightful overview of fault localization methodologies, emphasizing the complexity of faults and the importance of integrating root-cause analysis for a comprehensive understanding of fault management. Zhang et al. [18] developed a two-level fault diagnosis

and root-cause analysis scheme for interconnected dynamic systems, underscoring the necessity of discerning root-causes at a granular level for effective fault resolution. Despite these advancements, challenges remain in achieving comprehensive and practical fault diagnosis solutions.

The concept of utilizing probes to diagnose network faults was initially introduced by Brodie et al. in [19], where they outlined several approximation algorithms for selecting a target probe set for the network. Rish et al. extended this idea in [20, 21], proposing cost-efficient and adaptive diagnostic probing techniques. These techniques employ an information-theoretic approach to choose the target probe set, initially selecting a small set of highly informative probes and dynamically adapting them based on the observed network state, but face weaknesses in computational complexity due to Bayesian network inference. Despite reducing probes by over 60%, real-time suitability is compromised.

Another approach, based on entropy approximation, was presented by Zheng et al. in [22]. This method used a loopy belief propagation model to compute approximate values for marginal and conditional entropy. Natu et al. explored adaptive strategies for probe selection in [23, 24], dividing the fault diagnosis process into fault detection and fault localization sub-steps. The fault detection step periodically sends out a target probe set to detect faults, triggering the fault localization step to identify the exact fault location. Lu et al. in [25] proposed an adaptive method inspired by [26], dividing the fault detection process into stages and selecting small probe sets to check network nodes progressively. Tang et al. introduced the Active Integrated Fault Reasoning technique in [27], combining network symptom correlation with active probing for fault localization. This method adapts the probe set based on observed symptoms.

Incremental adaptive probing approaches, suitable for real-time monitoring and diagnosis, select and send probes as needed in response to failures. Carmo et al. in [28] applied active probing for intrusion detection in wireless multi-hop networks using a recursive probe selection scheme and a Bayesian classifier. Garshasbi in [29] proposed an algorithm combining active and passive monitoring for fault diagnosis based on Ant Colony optimizations. These approaches rely on different dependency models for networks, which can be deterministic or non-deterministic. Natu et al. in [26] proposed a probabilistic dependency model representing the relationship between a probe and probed nodes with probability values. However, obtaining accurate probabilistic models with dynamic probabilities is largely infeasible. The placement of probing stations in the network poses another challenge. Heuristic-based approaches described in [23, 30, 31] incrementally select nodes for probing stations based on the number of independent paths to a node. Traverso et al. in [32] developed a network monitoring tool using a hybrid approach that correlates throughput measurements from probes with alarm notifications from passive measurement utilities.

Several works provide comprehensive surveys of RCA techniques and methodologies, highlighting diverse approaches and applications within network diagnostics and management[33–37]. El-Shekeil et al. in [38] presented the CloudMiner framework for systematic failure diagnosis in enterprise cloud environments. This framework emphasizes probing station selection and utilizes a minimal set of network probes. It is specifically designed to address the complexities and interdependencies of network services and components in cloud systems, aiming to efficiently detect and diagnose failures in these intricate environments.

7 CONCLUSION AND FUTURE WORK

In this paper, we have addressed the problem of diagnosing misconfigurations in enterprise IT systems. ConfExp focuses on reactive diagnosis, sequentially selecting diagnosis tests to minimize testing effort. We show that many of the commonly occurring misconfiguration problems can be tackled automatically without any manual intervention. We saw that the root-cause of the problem takes less than 35 seconds in all cases. This is to be compared against the typical manual process where the administrator might try different tests with diagnosis times ranging from minutes to hours.

Furthermore, since we only use the universally available test commands, the mechanism can be easily implemented in any enterprise network. We plan to open-source our solution so that it can be used and further enhanced by the community.

Our effort so far made several simplifying assumptions that we plan to address in the future. The most prominent of these is the assumption of a global console which we discuss below. Other aspects relate to the limitations of the SEED emulator in its current state, but given the flexible architecture, it is possible to extend SEED for many additional capabilities. In particular, SEED currently does not support layer2 networks but can be implemented by using open-source switching software like OPX. Another issue concerns our assumption of visibility being limited to a subnet that has its own test agent. However, defining visibility boundaries and, hence, the scope of LMC across multiple subnets is straightforward.

7.1 Limitations of Diagnosis Methodology

It is important to note that our diagnosis methodology is not intended to encompass the deep semantics of complex enterprise applications; instead, it is largely intended for interactions between applications through the network and their basic operability status. Thus the only CVs of interest to our algorithm are those relating to the reachability of a service and whether the basic parameters of the service (e.g., DNS mapping, firewall settings, routing, etc.) are set up properly or not. In case the service is found to be nonresponsive or ill configured by these very basic criteria, the responsibility of detailed debugging shifts to other, focused tools that can deal with the complexities of the application. From this perspective, *the foundational principles of our model are adaptable and can be extended to other applications and for somewhat deeper examination of the applications that we have already considered.* The methodology would thus lead to automated root-causing up to the level of a small set of individual services, which could then be examined further.

However, there are cases where the LMC might not indicate a failure, even when clients report issues. Such instances may include complex network problems like BGP hijacking, specific routing anomalies, or misconfigured firewall rules that affect particular paths without causing a complete network outage. Addressing these gaps is beyond the scope of this paper and will be addressed in future work.

As an example, DNS itself has many aspects to its configuration that we have not touched upon. In particular, DNS allows records for specific services (e.g., email), and such records have a priority as a configuration parameter. Now if the Mail Exchange (MX) record with the smallest priority value (which is actually the highest priority) points to a mail server that does not accept Simple Mail Transfer Protocol (SMTP) connections, the mail send will fail over to the other exchange servers. Diagnosing such a problem would require tests to check the mail server configuration, too. Similarly, a missing pointer (PTR) record (reverse translation) may flag the incoming mail as spam. Suitable tests, along with their relevance, need to be added to handle such features.

7.2 Global Console Emulation

As mentioned before, the abstraction of a global console must deal with the unavoidable distributed systems issues. In particular, consolidating status from multiple LMCs will involve varying delays on top of the varying delays in reflecting status changes in the LMC (usually very small) and delays in remote access to this information. As is well recognized, it is impossible to learn the true status of remote entities in a distributed system. Furthermore, very frequent or entirely event-driven status transmission from all LMCs to a centralized GMC site is unscalable. With permanent and relatively infrequent faults, a periodic fetching of LMC status along with retry in case of incorrect diagnosis should

be adequate; however, other methods, such as hierarchical diagnosis, may be needed in other cases and will be explored in the future.

Another major issue in a large distributed system is that of complex accessibility and visibility issues, even for testing purposes. In particular, even the Points of Presence (PoPs) of a single geographically distributed organization may be unwilling to allow remote testing or restrict what types of tests are allowed. This substantially complicates testing and may require testing from specific testing agents. We have examined the issue of accessibility in the past [38–40], but its integration into a comprehensive testing strategy remains.

REFERENCES

[1] S. Newman, *Building microservices*. " O'Reilly Media, Inc.", 2021.

[2] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[3] M. Williams and A. Vance, "Microsoft takes blame for web site access failures," URL http://www.computerworld.com/article/2590639/networking/microsoft-takes-blame-for-web-site-access-failures.html, 2001, [Online; accessed 3-July-2023].

[4] "Misconfiguration brings down entire .se domain in sweden," URL http://www.circleid.com/posts/misconfiguration_brings_down_entire_se_domain_in_sweden, 2009, [Online; accessed 3-July-2023].

[5] "Apple blames itunes outage on dns error. what does that mean?" URL https://www.csmonitor.com/Technology/2015/0311/Apple-blames-iTunes-outage-on-DNS-error.-What-does-that-mean, 2015, [Online; accessed 3-July-2023].

[6] A. Kurtz, "Delta malfunction on land keeps a fleet of planes from the sky," URL https://www.nytimes.com/2016/08/09/business/delta-air-lines-delays-computer-failure.html, 2016, [Online; accessed 3-July-2023].

[7] "Southwest airlines' router grounds 2,300 flights," URL https://availabilitydigest.com/public_articles/1108/southwest_airlines.pdf, 2016, [Online; accessed 3-July-2023].

[8] W. Du, H. Zeng, and K. Won, "Seed emulator: an internet emulator for research and education," in *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, 2022, pp. 101–107.

[9] P.-W. Tsai, F. Piccialli, C.-W. Tsai, M.-Y. Luo, and C.-S. Yang, "Control frameworks in network emulation testbeds: A survey," *Journal of computational science*, vol. 22, pp. 148–161, 2017.

[10] J. Ahrenholz, C. Danilov, T. R. Henderson, and J. H. Kim, "CORE: A real-time network emulator," in *MILCOM 2008-2008 IEEE Military Communications Conference*. IEEE, 2008, pp. 1–7.

[11] W. Du and H. Zeng, "The seed internet emulator and its applications in cybersecurity education," 2022.

[12] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "Simple network management protocol (SNMP)," *Internet Engineering Task Force (IETF) RFC*, vol. 1157, 1990. [Online]. Available: https://datatracker.ietf.org/doc/rfc1157/

[13] R. Enns, "Netconf configuration protocol," *Internet Engineering Task Force (IETF) RFC*, vol. 4741, 2006. [Online]. Available: https://datatracker.ietf.org/doc/rfc4741/

[14] W. Barth, *Nagios: System and Network Monitoring*. San Francisco, CA: No Starch Press, 2008.

[15] S.-M. Lamraoui and S. Nakajima, "A formula-based approach for automatic fault localization of multi-fault programs," *Journal of Information Processing*, vol. 24, no. 1, pp. 88–98, 2016.

[16] C. Chen, H. Yu, Z. Lei, J. Li, S. Ren, T. Zhang, S. Hu, J. Wang, and W. Shi, "Balance: Bayesian linear attribution for root cause localization," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–26, 2023.

[17] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[18] M. Zhang, Z. Li, B. Dahhou, M. Cabassud, and C. Volosencu, "Root cause analysis of actuator fault," in *Actuators*. IntechOpen, 2018, p. 131.

[19] M. Brodie, I. Rish, and S. Ma, "Optimizing probe selectionfor fault localization," in *Proceedings of the 12th IFIP/IEEEInternational Workshop on Distributed Systems: Operationsand Management (DSOM01)*, 2001.

[20] I. Rish, M. Brodie, N. Odintsova, S. Ma, and G. Grabarnik, "Real-time problem determination in distributed systems using active probing," in *2004 IEEE/IFIP Network Operations and Management Symposium (IEEE Cat. No. 04CH37507)*, vol. 1. IEEE, 2004, pp. 133–146.

[21] I. Rish, M. Brodie, S. Ma, N. Odintsova, A. Beygelzimer, G. Grabarnik, and K. Hernandez, "Adaptive diagnosis in distributed systems," *IEEE Transactions on neural networks*, vol. 16, no. 5, pp. 1088–1109, 2005.

[22] A. X. Zheng and I. Rish, "Efficient test selection in active diagnosis via entropy approximation," *arXiv preprint arXiv:1207.1418*, 2012.

[23] M. Natu and A. S. Sethi, "Probe station placement for fault diagnosis," in *IEEE GLOBECOM 2007-IEEE Global Telecommunications Conference*. IEEE, 2007, pp. 113–117.

[24] D. Jeswani, M. Natu, and R. K. Ghosh, "Adaptive monitoring: application of probing to adapt passive monitoring," *Journal of Network and Systems Management*, vol. 23, pp. 950–977, 2015.

[25] L. Lu, Z. Xu, W. Wang, and Y. Sun, "A new fault detection method for computer networks," *Reliability Engineering & System Safety*, vol. 114, pp. 45–51, 2013.

[26] M. Natu and A. S. Sethi, "Active probing approach for fault localization in computer networks," in *2006 4th IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services*. IEEE, 2006, pp. 25–33.

[27] Y. Tang, E. S. Al-Shaer, and R. Boutaba, "Active integrated fault localization in communication networks," in *2005 9th IFIP/IEEE International Symposium on Integrated Network Management, 2005. IM 2005.* IEEE, 2005, pp. 543–556.

[28] R. do Carmo, J. Hoffmann, V. Willert, and M. Hollick, "Making active-probing-based network intrusion detection in wireless multihop networks practical: A bayesian inference approach to probe selection," in *39th Annual IEEE Conference on LCN*. IEEE, 2014, pp. 345–353.

[29] M. S. Garshasbi, "Fault localization based on combines active and passive measurements in computer networks by ant colony optimization," *Reliability Engineering & System Safety*, vol. 152, pp. 205–212, 2016.

[30] B. Patil, S. Kinger, and V. K. Pathak, "Probe station placement algorithm for probe set reduction in network fault localization," in *2013 International Conference on Information Systems and Computer Networks*. IEEE, 2013, pp. 164–169.

[31] E. Salhi, S. Lahoud, and B. Cousin, "Localization of single link-level network anomalies," in *2012 21st International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2012, pp. 1–9.

[32] S. Traverso, E. Tego, E. Kowallik, S. Raffaglio, A. Fregosi, M. Mellia, and F. Matera, "Exploiting hybrid measurements for network troubleshooting," in *2014 16th International Telecommunications Network Strategy and Planning Symposium (Networks)*. IEEE, 2014, pp. 1–6.

[33] A. Dusia and A. S. Sethi, "Recent advances in fault localization in computer networks," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 4, pp. 3030–3051, 2016.

[34] G. V. Maia, T. M. Coutinho, E. B. Gonçalves, G. R. Silva, E. M. Mendes, M. M. Mendes, S. R. Caetano, G. M. Mitt, and A. P. Braga, "One class density estimation approach for fault detection and rootcause analysis in computer networks," *Journal of Network and Systems Management*, vol. 30, no. 4, p. 69, 2022.

[35] C.-C. Yen, W. Sun, H. Purmehdi, W. Park, K. R. Deshmukh, N. Thakrar, O. Nassef, and A. Jacobs, "Graph neural network based root cause analysis using multivariate time-series kpis for wireless networks," in *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2022, pp. 1–7.

[36] P. Casas, J. Vanerio, and K. Fukuda, "Gml learning, a generic machine learning model for network measurements analysis," in *2017 13th International Conference on Network and Service Management (CNSM)*. IEEE, 2017, pp. 1–9.

[37] W. Wang, L. Tang, C. Wang, and Q. Chen, "Real-time analysis of multiple root causes for anomalies assisted by digital twin in nfv environment," *IEEE transactions on network and service management*, vol. 19, no. 2, pp. 905–921, 2022.

[38] I. El-Shekeil, A. Pal, and K. Kant, "CloudMiner: A systematic failure diagnosis framework in enterprise cloud environments," *Proc. of CLOUDCOM, Nicosia, Greece*, Dec 2018.

[39] M. . Athamnah, A. Pal, and K. Kant, "A framework for misconfiguration diagnosis in interconnected multi-party systems," *Proc. of ICCCN 2018*, 2018.

[40] M. Athamnah and K. Kant, "Multiparty database sharing with generalized access rules," in *Proc. of CloudCom, Luxemburg*, Dec 2016, pp. 198–205.