IJARCCE A. ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 9, September 2026

DOI: 10.17148/IJARCCE.2026.15921

XplainGuard: Dynamic Policy Enforcement in Cloud Infrastructure as Code: Integrating Generative AI for Auto-Remediation and Explainability

Aadrika Khare[1], Poonam Bhartiya[2] & Shailendra Shriwastava[3]

M.Tech Scholar, Department of Computer Science and Engineering,

Babulal Tarabai Institute of Research & Technology (BTIRT)Rajiv Gandhi Proudyogiki Vishwavidyalaya (RGPV), Bhopal[1]

Guide, Department of Computer Science and Engineering, Babulal Tarabai Institute of Research & Technology (BTIRT)Rajiv Gandhi Proudyogiki Vishwavidyalaya (RGPV), Bhopal[2,3]

Abstract: Infrastructure as Code (IaC) concentrates cloud security posture into text files, where one misconfigured attribute propagates to every environment where it is applied. Four gaps persist in prior tooling: static scanners such as Checkov and Trivy detect misconfigurations but generate no corrective code, leaving a detection-to-repair gap; generative repair methods report incorrect-code rates as high as 20.4%, precluding unattended use; automated fixes carry no security rationale, so practitioners cannot judge why a change is needed; and existing tools evaluate misconfigurations in isolation, missing compound attack paths formed by co-located resources. This paper presents XplainGuard, a command-line Terraform auditor that targets all four directly: model output is constrained to a strict JSON schema; every generated patch passes through a self-verification guardrail that executes terraform validate before the developer sees it, with one bounded correction attempt; each finding carries a structured four-element rationale (resource and attribute, attack vector, compound-risk context, compliance control). Across 21 runs on a seven-file fixture, every response parsed as valid JSON, and the guardrail reduced the observed incorrect code rate from 6.7% to 0%. Detection coverage was 100% on isolated vulnerabilities and 91.7% overall, and locally executed scans show Trivy 0.72.0 missing two of the four target categories that Checkov 3.3.8 detects, illustrating the specific gaps this work addresses across prior tooling rather than uniform superiority over any single tool. The evaluation is small; its statistical limits are stated explicitly.

Keywords: Infrastructure as Code, Terraform, dynamic policy enforcement, generative AI, large language models, auto-remediation, explainable AI, cloud security, self-verification.

I. INTRODUCTION

Infrastructure as Code (IaC) replaces manual cloud provisioning with declarative configuration files that are version-controlled alongside application code. HashiCorp Terraform, using the HashiCorp Configuration Language (HCL), is the dominant framework in this space [1], and industry studies report broad adoption alongside persistent operational challenges [16]. The security consequence is dual-directional: security controls become reviewable engineering artefacts, but a single incorrect attribute value propagates identically to every environment where the file is applied. Empirical taxonomies of IaC defects [13] and security smells [6], [14] document the recurring classes of error that arise in practice. Two generations of tooling address this. Pattern-based static analysis tools, Checkov [2] and Trivy [3] among them, evaluate resource attributes against curated rule libraries; Chiari et al. [15] survey this class comprehensively. They are fast and deterministic, but produce no corrective code and evaluate resources largely in isolation. The second generation applies large language models. De Vito et al. [4] demonstrated that LLM-based semantic analysis outperforms rule-based detection for security smells in Ansible, Chef, and Puppet. Low et al. [5] applied LLMs to Terraform repair specifically, and documented the central obstacle: hallucinated fixes.

This paper describes XplainGuard, which targets the specific failure mode Low et al. identify. Its contributions are: (C1) a self-verification guardrail that validates every generated patch with terraform validate before presentation, with one bounded correction attempt, implementing the mitigation proposed but not built in [5]; (C2) a structured four-element explanation (resource and attribute, attack vector, compound risk, compliance control) attached to every finding; and

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 9, September 2026
DOI: 10.17148/IJARCCE.2026.15921

(C3) an empirical evaluation of 21 measured runs, with all raw records retained, and an independently executed comparison against Checkov 3.3.8 and Trivy 0.72.0.

II. LITERATURE REVIEW

A. Classification Dimensions

Prior IaC security tools are separated along four dimensions: detection approach (pattern matching, structural analysis, policy-as-code, or AI semantic analysis); repair capability (detection only, guidance only, or code repair); context awareness (resource-isolated, cross-resource, or architectural); and explanation quality (minimal rule code, natural language, or structured explainable output).

B. Pattern-Based Static Analysis

Checkov [2] maintains an extensive rule library derived from CIS Benchmarks and related standards. Trivy [3] performs comparable misconfiguration scanning. Both are open-source, free, and widely integrated into CI/CD pipelines. Neither generates corrective code, and both evaluate rules independently, so neither reports a compound relationship between co-located misconfigurations.

C. LLM-Based Detection

De Vito et al. [4] present SecLLM, evaluated against the GLITCH detector [7] across Ansible, Chef, and Puppet oracle datasets. Their strongest configuration reaches average precision 0.89-1.00 and recall 0.90-1.00, exceeding GLITCH by 12-21 percentage points of precision, at $0.003-$0.015 per script. SecLLM is detection-only and does not target Terraform, so its figures are not directly comparable to a Terraform repair system; they are reported here for completeness rather than as a baseline.

D. Automated Program Repair Context

Automated program repair (APR) for conventional languages is a mature field [11]. A recurring hazard is overfitting: a patch that satisfies the validating oracle without preserving intended behavior [12]. This concern transfers directly to IaC repair, where a patch may silence a scanner rule without eliminating the underlying exposure, precisely the failure mode Low et al. [5] observe in 12.2% of their manually validated fixes. LLM-based APR has been examined for C and Python [9] and in end-to-end pipelines combining static analysis with retrieval and a fine-tuned model [10]; both report that richer prompt context improves patch quality. Within IaC specifically, Saavedra et al. [8] propose InfraFix, a technology-agnostic repair framework using SMT-based reasoning and state inference rather than generative models, indicating that non-LLM repair strategies remain an active alternative.

E. LLM-Based Repair

Low et al. [5] are the closest prior work. They feed scanner output and Terraform blocks to GPT-3.5 and GPT-4 and collect repaired code, evaluating on the TerraGoat and KaiMonkey vulnerable repositories containing over 200 misconfigurations. Their reported outcomes are summarized in Table I. Two properties are material to the present work. First, their strongest result, 87.4% of Checkov-detected misconfigurations repaired, requires a human to supply additional context in a second pass; the fully automated first pass reaches 27.4% on Checkov and 44.2% on Trivy. Second, manual validation of 49 Compute-resource fixes found 79.6% correct, with 8.2% carrying syntax errors and 12.2% not addressing the underlying issue. The authors conclude that LLMs are more suited to being used as an assistant rather than a completely automated code repair tool, and propose passing generated output through a language validator as a mitigation for future work.

TABLE I REPORTED REPAIR OUTCOMES OF LOW ET AL. [5] USING GPT-4

Configuration	Checkov	Trivy
Pass 1 (fully automated)	27.4%	44.2%
Pass 2 (human-in-the-loop)	87.4%	67.3%
Correct on manual validation*	79.6%	-
Hallucinated (syntax / no fix)*	8.2% / 12.2%	-

*Manual validation of 49 Compute-resource fixes after the second pass.

IJARCCE A. ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 9, September 2026

DOI: 10.17148/IJARCCE.2026.15921

III. METHODS AND MATERIALS

A. Formal Problem Definition

No existing IaC security tool simultaneously provides compound-risk-aware detection, a reliably error-free automated repair path, and structured, compliance-mapped explanation for every finding. Checkov and Trivy evaluate resources in isolation and generate no corrective code. LLM-based repair narrows the detection-to-repair gap but introduces a reliability problem: Low et al. [5] report an incorrect-code rate of 20.4% on their strongest fully automated configuration, which precludes unattended use in a DevSecOps pipeline. Rule-based taxonomies such as Rahman et al. [6] evaluate misconfigurations independently, so a set of co-located findings that jointly constitute an attack path is reported as several unrelated, low-severity alerts rather than one compound risk. The problem this work addresses is therefore precisely bounded: design and empirically validate a Terraform auditor that (i) drives its measured incorrect code rate toward 0% through automated self-verification rather than manual review, (ii) evaluates co-located misconfigurations jointly rather than in isolation, and (iii) attaches a structured, compliance-referenced rationale, for example against the CIS AWS Foundations Benchmark [17], to every finding, without requiring paid infrastructure or model fine-tuning.

B. System Architecture

XplainGuard is a Python command-line application. Figure 1 shows the system architecture, in which Stage 4.5 is the self-verification guardrail introduced by this work.

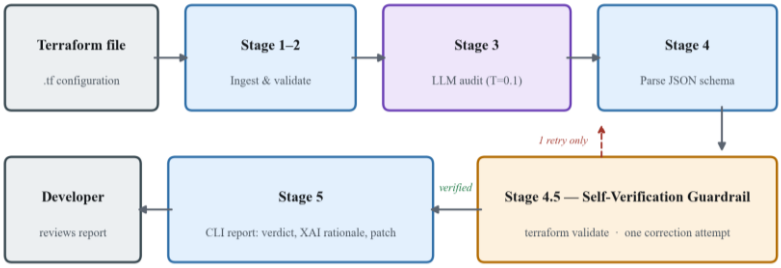

Fig. 1. XplainGuard pipeline. Stage 4.5 is the self-verification guardrail.

C. Bounded Inference and Schema Enforcement

The model is invoked at temperature 0.1 with a system prompt that constrains output to a four-field JSON object: verdict, vulnerability name, explanation, and remediation code. A sanitization step strips markdown fences before parsing. Low temperature reduces stochastic token selection; the schema constrains the response into a machine-parseable form.

D. Self-Verification Guardrail

This is the principal contribution. When the verdict is FAIL and a patch has been produced, the patch is spliced back into a complete copy of the original file, not validated as an isolated snippet, since generated patches routinely reference sibling resources, and terraform validate is executed against it using a pre-warmed provider cache, requiring no network access. Only error-severity diagnostics invalidate a patch; warnings such as deprecated attributes do not.

If validation fails, the exact diagnostic is returned to the model with a request for one corrected patch, which is then validated once more. No third attempt is made. This bound guarantees termination and caps cost at two API calls per file. If the second attempt also fails, the patch is presented with an advisory warning rather than suppressed or shown as clean. Where Terraform is unavailable the routine returns valid, so the guardrail degrades gracefully rather than blocking the scan.

E. Structured Explanation Output

Each finding carries a four-element rationale: the specific resource and misconfigured attribute; the concrete attack vector enabled; interaction with co-located resources; and the violated compliance control. This addresses the absence of security rationale in the repair output of [5].

F. Experimental Setup

Evaluation used seven purpose-built Terraform files: four containing a single target vulnerability each (public S3 ACL, public EC2 IP, unencrypted EBS root volume, wildcard IAM policy), one compound file containing all four, and two clean files with no target-category vulnerability. Vulnerability selection follows the categories enumerated in the CIS AWS Foundations Benchmark [17]. Each file was scanned three times, giving 21 runs. The model was llama-3.3-70b-versatile [18], accessed through the Groq free tier at temperature 0.1. Baselines were Checkov 3.3.8 and Trivy 0.72.0, executed locally on the same files. All raw run records are retained.

IJARCCE A. ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 9, September 2026

DOI: 10.17148/IJARCCE.2026.15921

TABLE II PERFORMANCE METRIC DEFINITIONS

Metric	Definition	n
JPSR	Responses parsed as valid JSON	21
ICR1	Patches failing initial validation	15
ICR2	Patches invalid at presentation	15
ARSR	Patches valid and correctly fixing	15
DC	Target vulnerabilities named	24
FAR	Findings on clean files	6

IV. RESULTS AND DISCUSSION

A. Measured Results

Table III and Figure 2 report the measured outcomes. Two Incorrect Code Rates are distinguished: ICR1 is measured on the model's first output, before the guardrail acts, and ICR2 on what the developer finally receives. The difference between them is the effect of the guardrail and is the quantity of interest.

TABLE III MEASURED RESULTS ACROSS 21 RUNS

Metric	Result	Basis	Note
JPSR	100%	21/21	no parse failures
ICR1	6.7%	1/15	before guardrail
ICR2	0%	0/15	after guardrail
ARSR	93.3%	14/15	first attempt
DC (isolated)	100%	12/12	single-vuln files
DC (compound)	83.3%	10/12	4-vuln file
DC (overall)	91.7%	22/24	all files
FAR	0%	0/6	scoped, see IV-F

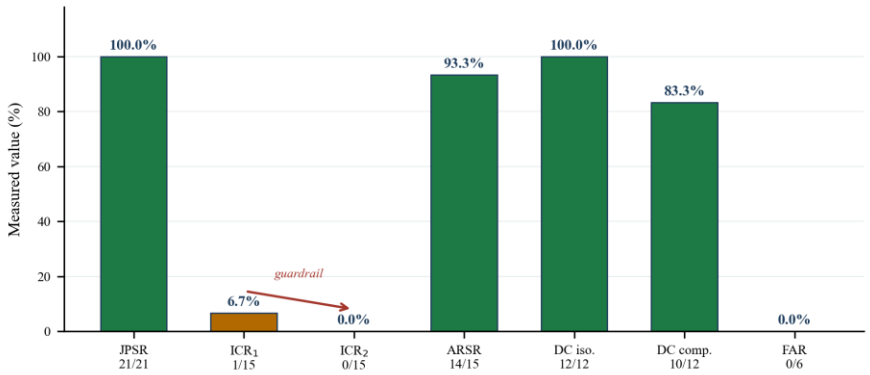

Fig. 2. Measured metric values across 21 runs.

B. Comparative Detection Against Baseline Scanners

Figure 3 shows detection of each target category by the three tools. Checkov reported the corresponding check for all four. Trivy reported the relevant check for the S3 ACL and the unencrypted volume, but emitted no check identifying the public IP on the EC2 instance, only an unrelated Instance Metadata Service finding, and reported no failed check at all on the wildcard IAM policy file.

IJARCCE A. ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 9, September 2026

DOI: 10.17148/IJARCCE.2026.15921

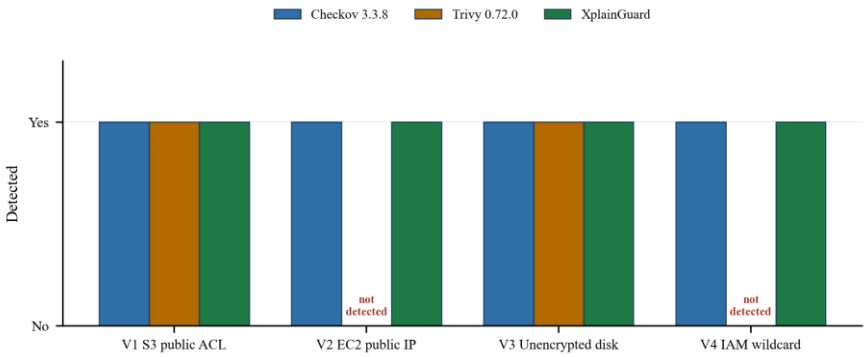

Fig. 3. Target-category detection by tool (locally executed scans).

C. Effect of the Self-Verification Guardrail

Guardrail behavior across the 15 runs that produced a patch is summarized in Figure 4. Fourteen patches (93.3%) passed validation on first generation. One did not: on the compound file the model referenced a security group it had not declared, producing an undeclared-resource error. The guardrail returned that diagnostic to the model, and the corrected patch, which defined the missing resource, passed validation. The guardrail therefore reduced the observed incorrect code rate from 6.7% to 0% on this sample, which is the mechanism's intended effect measured directly.

Detection coverage is not uniform. On the four isolated files the model named the target vulnerability in all 12 trials. On the compound file, where four vulnerabilities are present simultaneously, the narrative explanation named the unencrypted root volume in only one of three trials, giving 10 of 12 vulnerability-mentions and an overall coverage of 91.7% across all files. Notably, the generated patch corrected the encryption setting in all three trials even when the explanation omitted it, indicating a reporting limitation of the single-finding output schema rather than a detection failure.

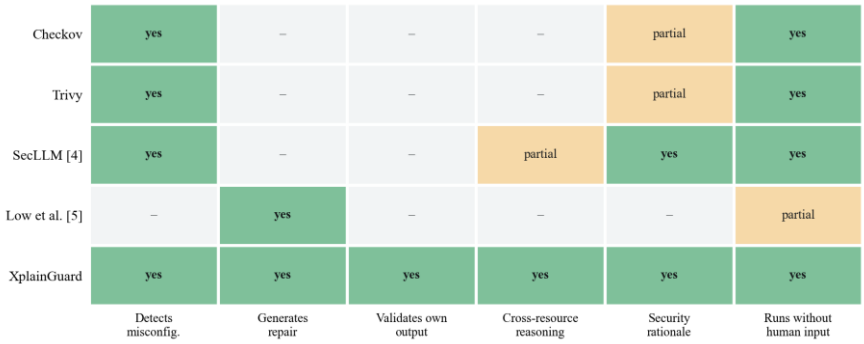

Fig. 4. Share of generated patches judged valid. The two systems are evaluated on different corpora and metrics and are shown side by side for context, not as a direct benchmark.

V. DISCUSSION

A. Positioning Against Prior Repair Work

A direct numeric comparison against [5] would not be methodologically sound: their figures count scanner alarms cleared across two real vulnerable repositories with human assistance, whereas the present ARSR counts patches passing terraform validate and correcting the flagged attribute on a controlled fixture without assistance. The metrics and the datasets differ. The defensible comparison is therefore by capability. Figure 5 places all reviewed systems on a common capability axis, and Table IV details the contrast with the closest prior repair system.

IJARCCE A. ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 9, September 2026

DOI: 10.17148/IJARCCE.2026.15921

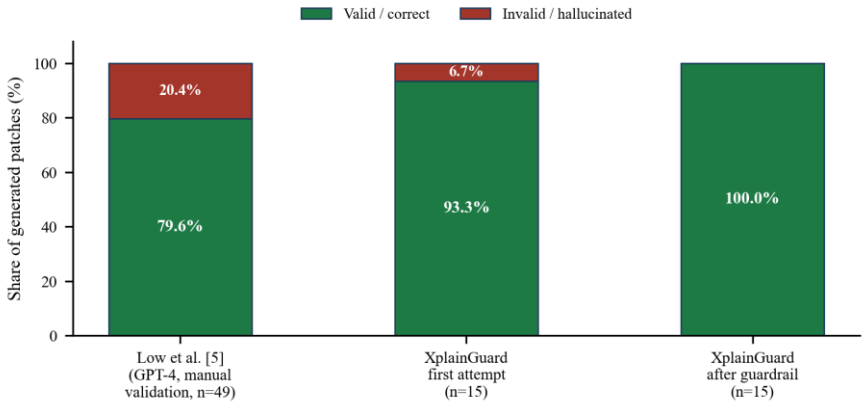

Fig. 5. Capability coverage across reviewed systems. Scanner rows reflect locally observed behavior; [4] and [5] reflect their published descriptions.

TABLE IV CAPABILITY COMPARISON WITH THE CLOSEST PRIOR REPAIR SYSTEM

Aspect	Low et al. [5]	XplainGuard
Human involvement	Required for best result	None
Output validation	Proposed as future work	Implemented
Cross-resource view	Blocks evaluated in isolation	Whole file
Security rationale	Not produced	Four elements
Evaluation scale	200+ misconfigs, 2 repos	21 runs, 7 files

B. Interpretation of the Guardrail Result

The single corrected case is the substantive finding, not the aggregate percentages. It is a concrete instance of the hallucination class documented in [5], generated code that is syntactically plausible but references an undeclared resource, detected automatically and repaired without human involvement. This supports the authors' own hypothesis that a language validator can reject such outputs.

A limitation of the mechanism should be stated alongside its benefit. Validation confirms that a patch parses and resolves its references; it does not confirm that the patch preserves the developer's intent or fully eliminates the exposure. This is the overfitting problem identified in the APR literature [12], and it applies here: a patch that satisfies terraform validate while silencing rather than resolving a security issue would pass the guardrail. Semantic correctness in this evaluation was assessed separately against the known ground truth of each fixture file.

C. Threats to Validity

The evaluation set is small and purpose-built, and the reported figures should be read accordingly. Zero post-guardrail failures in 15 patch trials gives, by the rule of three, an upper bound of approximately 20% on the true post-guardrail failure rate at 95% confidence; ICR2 = 0% is an observation on this sample, not evidence of a defect-free system. Equally, a single observed pre-guardrail failure is too few to characterize the baseline rate precisely: the 6.7% figure carries wide uncertainty and should not be compared directly against the 20.4% hallucination rate reported in [5], which was measured on a different corpus, a different model, and a larger sample.

The fixture contains no safe configuration closely resembling a vulnerable one, so the 0% false alarm rate is a property of this fixture rather than a general result; FAR is additionally scoped to the four target categories, and both baseline scanners raise unrelated findings on the clean files. Detection coverage degrades measurably as a file becomes denser, from 100% on isolated vulnerabilities to 83.3% on the four-vulnerability file, which suggests the single-finding schema is the binding constraint on reporting completeness. Finally, the reported figures for [4] and [5] are taken from those papers rather than reproduced locally.

VI. CONCLUSION

XplainGuard implements self-verification for LLM-generated Terraform repairs: each patch is checked with terraform validate before presentation, with one bounded correction attempt. Across 21 measured runs, 93.3% of patches were valid on first generation and the guardrail reduced the observed incorrect code rate from 6.7% to 0%. Detection coverage was 12 of 12 (100%) on isolated vulnerabilities and 10 of 12 (83.3%) on the compound file, where the single-finding schema caused one corrected issue to go unreported. The mechanism is the mitigation proposed but not built by Low et al. [5],

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 9, September 2026
DOI: 10.17148/IJARCCE.2026.15921

and it operates without the human-in-the-loop step their strongest result depends on. The evaluation is small; extending it to established vulnerable corpora such as TerraGoat [19] and KaiMonkey, and to a multi-finding output schema, is the immediate next step.

ACKNOWLEDGMENT

This work received no external, institutional, or industry funding; it was conducted as part of the author's M.Tech thesis research using free-tier access to the Groq API. The author declares no conflict of interest. The study involved no human participants, animal subjects, or personally identifiable data; all Terraform files used for evaluation were synthetic fixtures with non-functional, illustrative credentials, and no production infrastructure was accessed. The datasets analysed during this study, including the seven-file evaluation fixture, all 21 raw run records, and the Checkov and Trivy scan outputs, are available from the author on reasonable request. The author thanks her supervisors, Dr. Poonam Bhartiya and Shailendra Kumar Shrivastava, for their guidance throughout this work.

REFERENCES

[1] HashiCorp Inc., "Terraform: Infrastructure as Code - Official Documentation," 2024. [Online]. Available: https://developer.hashicorp.com/terraform
[2] Prisma Cloud / Bridgecrew, "Checkov: Policy-as-code for infrastructure," version 3.3.8, 2024. [Online]. Available: https://www.checkov.io
[3] Aqua Security, "Trivy: Comprehensive security scanner," version 0.72.0, 2024. [Online]. Available: https://trivy.dev
[4] G. De Vito, F. Palomba, and F. Ferrucci, "SecLLM: Enhancing Security Smell Detection in IaC with Large Language Models," IEEE Access, 2025, doi: 10.1109/ACCESS.2025.3637505.
[5] E. Low, C. Cheh, and B. Chen, "Repairing Infrastructure-as-Code using Large Language Models," in Proc. 2024 IEEE Secure Development Conference (SecDev), Pittsburgh, PA, USA, 2024, pp. 20-27, doi: 10.1109/SecDev61143.2024.00008.
[6] A. Rahman, C. Parnin, and L. Williams, "The Seven Sins: Security Smells in Infrastructure as Code Scripts," in Proc. IEEE/ACM 41st Int. Conf. on Software Engineering (ICSE), 2019, pp. 164-175.
[7] N. Saavedra and J. F. Ferreira, "GLITCH: Automated Polyglot Security Smell Detection in Infrastructure as Code," in Proc. 37th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE), 2022, pp. 1-12.
[8] N. Saavedra, J. Goncalves, M. Henriques, J. F. Ferreira, and A. Mendes, "InfraFix: Technology-Agnostic Repair of Infrastructure as Code," in Proc. 34th ACM SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA), Trondheim, Norway, 2025, doi: 10.1145/3713081.3731735.
[9] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining Zero-Shot Vulnerability Repair with Large Language Models," in Proc. IEEE Symp. on Security and Privacy (SP), 2023, pp. 2339-2356.
[10] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "InferFix: End-to-End Program Repair with LLMs," in Proc. 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE), 2023, pp. 1646-1656.
[11] C. Le Goues, M. Pradel, A. Roychoudhury, and S. Chandra, "Automatic Program Repair," IEEE Software, vol. 38, no. 4, pp. 22-27, Jul. 2021.
[12] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair," in Proc. 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), 2015, pp. 532-543.
[13] A. Rahman, E. Farhana, C. Parnin, and L. Williams, "Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts," in Proc. ACM/IEEE 42nd Int. Conf. on Software Engineering (ICSE), Seoul, Republic of Korea, 2020, pp. 752-764.
[14] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams, "Security Smells in Ansible and Chef Scripts: A Replication Study," ACM Trans. Softw. Eng. Methodol., vol. 30, no. 1, pp. 1-31, 2021.
[15] M. Chiari, M. De Pascalis, and M. Pradella, "Static Analysis of Infrastructure as Code: A Survey," in Proc. IEEE 19th Int. Conf. on Software Architecture Companion (ICSA-C), 2022, pp. 218-225.
[16] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry," in Proc. IEEE Int. Conf. on Software Maintenance and Evolution (ICSME), 2019, pp. 580-589.
[17] Center for Internet Security, "CIS Amazon Web Services Foundations Benchmark, v2.0," 2023.
[18] A. Dubey et al., "The Llama 3 Herd of Models," arXiv:2407.21783, 2024.
[19] BridgeCrew, "TerraGoat: Vulnerable Terraform Infrastructure," 2024. [Online]. Available: https://github.com/bridgecrewio/terragoat