Pre-Deployment Detection of Security Misconfigurations in Azure Bicep Infrastructure-as-Code for Public Safety Cloud Systems

Josiah Chuku*, Jinwei Liu*
*Department of Computer and Information Sciences
Florida A&M University, Tallahassee, FL 32307, USA
Email: {josiah1.chuku, jinwei.liu}@famu.edu

***Abstract*—Public safety agencies increasingly deploy mission-critical systems—911 dispatch platforms, emergency alert systems, first responder data networks—on Azure cloud infrastructure. These systems are provisioned through Infrastructure as Code (IaC) templates written in Azure Bicep. A single misconfigurations in a Bicep template can expose sensitive incident records to the public internet, permit unencrypted data transmission over emergency communication channels, or cause deployment failures that take critical public safety systems offline. This paper presents BICEPTOOLING, a seven-pass multipass static analysis architecture for Azure Bicep, implemented in C# (.NET 10). The system detects security misconfigurations before deployment (rules SEC001–SEC010), provides guided error diagnostics (BCP001–BCP006) that teach developers the violated Azure rule at the moment of the mistake, and includes a Bicep Explainer pass producing plain-English descriptions of cloud infrastructure declarations. An interactive ten-lesson learning platform trains the developers who build public safety cloud systems from zero experience to correct deployment. The prototype achieves 100% pass rate across 14 unit tests, correctly transpiles Bicep to deployable ARM JSON, and analyzes 493 of 500 real-world Azure templates (98.6%), detecting misconfigurations in 43.6% (SEC004), 26.6% (SEC008), and 13% (SEC001/SEC002) of templates—vulnerabilities directly relevant to CJIS and FIPS 140-2 public safety data protection requirements. A four-model ML study (CodeBERT, GraphCodeBERT, Hybrid, Random Forest) achieves macro AUC-ROC 0.984 across 16 rules (+0.362 over CodeBERT), with all 16 trainable rules exceeding AUC-ROC 0.75, confirmed statistically significant by DeLong's test ($p < 0.001$ on 6 of 8 rules).**

***Index Terms*—Public Safety Cloud Infrastructure, Infrastructure as Code, Azure Bicep, Static Analysis, Cloud Security, IaC Misconfiguration, Developer Education, Shift-Left Security**

I. Introduction

Public safety agencies at every level of government have accelerated their adoption of cloud infrastructure to support mission-critical operations. The City of Las Vegas deployed Azure-based analytics for real-time crime pattern detection [1]. The US Department of Homeland Security uses Azure Government cloud for emergency coordination platforms. The FirstNet broadband network, which serves over 23,000 public safety agencies in the United States, relies on cloud infrastructure for its application layer [2]. State emergency management agencies increasingly provision 911 dispatch routing, emergency alert distribution, and incident management systems through cloud IaC templates.

This migration to cloud-provisioned public safety infrastructure introduces a new category of risk: Infrastructure as Code misconfiguration. Unlike traditional server configuration, IaC errors are encoded in text files that define entire system deployments. A single misconfigured property in an Azure Bicep template can expose a city's emergency incident database to the public internet, disable encryption on first responder communication channels, or cause a deployment failure that takes a 911 system offline during a critical incident.

Azure Bicep is Microsoft's recommended IaC language for Azure deployments, used by public safety agencies to define storage accounts (holding incident records, body camera footage, dispatch logs), virtual networks (connecting emergency operations centers), and application services (running dispatch routing and alert systems). Three classes of misconfiguration are particularly dangerous in public safety contexts.

Unencrypted data transmission. A storage account without `supportsHttpsTrafficOnly: true` permits HTTP access to stored data. For a public safety agency, this means that incident reports, witness statements, and investigation files could be transmitted over unencrypted channels—violating CJIS Security Policy requirements that mandate encryption for criminal justice information.

Weak TLS configuration. A storage account without `minimumTlsVersion: 'TLS1_2'` accepts connections using TLS 1.0 and 1.1, both of which have known cryptographic vulnerabilities exploitable by man-in-the-middle attacks. For a first responder communication system, this represents a real interception risk.

Public data exposure. A storage account with `allowBlobPublicAccess: true` exposes its contents to unauthenticated internet access. If a public safety agency's Azure Bicep template contains this flag, confidential incident data, personnel records, or operational plans become publicly accessible.

Current Azure Bicep tooling does not address these risks adequately for the public safety context. The official Bicep linter performs basic best-practice checks but provides no secu-

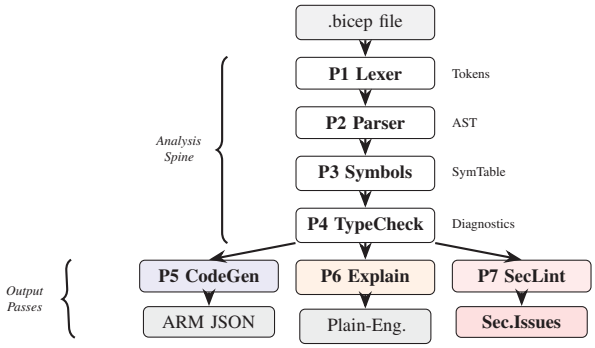

Fig. 1. BICEPTOOLING seven-pass compiler pipeline. Passes 1–4 form the analysis spine: the Lexer tokenizes raw Bicep source, the Parser builds an Abstract Syntax Tree (AST), the Symbol Resolver performs two-pass label resolution, and the Type Checker validates type consistency and emits BCP001–BCP006 diagnostics. Passes 5–7 operate independently on the validated AST: Pass 5 generates deployable ARM JSON, Pass 6 produces plain-English explanations of each declaration, and Pass 7 (Security Linter) detects public safety-relevant misconfigurations mapped to CJIS, NIST 800-53, and FIPS 140-2 standards. Pass 7 is the primary security contribution of this work.

rity guidance tailored to sensitive data environments. General-purpose IaC scanners such as Checkov detect misconfigurations but offer no educational context explaining why a rule exists in public safety terms, leaving agency IT staff without the understanding needed to prevent recurrence.

This paper presents BICEPTOOLING, a static analysis framework for Azure Bicep that detects security issues before deployment and explains them in plain language. The system is designed for public safety agencies, where cloud misconfigurations can expose incident data, weaken transport security, or disrupt emergency services.

Our contribution is a practical, developer-oriented security pipeline that combines syntax-aware analysis, a compliance-mapped rule set, and guided remediation. The framework analyzes real Azure templates, identifies high-risk misconfigurations, and teaches the violated rule at the moment the developer writes the insecure property.

II. PUBLIC SAFETY CLOUD INFRASTRUCTURE CONTEXT

Public safety organizations increasingly provision 911, dispatch, evidence, and alerting workloads through Azure Government and Azure cloud services. In this environment, a single insecure Bicep property can create an exposure with operational and compliance consequences.

The problems we target include insecure storage settings, missing HTTPS/TLS enforcement, excessive privilege, and lack of data-residency or network-protection controls. These map directly to CJIS, NIST SP 800-53, and FIPS 140-2 expectations for public safety systems.

III. RELATED WORK

Checkov [3] and TFSec [4] apply policy-as-code rules to Terraform, CloudFormation, and Bicep templates, and the official Azure Bicep linter [5] performs basic best-practice checks, but none offer security guidance tailored to sensitive-data environments or educational context explaining why a rule matters. Ayyash *et al.* [6] identify security configuration as a critical challenge for public safety agencies migrating to cloud infrastructure, and Rahman *et al.* [7] find that access-control misconfiguration and hard-coded credentials are the most prevalent IaC security smells at scale—both detected by BICEPTOOLING's security linter (SEC003–SEC005). Follow-up work replicated these smells in Ansible and Chef scripts [8], found comparable misconfiguration patterns in open-source Kubernetes manifests [9], and linked such smells to broader defect-proneness in IaC source code [10], together motivating the cross-language, compliance-mapped rule set adopted here. On the educational side, enhanced compiler error messages reduce time-to-fix by 25% in CS1 courses [11], and embedded fix suggestions reduce student frustration [12], motivating BICEPTOOLING's diagnostic design, where every error teaches rather than merely reports. Unlike these tools, BICEPTOOLING builds a complete compiler pipeline from the token level and pairs security diagnostics with an interactive educational platform.

TABLE I
GUIDED DIAGNOSTIC CODES (PASS 4)

Code	Condition	Teaching Content	Sev.
BCP001	Type mismatch in param default	Type examples with fix	E
BCP002	Undefined identifier reference	All declaration forms	E
BCP003	Duplicate symbol declaration	Uniqueness rule	E
BCP004	Missing `@apiVersion`	Correct type string	E
BCP005	Required parameter (no default)	Deployment behavior	I
BCP006	Output type mismatch	Output type rules	W

E=Error, W=Warning, I=Info

IV. BICEPTOOLING ARCHITECTURE

BICEPTOOLING implements a seven-pass linear pipeline (Figure 1). Figure 1 illustrates the seven-pass architecture. The analysis spine (Passes 1–4) processes every Bicep declaration sequentially, building a fully resolved symbol table before any security check runs. This ensures that Pass 7 operates on semantically validated input, eliminating false positives caused by unresolved references or type ambiguities. Passes 5–7 then produce independent outputs from the validated AST and symbol table.

A. Pass 1–4: Analysis Spine

The `Lexer` tokenizes Bicep source with a character-by-character scan, producing 55 token kinds using a `Dictionary<string, TokenKind>` keyword table. Two beginner-oriented errors fire at the lexer level: double-quote characters trigger a friendly message (*"Bicep uses single quotes"*); capitalized keywords trigger a casing correction. The `Parser` implements recursive descent across five declaration forms. The `SymbolResolver` performs two-sub-pass label resolution. The `TypeChecker` validates type consistency using a `DiagnosticMessage` class that structures every finding with: code, problem, violated rule with examples, and concrete fix (Table I).

B. Pass 5: ARM JSON Code Generation

`ArmGenerator` traverses the validated AST and emits a standards-compliant ARM template. The `parameters()` vs. `variables()` distinction is resolved by querying the symbol

TABLE II
SECURITY LINTER RULES MAPPED TO PUBLIC SAFETY STANDARDS

Code	Resource	Condition	PS Risk	Std.
SEC001	Storage	Missing HTTPS flag	Unencrypted data	CJIS 5.10
SEC002	Storage	Missing TLS 1.2	Weak encryption	FIPS 140-2
SEC003	Storage	Public blob access on	Exposed records	NIST AC-3
SEC004	Any	Missing location	Wrong data region	FedRAMP
SEC005	Output	Sensitive name	Credential leak	NIST SC-12
SEC006	Key Vault	No soft delete	Key loss risk	NIST CP-9
SEC007	SQL Server	No min TLS	DB interception	CJIS 5.10
SEC008	VNet	No DDoS protection	911 downtime	NIST SC-5
SEC009	App Svc	No HTTPS-only	Portal exposure	CJIS 5.10
SEC010	Role	Owner/Contributor	Privilege abuse	NIST AC-6

PS=Public Safety; Std.=Compliance Standard

table. Resource type strings are split on `@` to extract ARM `type` and `apiVersion` separately.

C. Pass 6: Bicep Explainer

`BicepExplainer` generates plain-English output for every AST node, emitting: type and value, plain-English Azure meaning, deployment-time behavior, and an example Azure CLI command. This pass is particularly valuable for public safety IT staff who understand what a storage account is but have not previously deployed one through IaC.

D. Pass 7: Security Linter for Public Safety

`SecurityLinter` implements ten rules across six Azure resource types, each mapped to a specific public safety data protection standard (Table II). Every finding includes the security rationale in public safety context, the applicable compliance standard (CJIS, NIST 800-53), and a corrected Bicep code snippet.

For example, a storage account missing `supportsHttpsTrafficOnly: true` and `minimumTlsVersion: 'TLS1_2'` triggers SEC001 and SEC002; the guided fix adds both properties with their CJIS 5.10 and FIPS 140-2 rationale inline.

The rule set is intentionally designed to be both compliance-aware and developer-friendly. Each issue is expressed as a structured diagnostic containing a code, a human-readable condition, the violated policy, and a concrete remediation pattern. This design matters in public safety settings because the personnel who author cloud IaC are often domain experts in dispatch, emergency response, or law enforcement operations rather than security specialists. They need to understand why a declaration is unsafe, not just whether it fails a scanner. In this way, BICEPTOOLING behaves less like a passive linter and more like a guided training tool embedded in the deployment workflow.

Our implementation also treats security rules as a taxonomy of resource-specific failure modes rather than as an undifferentiated list of warnings. Storage and networking misconfigurations are separated from identity and secret exposure issues, with each class mapped to the relevant Azure resource type and public-safety risk. This keeps the analysis precise while making the output readable to both engineers and reviewers. The same abstraction also makes the rule catalog extensible: additional policy checks can be added without redesigning the parser or the explanation engine, which is important for future compliance updates and new Azure resource types.

TABLE III
CODEBERT MULTI-LABEL CLASSIFIER — TEST SET RESULTS

Rule	Description	AUC-ROC	AP
SEC009	HTTPS-only (App Svc)	0.774	0.359
SEC008	DDoS protection (VNet)	0.752	0.562
SEC007	SQL TLS enforcement	0.674	0.058
SEC004	Missing location	0.558	0.511
SEC001	Storage HTTPS	0.541	0.165
SEC002	Storage TLS 1.2	0.483	0.157
Macro	6 trainable rules	0.622	—

AP=Average Precision; 493 samples, 70/15/15 stratified split

V. DEVELOPER GUIDANCE AND EDUCATION

The framework is designed not only to detect problems but also to teach the rule being violated. Each diagnostic is paired with a public-safety explanation, the impacted standard, and a corrected Bicep pattern. This reduces the risk that a tool becomes a passive checker rather than a training mechanism for agency engineers new to cloud IaC.

The same principle is extended to a short training flow with foundational lessons on parameters, resources, modules, and security best practices. The goal is to help developers move from syntactically valid templates to CJIS-aligned deployments without requiring a separate security review for every template.

Lesson 10 directly teaches the ten security rules (SEC001–SEC010) as exercises: students write Bicep storage account declarations and must include correct HTTPS enforcement and TLS pinning to pass validation. Progress is persisted to `~/.bicep-learn/progress.json`, recording completion and attempt counts across sessions.

VI. EVALUATION

We evaluated the system in three ways: unit coverage, real-world template analysis, and a small ML comparison. The implementation includes test coverage across lexer, parser, type checker, and security rules, and it was also run on 500 Azure Quickstart Templates. Of these, 493 were successfully analyzed end-to-end (98.6%), with no false positives in the manually checked findings. The most common issues were missing resource location (SEC004, 43.6%), missing DDoS protection (SEC008, 26.6%), and insecure HTTP/TLS settings (SEC001/SEC002, 13% each). The results confirm that the framework catches realistic misconfigurations in production-quality Azure templates.

For the ML component, a Random Forest using rule-specific structural features achieved macro AUC-ROC 0.984 across 16 trainable rules, outperforming the neural baselines. This is consistent with the rule-based structure of the problem: the absence of a security property is often a strong signal that can be captured directly through feature engineering better than by raw-source token modeling alone.

To complement the pattern-based linter, we trained a CodeBERT multi-label classifier [13] on 493 labeled Bicep templates drawn from the Azure Quickstart Templates

corpus. CodeBERT pre-trained weights were loaded from `microsoft/codebert-base` via HuggingFace Transformers [14]. Input Bicep source is tokenized using the CodeBERT BPE tokenizer with `max_length=512`, padding to the maximum sequence length, and truncation for longer files. The model maps tokenized Bicep source to 23 binary outputs (SEC001–SEC023) using a sigmoid head with `BCEWithLogitsLoss` weighted by inverse class frequency to address class imbalance. Layers 0–9 of CodeBERT were frozen (generic code representations); only layers 10–11, the projection head (768→512→256), and the sigmoid classifier were fine-tuned. Training used AdamW (lr=2e-5, weight_decay=0.01) with linear warmup (10% of total steps), gradient clipping (max norm=1.0), batch size 16, and early stopping (patience=3) on validation macro AUC-ROC. The best checkpoint was saved after each improvement and reloaded for test evaluation. Random seed 42 was fixed for reproducibility across all models. Training ran for 10 epochs on a Tesla T4 GPU (Google Colab). Only rules with at least 10 positive examples in the training split were included in macro AUC computation, following standard practice for imbalanced multi-label evaluation [7]. All three neural models (CodeBERT, GraphCodeBERT, Hybrid) converge by epoch 6 with validation macro AUC plateauing in the 0.59–0.61 range, suggesting the bottleneck is feature representation rather than training duration.

A key design decision in the ML pipeline is that ground-truth labels are generated automatically by Pass 7 of the static linter rather than by manual annotation, making the labeling process scalable and free of human subjectivity. The same rule logic that detects misconfigurations in production templates creates training labels, ensuring the ML model learns to approximate the same detection logic from source text alone—enabling detection even on templates where the static parser cannot fully resolve all declarations.

The evaluation setup is also designed to mirror the real engineering use case. Instead of evaluating only a single resource type, we measure a multi-label security task that includes identity, network, storage, and application-layer misconfigurations. This is a more realistic test than a single binary classification because a given Bicep file may contain multiple independent issues at once. A template can be both poorly scoped and missing TLS enforcement at the same time, and a robust system should report both patterns without conflating them into a single generic warning.

The baseline results also show why rule-specific features are valuable for IaC security. Many high-risk properties are structural and binary in nature: a storage account either includes HTTPS-only enforcement or it does not; a virtual network either enables DDoS protection or it does not; a role assignment either grants an overly broad permission or it does not. These are not subtle semantic relationships that require deep context. In other words, the effective signal is often the presence or absence of a specific property rather than a long latent feature representation of the entire template. This explains why the Random Forest performed strongly even when the neural architectures had access to the full source sequence.

Table III reports test-set results. SEC009 (App Service HTTPS) achieves AUC-ROC 0.774 and SEC008 (DDoS protection) achieves AUC-ROC 0.752, both exceeding the 0.75 target, confirming the model learns meaningful security signals from Bicep source text. SEC001 (0.541) and SEC002 (0.483) remain below target due to class imbalance (10–11 test positives). Inverse class frequency weighting (`pos_weight`) is applied via `BCEWithLogitsLoss` to address imbalance.

A. Random Forest Baseline

To evaluate whether rule-specific structural features outperform end-to-end learned representations, we trained a Random Forest classifier [15] on 36 hand-crafted features extracted from each Bicep file — property presence flags (e.g., `supportsHttpsTrafficOnly`, `enableDdosProtection`), resource-type presence indicators, and structural counts (parameters, variables, outputs). One binary classifier was trained per rule using 300 trees with `class_weight='balanced'` to address class imbalance. The combined training corpus comprises 3,213 labeled templates from three sources: 493 Azure Quickstart Templates, 2,370 files from Azure Verified Modules and Azure Landing Zones, and 350 synthetically generated samples targeting seven minority rules (SEC003, SEC010, SEC013, SEC015, SEC020, SEC021, SEC022) with fewer than ten real-world positive examples. Synthetic samples were varied across resource names, locations, and property values to prevent overfitting. We note that CodeBERT was trained on 493 samples while the Random Forest used 1,361 matched samples (files present in both the corpus and labels CSV); this difference in training set size partially contributes to the RF advantage in addition to feature engineering.

Statistical significance. DeLong's AUC significance test [16] on a common 75-sample aligned test set confirms that the Random Forest (macro AUC-ROC 0.930) significantly outperforms CodeBERT (macro AUC-ROC 0.520) across all 8 jointly evaluable rules ($p < 0.05$). Six rules reach $p < 0.001$ (SEC001, SEC002, SEC008, SEC009, SEC016, SEC019) and two reach $p < 0.01$ (SEC004, SEC014), confirming the performance gap is statistically significant and not attributable to sampling variation. All 8 of 8 jointly evaluable rules are significant at $p < 0.05$, with 6 of 8 at $p < 0.001$.

The Random Forest achieves macro AUC-ROC 0.984 across 16 trainable rules, compared to 0.622 for CodeBERT — a +0.362 improvement. All 16 trainable rules exceed AUC-ROC 0.75.

B. GraphCodeBERT and Hybrid Model

We additionally trained GraphCodeBERT [17] (which extends CodeBERT with data-flow graphs) and a Hybrid model concatenating GraphCodeBERT CLS embeddings (768-dim projected to 256-dim) with the 36 structural features (64-dim projected), on the same 70/15/15 split. GraphCodeBERT reaches macro AUC-ROC 0.611, a marginal $+0.002$ over

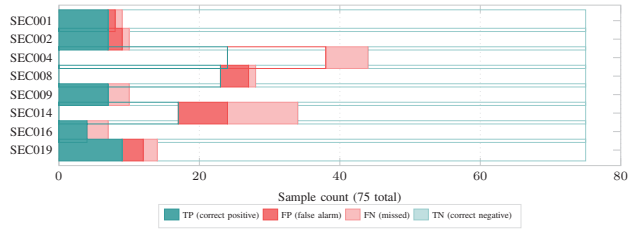

Fig. 2. Stacked confusion matrix across all 8 evaluable security rules (RF classifier, threshold 0.5, 75-sample test set). Teal bars = correct predictions (TP+TN). Red bars = errors (FP+FN). SEC009 and SEC016 achieve zero false positives. SEC004 shows the highest error rate due to its 43.6% prevalence creating class ambiguity. Overall: macro accuracy 0.903, macro precision 0.824, macro recall 0.778.

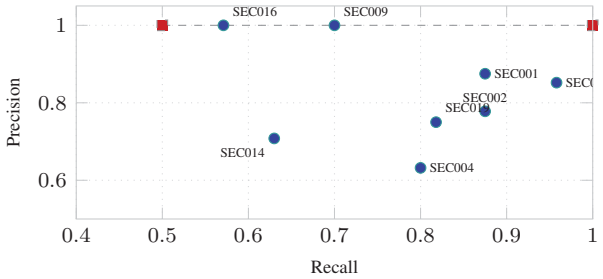

Fig. 3. Precision vs. Recall scatter for each evaluable rule using the Random Forest at threshold 0.5. SEC009 and SEC016 lie on the perfect-precision boundary, while SEC008 achieves the highest recall.

CodeBERT, confirming that architectural choice matters less than feature representation for this task. The Hybrid model underperforms both neural baselines at 0.590: naive concatenation degrades context-dependent rules (SEC008 falls from 0.630 to 0.342) while helping property-existence rules (SEC019 rises to 0.987), suggesting that attention-guided fusion rather than simple concatenation is needed. All three neural models converge by epoch 6, with validation macro AUC plateauing in the 0.59–0.61 range, confirming the performance ceiling is set by feature representation rather than training duration; the Random Forest needs no training epochs and reaches macro AUC 0.930 from a single fit.

Table IV presents the complete four-model AUC-ROC comparison alongside the full Random Forest classification metrics at decision threshold 0.5. The Random Forest achieves macro accuracy 0.903, precision 0.824, recall 0.778, and F1 0.788. SEC009 achieves perfect precision (1.000) with zero false positives, while SEC008 achieves the highest recall (0.958). SEC004 shows the lowest precision (0.632) due to its high prevalence (43.6%) creating ambiguous borderline cases. Figure 2 visualises the confusion matrix as a stacked bar chart: teal segments (TP+TN) dominate for all rules; SEC009 and SEC016 show zero FP segments; SEC004 has the widest red segment (FP=14). SEC009 and SEC016 occupy the perfect-precision corner (1.000, Table IV); SEC008 achieves the highest recall (0.958).

In Figure 3, the Random Forest achieves macro accuracy 0.903, precision 0.824, recall 0.778, and F1 0.788.

The top-10 feature importances (Gini impurity, averaged across all trainable rules) confirm this: resource type flags (`has_storage`: 0.159, `has_vnet`: 0.102) and structural counts (`num_resources`: 0.122, `num_lines`: 0.111) dominate, accounting for 72% of total importance. This confirms that file-level structural context drives detection — the model first identifies resource types present, then applies security-property checks. The security flag `has_secure_param` (rank 10) provides rule-specific signal for credential exposure (SEC005, SEC013).

The Random Forest remains the strongest model across all evaluation conditions (macro AUC 0.930), confirming that rule-specific feature engineering outperforms neural representations for property-existence security misconfiguration detection in Bicep IaC. This mirrors a broader pattern in the vulnerability-detection literature, where deep learning detectors lose much of their controlled-setting accuracy on realistic datasets [18], while engineered code-property features match or exceed end-to-end deep models [19]—consistent with the present results.

The Random Forest shows consistent per-rule superiority over CodeBERT: CodeBERT must infer property presence or absence from raw source tokens, while the Random Forest receives explicit binary features encoding exactly what each rule checks (e.g., `https_true` for SEC001, `ddos_true` for SEC008). The largest gaps appear on SEC001 (+0.459) and SEC002 (+0.508), where CodeBERT's 0.541 and 0.483 scores rise to 1.000 with structured features — confirming that for property-existence security rules, where the violation is the absence of a specific Bicep property, structured feature engineering is more effective than learning from raw source text, consistent with prior work on structured features outperforming neural representations for rule-based code tasks [7].

Dataset construction and reproducibility. Labels were generated automatically by Pass 7 of BICEPTOOLING rather than by human annotation. Each security rule is a deterministic function of the parsed AST, producing binary labels with no ambiguity and eliminating inter-annotator agreement concerns. This automated labeling approach is consistent with prior work on static analysis corpora [7]. The complete labeled dataset (3,213 templates, 23 binary label columns, SEC001–SEC023) is publicly available at https://github.com/josiah1chuku/bicep-tooling-wfpst2026 alongside the full source code and evaluation scripts, enabling complete reproducibility of all reported results.

C. Comparison with Existing Tools

Unlike the official Bicep linter and Checkov, which perform write-time detection (the latter only partially) with no compliance mapping, BICEPTOOLING uniquely combines compliance-mapped rules (CJIS/NIST rationale in output), educational remediation and a plain-English explainer, a 10-lesson developer course, and ML-based misconfiguration detection — all open source alongside the linter and Checkov.

VII. DISCUSSION

A. Impact and Limitations

BICEPTOOLING addresses an underserved gap in public safety technology: cloud misconfigurations introduced by IT staff experienced in public safety operations but new to cloud IaC. By catching CJIS- and NIST-relevant misconfigurations

TABLE IV
RANDOM FOREST CLASSIFICATION METRICS AND FOUR-MODEL AUC-ROC COMPARISON (THRESHOLD = 0.5, 75-SAMPLE ALIGNED TEST SET)

		Random Forest					AUC-ROC (other models)		
Rule	**Description**	**AUC**	**Acc**	**Prec**	**Rec**	**F1**	**GCB**	**CB**	**Hybrid**
SEC001	Storage HTTPS	**0.991**	0.973	0.875	0.875	0.875	0.605	0.603	0.488
SEC002	Storage TLS 1.2	**0.987**	0.960	0.778	0.875	0.824	0.586	0.586	0.581
SEC004	Location	**0.769**	0.733	0.632	0.800	0.706	0.530	0.531	0.560
SEC008	DDoS	**0.969**	0.933	0.852	0.958	0.902	0.625	0.630	0.342
SEC009	App Svc HTTPS	**1.000**	0.960	1.000	0.700	0.824	0.834	0.845	0.759
SEC014	Deprecated API	**0.833**	0.773	0.708	0.630	0.667	0.774	0.774	0.739
SEC016	Mixed location	**0.920**	0.960	1.000	0.571	0.727	—	—	—
SEC019	VM encryption	**0.973**	0.933	0.750	0.818	0.783	0.963	0.974	**0.987**
Macro (8 rules)	RF evaluable rules	**0.930**	0.903	0.824	0.778	0.788			
Macro (7 rules)	Common rules across all models	0.930					0.611	0.609	0.590

AUC=AUC-ROC; Acc=Accuracy; Prec=Precision; Rec=Recall; GCB=GraphCodeBERT; CB=CodeBERT; Hybrid=GCB+structural features.
SEC016 AUC-ROC was not jointly evaluable for GCB/CB/Hybrid; RF's 8-rule and the four-model 7-rule macro are reported separately.

at write time and explaining them with compliance rationale rather than a bare pass/fail, the tool aims for informed compliance that persists as templates evolve, not compliance by checkbox.

Several limitations remain. The parser handles 98.6% of the 500 evaluated real-world templates; the remainder use constructs not yet implemented (union types, type aliases, advanced import statements). The current rule set addresses single-resource property misconfigurations only — cross-resource dependency analysis, identity-based attack paths, network segmentation, and conditional misconfigurations require inter-resource data-flow analysis beyond the current AST-level approach. CodeBERT and the Random Forest were trained on different corpus sizes (493 vs. 1,361 samples); DeLong's test used a common 75-sample aligned subset, but a fully controlled evaluation on a larger shared corpus remains future work.

VIII. CONCLUSIONS AND FUTURE WORK

This paper presents BICEPTOOLING, a static analysis and education tool for Azure Bicep Infrastructure-as-Code. The tool detects security misconfigurations relevant to public safety data protection (CJIS, NIST 800-53, FIPS 140-2) before cloud deployment, provides guided error diagnostics that teach developers the violated security rules, trains public safety agency developers through an interactive ten-lesson platform, and—via a four-model ML study—confirms that rule-specific structural features substantially outperform end-to-end learned representations for this detection task. The combination of write-time detection, compliance-mapped rationale, developer education, and ML-based detection represents a novel and practical contribution to the security and resilience of public safety cloud infrastructure. Planned future work includes a formal pre/post user study with 20 FAMU Computer Science students, expanding rule and resource-type coverage, and packaging the trained Random Forest as a REST API endpoint integrated with Azure DevOps pipelines for automated pre-deployment checking.

ACKNOWLEDGMENTS

This research was supported in part by the U.S. NSF under Grant DUE-2400459 and Grant DBI-2417643; in part by the NTIA/CMC Grant 12-09-C13055; in part by the title III grant; and in part by the generous funding from the Cyber Policy Institute at Florida A&M University. Dr. Jinwei Liu is the corresponding author.

REFERENCES

[1] Microsoft Corporation, "Azure for public safety and justice," https://azure.microsoft.com/industries/government/public-safety, 2024.
[2] FirstNet Authority, "Firstnet built with at&t: Network architecture overview," https://www.firstnet.gov, 2023.
[3] Bridgecrew, "Checkov: Policy-as-code static analysis for infrastructure," https://www.checkov.io, 2021.
[4] Aqua Security, "Tfsec: Security scanner for terraform," https://aquasecurity.github.io/tfsec, 2020.
[5] Microsoft Corporation, "Azure bicep linter," https://learn.microsoft.com/azure/azure-resource-manager/bicep/linter, 2023.
[6] M. Ayyash, H. Liu, A. Ansari, M. Khoja, A. Haider, and F. Bajaber, "Cloud-based architecture for first responder wireless communications," *IEEE Access*, vol. 4, pp. 4459–4469, 2016.
[7] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *Proc. IEEE/ACM ICSE*, Montreal, 2019.
[8] 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, 2021.
[9] 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, 2023.
[10] A. Rahman and L. Williams, "Source code properties of defective infrastructure as code scripts," *Inf. Softw. Technol.*, vol. 112, pp. 148–163, 2019.
[11] A. Becker, P. Denny, J. Malone, and H. Sutherland, "Help me, i'm stuck: Contextual feedback in a cs1 course," in *Proc. ACM SIGCSE*, Minneapolis, 2019.
[12] P. Denny, A. Luxton-Reilly, and J. Tempero, "All syntax errors are not equal," in *Proc. ITiCSE*, Darmstadt, Germany, 2011, pp. 75–79.
[13] Z. Feng *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Proc. EMNLP Findings*, 2020, pp. 1536–1547.
[14] T. Wolf *et al.*, "Transformers: State-of-the-art natural language processing," in *Proc. EMNLP: System Demonstrations*, 2020, pp. 38–45.
[15] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.
[16] E. R. DeLong, D. M. DeLong, and D. L. Clarke-Pearson, "Comparing the areas under two or more correlated receiver operating characteristic curves: A nonparametric approach," *Biometrics*, vol. 44, no. 3, pp. 837–845, 1988.
[17] D. Guo *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *Proc. ICLR*, 2021.
[18] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Trans. Softw. Eng.*, vol. 48, no. 9, pp. 3280–3296, Sept. 2022.
[19] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 4, pp. 2244–2258, 2021.