



(REVIEW ARTICLE)



A SCALABLE TESTING FRAMEWORK FOR VALIDATING TERRAFORM INFRASTRUCTURE SECURITY

Julien Bernard *

Cloud expert France.

* Corresponding Author; Email: julien.bd86@gmail.com

World Journal of Advanced Research and Reviews, 2024, 24(02), 2959–2969

Publication history: Received on 07 October 2024; revised on 22 November 2024; accepted on 29 November 2024

Article DOI: <https://doi.org/10.30574/wjarr.2024.24.2.3472>

Copyright © 2024 Author(s) retain the copyright of this article. This article is published under the terms of the Creative Commons Attribution License 4.0

ABSTRACT

Infrastructure as Code (IaC) is a key component for implementing DevSecOps in cloud. However, IaC misconfigurations can be automatically propagated to production systems, introducing significant security risks. This study analyzes the inconsistency of vulnerability detection across platforms in Terraform-based IaC security auditing tools and proposes an extensible test framework to address this issue. Experimental results show that even with identical IaC configurations, vulnerability detection outcomes differ across Terraform providers with distinct operational environments. The proposed framework automates tfsec-based security evaluation by injecting predefined vulnerabilities and customizing configurations for public-cloud, Kubernetes, and on-premises environments.

Keywords: DevSecOps, IaC (Infrastructure As Code), Terraform, Cloud Vulnerability, Tfsec

1 INTRODUCTION

In recent cloud environments, the DevSecOps paradigm—which incorporates security considerations throughout the entire software development lifecycle—is gaining attention. Among the component technologies requiring strict security review, Infrastructure as Code (IaC), which is utilized for scalability and consistency of infrastructure within DevOps pipelines, is particularly significant. IaC is a method of managing and provisioning computing infrastructure through specification files written in code form, including physical hardware configuration, permission management, and access control [1]. By adopting IaC, developers can perform version control and deployment automation while writing code that defines infrastructure, enabling efficient infrastructure management. Various IaC tools have been developed for commercial and public cloud services such as AWS, Azure, and Google Cloud [2], with representative IaC tools including Terraform [3], AWS CloudFormation [4], and ARM (Azure Resource Manager) [5].

Among these, Terraform is an open-source tool developed by HashiCorp and can be used not only for general cloud environments but also for automating Kubernetes cluster provisioning and cloud-native application deployment. Terraform has the strength of being platform-agnostic—operating across diverse infrastructure environments—while also being open source, which gives it excellent usability and universality. In fact, according to a Firefly survey, over 80% of respondents reported currently using Terraform, making it the most actively used IaC tool [6].

While IaC-based infrastructure management automation like Terraform enhances efficiency by rapidly reflecting infrastructure settings within DevSecOps pipelines alongside code changes, there is also a concurrent risk that misconfigurations or security vulnerabilities may be directly reflected in the infrastructure. Since Terraform can also introduce security vulnerabilities, the use of static security audit tools like tfsec [7] is essential. tfsec is a tool that statically analyzes Terraform code to detect potential security vulnerabilities in cloud environments. Among tfsec's official documentation, Checks provides Security Checks by risk level, organizing vulnerable code descriptions and remediation guides for various providers [8].

The need for automated security validation is further emphasized by recent patent developments in the cloud security domain [9], which describe comprehensive systems for real-time anomaly detection and automated threat response. These systems leverage machine learning algorithms and contextual analysis to continuously monitor cloud infrastructure, detect deviations from normal behavior, and execute predefined response actions without human intervention. This trend toward automation in cloud security directly supports the motivation for our proposed extensible testing framework, as both approaches aim to reduce manual intervention and improve the speed and accuracy of threat detection and mitigation.

However, the additional use of static security tools still has limitations from a security perspective. In the current context of increasingly diverse operational environments including multi-cloud and on-premises environments (private clouds), resource models, required fields, default values, and network semantics differ between public clouds (AWS, etc.), Kubernetes, and on-premises environments. Therefore, even when defining infrastructure for the same purpose using Terraform, static analysis tools like tfsec may not consistently recognize and represent the same vulnerable configurations across platforms. When performing automated infrastructure migration in the transition between diverse operational environments, security vulnerabilities may exist.

In fact, according to a case study conducted in this research, when code containing a vulnerability that grants public read permissions in an AWS environment was used to configure infrastructure with identical composition in an on-premises Kubernetes cluster environment, the vulnerability detection results differed. This result suggests that when the same IaC requires changing the Terraform provider due to a change in operational environment, resource names and values appropriate to each environment must be used, indicating the need for platform-specific customization.

To address these limitations, this study proposes an extensible security audit testing framework for infrastructure written in Terraform. To demonstrate the limitations of existing IaC static analysis tools in terms of extensibility, preliminary evaluation results for various Terraform providers identified different vulnerability detection outcomes. The proposed framework provides customization for security evaluation across three representative operational environments that can be built with Terraform: public cloud, Kubernetes clusters within public cloud, and on-premises Kubernetes cluster environments. Specifically, for each deployment scenario, we customize provider differences based on identical skeleton IaC code, then manually inject specific vulnerabilities. Subsequently, we design and conduct tfsec-based security evaluations for each operational environment to identify whether identical IaC vulnerabilities are well detected through static analysis.

This framework aligns with the broader industry movement toward automated, systematic security validation. Just as the patent [9] describes systems that integrate data collection, anomaly detection, and automated response modules to enhance cloud security, our framework provides a structured methodology for validating that security tools like tfsec can consistently detect vulnerabilities across different cloud platforms. Both approaches recognize that manual security processes are insufficient for the scale and complexity of modern cloud environments.

The structure of this paper is as follows. Chapter 2 provides background knowledge including an overview of IaC and Terraform, as well as representative commercial platform providers for building on-premises private clusters. Chapter 3 presents a case study including preliminary evaluation experiments on Terraform's security audit capabilities and comparative analysis of three on-premises providers. Chapter 4 covers the proposed testing framework, presents experimental performance analysis based on the framework, and Chapter 5 concludes the paper.

2 BACKGROUND KNOWLEDGE

2.1 Overview of IaC and Terraform

Infrastructure as Code (IaC), which defines infrastructure as code, enables infrastructure automation in cloud and on-premises environments. Terraform is a representative IaC tool that defines infrastructure as code and supports various operational environments. Terraform uses HCL (HashiCorp Configuration Language), a human-readable language, to define the desired state of infrastructure needed in cloud or on-premises environments, and upon completion of this phase, Terraform automatically builds the defined infrastructure. Based on these characteristics, Terraform can be

applied not only to public clouds such as AWS, Azure, and Google Cloud but also to on-premises cluster environments for building private cloud-native environments using tools like Matchbox [10]. In this case, even for identical resource configurations, resource names and syntax within IaC differ by platform.

```
resource "aws_instance" "app_server" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
  tags = {
    Name = "learn-terraform"
  }
}
```

Figure 1: Example of Provider-Specific Resources in Terraform (AWS)

```
provider "matchbox" {
  endpoint = var.matchbox_rpc_endpoint
  client_cert = file("~/matchbox/client.crt")
  client_key = file("~/matchbox/client.key")
  ca = file("~/matchbox/ca.crt")
}
```

Figure 2: Example of Provider-Specific Resources in Terraform (Matchbox)

2.2 On-Premises Environments: Matchbox, Rancher, Proxmox

Unlike when using public clouds, configuring on-premises environments is required within IaC when building private cloud environments. Terraform can be used in on-premises environments as it is compatible with APIs from various service providers and systems [12]. For building cloud-native environments, it is common to use fully managed services based on public clouds such as Amazon EKS (Elastic Kubernetes Service) and AKS (Azure Kubernetes Service) as they allow easy and secure cluster configuration. Meanwhile, an increasing number of companies are building and operating private cloud or hybrid cloud environments based on dedicated resources for handling sensitive data or ensuring compliance.

Platform providers for building Kubernetes clusters in on-premises environments include Matchbox [10], Rancher [13], and Proxmox [14]. Matchbox is a service that matches profiles for PXE booting and provisioning bare-metal machines. When used with Terraform, the poseidon/matchbox provider is used to automatically create Kubernetes nodes on bare-metal servers [10]. Rancher is a single platform for Kubernetes management, supporting deployment, execution, and operational management of containerized workloads. When used with Terraform, the rancher/rke provider can be utilized to install and manage Kubernetes clusters on SSH-accessible nodes. Proxmox is a KVM-based open-source virtualization platform that provides integrated management of LXC containers and virtual machines (VMs), while also supporting cluster configuration and resource management.

3 CASE STUDY

Whether static analysis tools like tfsec can effectively detect both configuration errors and security vulnerabilities has not been sufficiently explored in prior research. For preliminary evaluation, this study defined system configuration error cases that could potentially cause vulnerabilities at the Kubernetes cluster level and identified what security audit functions operate during the plan and launch processes using Terraform tools. As a first step, three types of intentional system configuration errors were injected into reference TF (Terraform) code. The injected system configuration errors were derived based on violation cases appearing in the Kubernetes manifest security configuration error categories [15]. These categories include various security configuration error types such as missing resource limits, absence of securityContext definitions, host network activation, and privilege misuse. Among these, this preliminary evaluation selected missing resource limits, duplicate IP allocation, and privilege misuse as experimental targets for analysis.

The first injected system configuration error is shown in Fig. 3, where resource limits are not specified. This configuration error category refers to cases where CPU and memory usage limits are not specified for containers within

a pod [15]. In this case, malicious users could increase traffic flow to cause unlimited CPU and memory requests, making the system vulnerable to denial-of-service attacks [16].

```
spec {
  container {
    name = "employee"
    image = "piomin/employee-service"
  }
  + container {
    + image = "piomin/employee-service"
    + image_pull_policy = (known after apply)
    + name = "employee"
    + stdin = false
    + stdin_once = false
    + termination_message_path = "/dev/termination-log"
    + termination_message_policy = (known after apply)
    + tty = false
  }
  + resources (known after apply)
}
```

Figure 3: Misconfiguration: Absent resources limit

The second injected configuration is a system setting error that assigns duplicate IP addresses to each node. To establish a normal network environment, IP address uniqueness must be guaranteed, and assigning duplicate IPs on the same network as in this configuration could expand into a security threat. Duplicate IP address allocation can degrade network stability and cause network collisions, thus serving as a factor that expands the potential attack surface. Fig. 4 shows an example configuration with duplicate IP allocation.

```
resource "libvirt_domain" "master" {
  name = "k8s-master"
  memory = "2048"
  vcpu = 2

  disk {
    volume_id = libvirt_volume.master_disk.id
  }

  network_interface {
    network_name = libvirt_network.k8s_network.name
    addresses = ["192.168.100.10"]
  }
}

resource "libvirt_domain" "worker" {
  count = 2
  name = "k8s-worker-${count.index}"
  memory = "2048"
  vcpu = 2

  disk {
    volume_id = libvirt_volume.worker_disk[count.index].id
  }

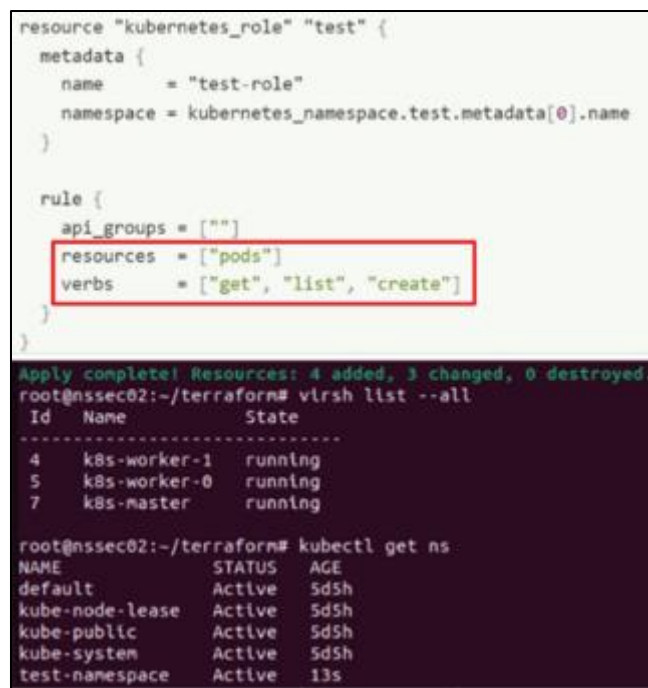
  network_interface {
    network_name = libvirt_network.k8s_network.name
    addresses = ["192.168.100.10"]
  }
}
```

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.

```
root@nssec02:~/terraform# virsh list --all
Id   Name               State
-----
4    k8s-worker-1      running
5    k8s-worker-0      running
7    k8s-master         running
```

Figure 4: Misconfiguration: Assigning duplicate IP address

The third injected system configuration error is a case of excessive privilege setting in the rule section. Fig. 5 shows an example of this system configuration error, permitting get, list, and create permissions on resources. When unnecessary excessive privileges are granted in this manner, if the service account is compromised, it could access or manipulate resources across the entire cluster, leading to serious security threats such as privilege escalation attacks.



```

resource "kubernetes_role" "test" {
  metadata {
    name      = "test-role"
    namespace = kubernetes_namespace.test.metadata[0].name
  }

  rule {
    api_groups = [""]
    resources  = ["pods"]
    verbs      = ["get", "list", "create"]
  }
}

```

```

Apply complete! Resources: 4 added, 3 changed, 0 destroyed.
root@nssec02:~/terraform# virsh list --all
Id      Name                State
-----
 4      k8s-worker-1        running
 5      k8s-worker-0        running
 7      k8s-master           running

root@nssec02:~/terraform# kubectl get ns
NAME                STATUS  AGE
default              Active  5d5h
kube-node-lease      Active  5d5h
kube-public          Active  5d5h
kube-system          Active  5d5h
test-namespace       Active  13s

```

Figure 5: Misconfiguration: Excessive authority setting in the rule section

Experimental results showed that the Terraform plan and launch processes for code injected with the three vulnerabilities operated normally without any warning messages. In other words, this means that despite the presence of IaC system configuration errors that could cause security threats within the code, plan and launch operations were still possible. This suggests that due to the absence of Terraform's own security verification process, vulnerable code can be immediately deployed, and without additional security audit tools, the built environment may be exposed to various vulnerabilities.

This finding is particularly significant when viewed alongside automated security systems described in recent patent literature [9]. The patent emphasizes the importance of automated detection mechanisms that can identify anomalies without requiring manual oversight. Our experimental results demonstrate a critical gap in existing IaC tools—they lack the built-in security verification that is increasingly recognized as essential in cloud security systems. This gap reinforces the need for both (1) static analysis tools like tfsec and (2) systematic frameworks like the one we propose to validate that these tools function consistently across platforms.

3.1 Comparison and Analysis of On-Premises Providers

Terraform utilization has thus far been more focused on public cloud environments than on-premises environments. Accordingly, there has been insufficient sharing of actual implementation cases or operational experiences applying Terraform when building Kubernetes clusters in on-premises environments. Furthermore, whether static security audit tools like tfsec produce the same level of security evaluation results when Terraform providers differ has not been clearly verified. To compare and analyze these points, this study analyzed three providers available for building private Kubernetes clusters based on on-premises environments using Terraform, along with their code.

Table 1 summarizes the characteristics of code for on-premises environments based on Matchbox, Rancher, and Proxmox. All three environments (Matchbox, Rancher, Proxmox) can automate Kubernetes cluster creation through Terraform. Each provider shares the commonality of defining the provider block in Terraform and performing provisioning by accessing VMs via SSH to configure nodes. That is, all follow the same flow of defining infrastructure as code and building Kubernetes clusters using Terraform. However, differences exist in infrastructure configuration methods and implementation details as they use different providers.

3.1.1 Matchbox-based configuration

In the node creation phase, libvirt provider is used to create PXE-bootable VMs. Subsequently, when PXE booting is performed, Matchbox delivers Ignition Config to install Fedora CoreOS as the node configuration method [17]. When creating clusters based on this, bootkube inside the Typhoon module is used to configure bootstrap nodes to automatically join the cluster. Node addresses are collected and managed by identifying nodes based on MAC addresses, after which Matchbox automatically assigns appropriate Ignition Config to each node. Additionally, authentication files such as .matchbox/ca.crt, .matchbox/client.crt, and .matchbox/client.key are required for VM execution. The overall

configuration process is relatively complex as it includes Ignition configuration, PXE environment setup, and external server configuration.

3.1.2 Rancher-based configuration

When creating nodes, VMs are automatically created by specifying the count value for `libvirt_domain` and `k8s_node` resources. After creating VMs based on QCOW2 images, initial configuration is applied via cloud-init to configure nodes. Cluster creation is performed using RKE, and necessary components are installed via SSH based on each node's role (control plane, etcd, worker) [18]. Node addresses are collected by extracting IPs of each VM from `locals.node_ips`, storing them in array form, and then passing them to RKE. Rancher-based configuration requires VM OS images and SSH keys (`id_rsa`, `id_rsa.pub`), and the overall configuration procedure is relatively simple.

3.1.3 Proxmox-based configuration

In Proxmox-based systems, virtual machine environments can be configured using cloud-init supported VM templates from PVE (Proxmox Virtual Environment), and node creation and initial configuration can be automated through integration with Terraform. When creating clusters, after VM initialization, clusters are configured by executing `kubeadm init` and `kubeadm join` via SSH or Ansible [19]. Node addresses are collected by setting fixed IPs through cloud-init, then querying actual IPs via Proxmox API and storing them in Terraform output. Required external elements include Proxmox VM templates, API tokens (`api-token`, `api-token-id`, `api-url`), and SSH keys (`id_rsa.pub`). While the configuration itself is relatively simple, considering that Proxmox VE pre-configuration is essential, the overall complexity can be rated as above average.

As a result of comparing the three on-premises environments, we confirmed that a common structure exists in the Kubernetes cluster automation process using Terraform. For comparison with public cloud and Kubernetes cluster environments, this study designed and implemented the proposed testing framework based on the Matchbox configuration among the introduced on-premises environments.

Table 1: Comparison of On-Premises Terraform Providers

	Matchbox	Rancher	Proxmox
Node creation	libvirt PXE-boot VMs	libvirt_domain / k8s_node	PVE cloud-init templates
Cluster bootstrap	Typhoon / bootkube	RKE via SSH	kubeadm init/join
Node addressing	MAC-based, Ignition	locals.node_ips array	cloud-init fixed IP + API query
Required assets	CA/client certs	OS image, SSH keys	API tokens, SSH key
Relative complexity	High	Low	Above average

4 PROPOSED TESTING FRAMEWORK

This study designed an extensible testing framework for Terraform-based infrastructure security validation. Existing IaC security analysis techniques have limitations in consistently reproducing and comparing identical vulnerability scenarios due to structural and implementation differences across platforms. To address this, the framework is designed to consider the platform dependencies of input code, reconfigure it according to experimental objectives, normalize it into platform-executable code, and then perform security analysis. This approach complements the broader vision of automated cloud security systems [9], which emphasize the need for systematic validation and response mechanisms. While the patent focuses on runtime anomaly detection and response, our framework addresses the proactive validation of infrastructure code before deployment—a critical upstream step in the DevSecOps pipeline. Fig. 6 shows the overall framework structure, and the main components are as follows:

In the Input phase, arbitrary Terraform IaC code not restricted to specific formats or platforms is used as input. This input code can accommodate various platform-based configurations including public cloud, Kubernetes, and Matchbox, and is subsequently refined and restructured in the contextualization phase to meet experimental purposes.

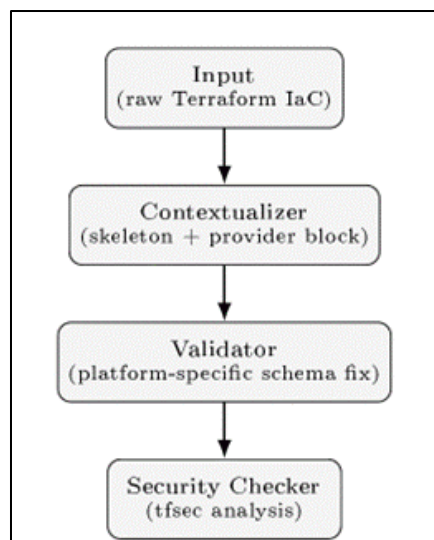


Figure 6: Framework Overview

In the Contextualization phase, minimal skeleton-form Terraform IaC code for experiment execution is generated. This process extracts and rearranges only the core resource definitions and components necessary for reproducing vulnerability scenarios based on the IaC code input in the previous phase, refining it into a standardized structure that enables cross-platform comparison. The code generated here includes only simple resource definitions, designed to facilitate the injection of security vulnerabilities in subsequent phases. This allows focusing on specific security scenarios by eliminating unnecessary complexity. Additionally, platform-specific (provider-specific) differences are reflected in the input skeleton tf code. The provider block applying the platform is configured according to platform requirements. After this phase, the code is transformed into contextualized IaC.

Each platform environment requires different resource models and required fields. The Validator reflects these platform-specific differences to ensure that identical security vulnerability configurations are transformed into platform-specific tf code. By passing contextualized IaC through this phase, inconsistencies that may arise from default values or resource relationships across platforms can be resolved, and the code is transformed into executable validated IaC.

Finally, the Security Checker uses tfsec to perform static security analysis on validated IaC. Through this, vulnerability detection status, detection accuracy, and detection scope can be evaluated, and how the same security scenario is detected differently across platforms can be compared.

4.1 Contextualizer

Contextualization is a component that provides a manual for substituting and generating minimal skeleton IaC required for experiments by combining existing tf code with execution environment in a platform-specific manner. It is written so that identical scenarios have equivalent intent across each platform (AWS public cloud, Kubernetes, on-premises). To this end, provider blocks and resource/variable schemas are adjusted according to platform-specific context and requirements, and the output prioritizes simplicity for easy vulnerability injection and comparative evaluation in the subsequent Validator-to-Security Checker phase.

First, verified code is collected from official repositories such as GitHub, and the runtime environment is input simultaneously. Then, contextualization modifies and replaces it according to the classification in Table 2 for each platform. Common principles: The code corresponding to terraform { required_providers ... required_version ... } specifies providers and versions and is commonly required. The part needed for node creation in Kubernetes cluster construction is configuring the libvirt provider. When configuring providers for the three platforms, libvirt must be configured as shown in the table.

Platform-specific principles: For public cloud, the core provider is "hashicorp/aws". For Kubernetes, "hashicorp/kubernetes" is the core provider, used for container orchestration environments. For Matchbox, the core provider is "poseidon/matchbox", used together with dmacvica/libvirt for on-premises PXE/bare-metal provisioning. Additionally, in Matchbox environments, the "poseidon/ct" provider is additionally required, needed to support boot configuration generation during provisioning.

Following the table and common/platform-specific principles, contextualized IaC is generated and moves to the Validator. The resource models and schemas for each platform's additional features differ. Simply matching the provider block in a single vulnerable Terraform code causes inconsistencies in schemas, required fields, and resource references, leading to Terraform plan failures. Therefore, code must be written to represent identical vulnerabilities executable on each platform through the two phases of Contextualizer and Validator, and tfsec must be executed on the resulting output for meaningful comparisons.

4.2 Validator

Even when contextualized IaC is generated through contextualization, entering it directly may result in code that does not operate as executable. This is because the resource models and schemas for each platform's additional features differ. That is, when simply changing the provider block in a single vulnerable Terraform code, errors are likely to occur at the Terraform plan stage due to inconsistencies in schema structure, required fields, and resource reference methods. Therefore, this framework restructures code into actually executable forms on each platform through the two phases of Contextualizer and Validator. This process corrects code to meet platform-specific resource structures and requirements while maintaining identical vulnerabilities. By executing tfsec on validated output, security analysis results for identical vulnerabilities across platforms can be compared.

Table 2: Provider Block Configurations by Provider

Provider	Required-providers configuration
Public Cloud (AWS)	aws (hashicorp/aws ~>5.0), libvirt (dmacvica/libvirt ~>0.6)
Kubernetes	kubernetes (hashicorp/kubernetes ~>2.0), libvirt (dmacvica/libvirt ~>0.6)
Matchbox	libvirt (dmacvica/libvirt ~>0.6.14), matchbox (poseidon/matchbox 0.5.0), ct (poseidon/ct 0.13)

Matchbox-based code, unlike other providers, requires the ct Provider. This setting is only needed in Matchbox environments using PXE booting and is unnecessary in AWS or Kubernetes environments. Security Group is essential in AWS environments, while Network Policy settings are required in Kubernetes and Matchbox environments.

Additionally, resource names must be modified differently for each provider. When writing AWS environment-based code, vpc_id must be replaced with the actual VPC ID, and it is advisable to assign different security group names per project for security group settings. For Kubernetes environment-based code, resources requiring modification are primarily namespace and name. For namespace, instead of default, a dedicated namespace should be specified, and unique names should be assigned. Specifying the actual kubeconfig path in the config_path section is also one of the essential modifications. For Matchbox environments, name and namespace must be appropriately modified and specified as in Kubernetes.

Regarding cloud credentials, as shown in Table 4, AWS requires AWS credentials, Kubernetes requires kubeconfig, and Matchbox requires the Matchbox API endpoint. Settings related to Storage volume definition and network resource definition are also provided as validation manuals in Tables 3 and 4.

Table 3: Checklist-Based Provider-Specific Requirements for Validation

Item	AWS	Kubernetes	Matchbox
ct Provider	✗	✗	✓
Security Group	✓	✗	✗
Network Policy	✗	✓	✓
Resource name customization	✓	✓	✓
Cloud credentials	✓	✓	✓
Storage volume definition	✓	✗	✗
Network resource definition	✓	✓	✓

4.3 Security Checker

The Security Checker executes static security analysis using tfsec on validated code with intentionally injected vulnerabilities.

IaC code with vulnerability settings is generated as Validated IaC in executable form for each provider through the Validator phase. It is then verified as actually executable code through terraform init and terraform plan. Static analysis is performed using tfsec on the verified code, and the results are collected and analyzed. Fig. 7 shows an example of tfsec static analysis results, based on which provider detection results can be compared and analyzed.

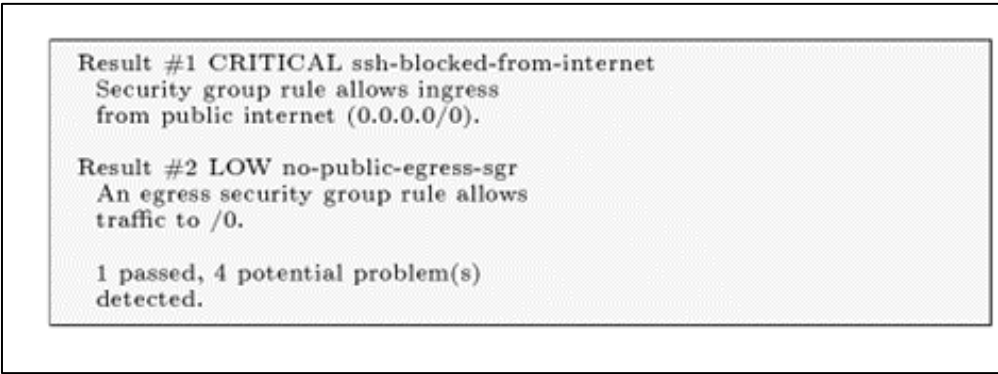


Figure 7: tfsec Static Analysis Output on Validated IaC with Injected SSH Port Exposure (illustrative, reconstructed from reported findings)

The vulnerability injected in this comparison analysis is the configuration that opens SSH ports to the outside. This is one of the representative security vulnerability settings included in the Checks section of tfsec official documentation [8], where SSH ports are opened to 0.0.0.0/0, meaning there are no restrictions on access from all external IPs. For SSH port settings, AWS code was written based on Security Group, while Kubernetes and Matchbox code was written based on NetworkPolicy. Differences in tfsec results appear to stem from these differences. For AWS-based code, "1 passed, 4 potential problem(s) detected" was reported with 2 Critical-level issues and 2 Low-level issues detected. In contrast, for Kubernetes and Matchbox-based code, "1 potential problem(s) detected" was reported with only 1 High-level issue detected. tfsec shows more comprehensive results for AWS than for Kubernetes and Matchbox environments.

Table 4: Details of Validation Checklist Items by Provider

Item	AWS	Kubernetes	Matchbox
Resource name customization	VPC ID	namespace	PXE profile
Cloud credentials	AWS credentials	kubeconfig	matchbox API endpoint
Network resource definition	VPC, Subnet	k8s, CNI, namespace	PXE + CNI
SSH port	aws_security_group	kubernetes_network_policy	kubernetes_network_policy

Table 5: tfsec Detection Results for Identical SSH Misconfigurations Across Different IaC Providers

Provider	Crit.	High	Med.	Low	Total
AWS	2	0	0	2	4
Kubernetes	0	1	0	0	1
Matchbox	0	1	0	0	1

These findings highlight a critical challenge in cloud security: the same security policy (restricting SSH access) is represented and detected differently depending on the underlying platform. This observation is consistent with the motivations behind automated cloud security systems [9], which emphasize the need for consistent security monitoring across diverse cloud environments. The patent's proposed system uses machine learning and contextual analysis to provide consistent threat detection regardless of the underlying infrastructure—an approach that complements our

framework's goal of validating that security tools like tfsec can provide consistent detection across different Terraform providers.

4.4 Performance Analysis

To evaluate the framework's performance, a comparative experiment was conducted. We compared and analyzed the operational success rates of code that followed the manual included in the framework versus code that did not. The experiment was conducted on a total of 11 code samples, and for both versions, Terraform init followed by Terraform plan commands were executed to verify success rates.

Table 6. Comparison by Provider

	Success rate
Manual applied	91%
Manual not applied	0%

The above results demonstrate that code written based on the framework's manual shows significantly higher success rates than code that does not. For code with the framework manual applied, 10 out of 11 codes operated normally, showing approximately 91% success rate. In contrast, for the version with identical vulnerability configurations but without manual compliance, not a single code executed successfully. For example, in Kubernetes-based code without manual application, the resource 'vpc_id' block was included, which does not conform to the Kubernetes provider's resource model, causing errors at the terraform init and terraform plan stages. When this block was removed according to the manual and the code was modified to a resource structure appropriate for the Kubernetes environment, normal operation was confirmed. This suggests that the framework proposed in this study can significantly contribute to improving consistency and operational reliability of code writing across platforms. In particular, it was demonstrated that errors occurring during terraform init and terraform plan execution can be prevented in advance, verifying that code execution success rates substantially improve.

5 CONCLUSION

This study analyzed the limitations of security validation using Terraform and tfsec in IaC (Infrastructure as Code) environments and proposed an extensible testing framework to address these limitations. Prior to proposing the framework, experiments confirmed that Terraform contains security risks by applying intentionally injected vulnerable code without any warnings. Additionally, even when code contains identical security vulnerabilities, different providers resulted in different detection outcomes from tfsec. This suggests that the current static analysis tool tfsec does not sufficiently reflect the environmental differences across various platforms. Accordingly, the framework proposed in this study consists of three phases—Contextualizer-Validator-Security Checker—that correct platform-specific differences and facilitate code transformation to reproduce identical vulnerability scenarios, enabling comparative evaluation using tfsec. Through this, provider-specific detection results for identical vulnerability scenarios can be systematically verified, providing a foundation for establishing reliable security validation systems across IaC-based cloud and on-premises environments. Ultimately, this presents a first step toward standardization of IaC security validation.

The proposed framework also complements the broader vision articulated in recent cloud security patents [9]. While the patent describes systems for runtime detection and automated response to security anomalies, our framework addresses the upstream validation of infrastructure code. Together, these approaches represent a comprehensive security strategy: validating infrastructure configurations before deployment (our framework) and continuously monitoring for anomalies during runtime (the patent's system). Both are essential for maintaining robust security in modern cloud environments.

The ultimate goal of this paper was to systematically identify differences in security validation results across platforms in IaC environments and establish an experimental basis for comparing identical vulnerability scenarios through consistent procedures. To this end, intentional vulnerability injection experiments were conducted to confirm the limitations of security validation in Terraform-based configurations, and tfsec detection deviations based on platform characteristics were empirically analyzed. Furthermore, it was verified through the proposed framework that identical vulnerability scenarios can be reproduced and compared after correcting platform dependencies. As follow-up research, we plan to conduct large-scale experiments using public IaC datasets to evaluate the utility of the proposed framework.

REFERENCES

- [1] A. Singh, "Terraform Infrastructure Code Explained," AST Consulting. [Online]. Available: <https://astconsulting.in/terraform/terraform-infrastructure-code-explained>
- [2] J. E. Lee, S. R. Kim, C. R. Han, H. J. Lee, and H. J. Kim, "A Study on Development of an IaC-Based Cloud Infrastructure Automation Tool and Security Enhancement Methods," *Journal of The Korea Institute of Information Security and Cryptology*, vol. 34, no. 6, pp. 1273–1282, 2024.
- [3] HashiCorp, Terraform. [Online]. Available: <https://developer.hashicorp.com/terraform>
- [4] Amazon Web Services, AWS CloudFormation. [Online]. Available: <https://aws.amazon.com/cloudformation/>
- [5] J. Roper, Spacelift, "What Is an Azure ARM Template? Overview, Tutorials & Examples." [Online]. Available: <https://spacelift.io/blog/arm-template>
- [6] Firefly, "The State of IaC 2024." [Online]. Available: <https://www.firefly.ai/state-of-iac-2024>
- [7] Aqua Security, tfsec Documentation (Latest). [Online]. Available: <https://aquasecurity.github.io/tfsec/latest/>
- [8] Aqua Security, tfsec Documentation (Latest), SSH Blocked From Internet Check. [Online]. Available: <https://aquasecurity.github.io/tfsec/latest/checks/azure/network/ssh-blocked-from-internet/>
- [9] H. Mistry, A. Goswami, and C. Mavani, "Automated Anomaly Detection and Response System for Enhancing Cloud Security" (Patent), Zenodo, 2024. [Online]. Available: <https://doi.org/10.5281/zenodo.18778285>
- [10] Typhoon, Matchbox. [Online]. Available: <https://typhoon.psdn.io/>
- [11] Amazon Web Services, "Amazon Machine Images in Amazon EC2." [Online]. Available: <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html>
- [12] F. Dinu, Spacelift, "How to Manage On-Premise Infrastructure with Terraform." [Online]. Available: <https://spacelift.io/blog/terraform-on-premise>
- [13] SUSE, Rancher. [Online]. Available: <https://www.rancher.com/>
- [14] Proxmox. [Online]. Available: https://pve.proxmox.com/wiki/Main_Page
- [15] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, "Security Misconfigurations in Open Source Kubernetes Manifests: An Empirical Study," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, Art. 99, 36 pp., 2023.
- [16] M. I. Shamim, F. A. Bhuiyan, and A. Rahman, "XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices," in *Proc. 2020 IEEE Secure Development (SecDev)*, IEEE Computer Society, Los Alamitos, CA, USA, 2020, pp. 58–64.
- [17] Poseidon, "Typhoon — Kubernetes Distribution," Poseidon Systems. [Online]. Available: <https://typhoon.psdn.io/>
- [18] HashiCorp, "Rancher Provider," Terraform Registry. [Online]. Available: <https://registry.terraform.io/namespaces/rancher>
- [19] HashiCorp, "Proxmox Documentation," Terraform Registry. [Online]. Available: <https://registry.terraform.io/providers/Telmate/proxmox/latest/docs>