Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Prevalence of Cross-File Dependencies in Terraform and Their Resolution by Security Scanners

Moatasem M. Draz

Moatasem.draz@fci.kfs.edu.eg

Kafrelsheikh University

Article

Keywords: Infrastructure as Code, Terraform, cross-file dependencies, static analysis, cloud security, security scanning

Posted Date: September 21st, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10885341/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Abstract

Terraform codebases are not flat collections of scripts: modules, variables, and outputs couple files across directories. Yet Infrastructure-as-Code research analyzes such artifacts one file at a time, assuming multi-file reasoning lies beyond current tooling. Both halves of that picture are examined using TerraDS, a public dataset of 62,406 Terraform repositories, from which a module-level dependency graph is reconstructed per repository. Cross-file dependencies are common (34.7% of repositories), highly right-skewed, and dominated by directory-traversing local references, which, with subdirectory references, account for 61.2% of all module dependencies. In 9,035 repositories, such a dependency targets a module declaring security-sensitive resources, most often IAM policies and security-group rules, and this coupling tracks structural complexity (modules rho = 0.752) rather than popularity (stars rho = 0.113). The capability assumption is then tested directly: a controlled experiment probes seven cross-file constructs across four widely used scanners and two security properties. Scanner support is graded rather than binary. Every tool resolves the shortest intra-module flows; only Checkov resolves terraform.tfvars and achieves complete inter-module propagation, while Terrascan resolves the direct module-input case.. The assumption that scanners cannot follow cross-file references is therefore inaccurate for the tools tested. All code and derived data are released.

1. Introduction

Infrastructure as Code (IaC) provisions computing infrastructure through machine-readable definitions, and HashiCorp's Terraform is among the most widely adopted Infrastructure-as-Code tools. As production environments migrate into Terraform, the correctness and security of these definitions become first-order concerns: a single permissive access-control value expressed in code can expose an entire deployment.

Empirical research on IaC quality and security has grown accordingly, security-smell catalogs [1, 2], defect taxonomies [3], polyglot detectors [4], and flow-sensitive analyses [5]. Two methodological assumptions run through this literature. The first is a *unit-of-analysis* assumption: studies collect, measure, and label individual configuration files. The second is a *capability* assumption: that reasoning across files, resolving a variable declared in one file and consumed in another, or following a value through a module boundary, is beyond current scanning tools and therefore a matter for future work [4]. Neither assumption has been examined at scale against real repositories and real tools.

This paper examines both. Its first half is a characterization study: using the 62,406-repository TerraDS corpus, it measures how prevalent cross-file dependencies are, what forms they take, and how often they target modules declaring security-sensitive resources. Its second half is a controlled evaluation: construct by construct, it tests which cross-file values four widely used scanners actually resolve, and finds, contrary to the prevailing assumption, that all of them resolve the simplest cases while differing in how many further indirections they follow. Terraform's override.tf semantics turn out to be handled inconsistently. Three of four tools resolve them for one property and none for the other, which is itself

part of the capability map rather than a single confirmed gap. These dependencies are studied at the module level, meaning those expressed through Terraform module calls, which is the granularity at which cross-file composition is declared. Finer variable- and attribute-level couplings form a complementary surface discussed as future work.

The study is guided by four research questions:

- **RQ1 (Prevalence).** How common are cross-file and cross-module dependencies in real-world Terraform repositories, and how are they distributed?
- **RQ2 (Patterns).** What recurring patterns characterize these dependencies, and which of them cross file and directory boundaries?
- **RQ3 (Security surface).** How frequently do cross-file dependencies reach modules that declare security-sensitive resources, and what repository characteristics are associated with this?
- **RQ4 (Tool capability).** Which composition constructs, intra-module multi-file and inter-module propagation, do widely used IaC scanners actually resolve, and where does resolution break down?

These questions form a single logical chain rather than two separate studies. RQ1 and RQ2 establish that cross-file dependencies are common and are the dominant way Terraform is composed. RQ3 shows that these dependencies routinely target modules declaring security-sensitive resources. RQ4 then asks the question those three make unavoidable, if security-relevant configuration is this often dispersed across files, can the tools meant to catch misconfigurations actually follow it there? The characterization motivates the evaluation, and the evaluation gives the characterization its practical stakes: the prevalence of cross-file structure matters precisely because whether tools resolve it turns out to depend heavily on which tool is used.

Contributions:

- To the best of the author's knowledge, the first large-scale characterization of cross-file dependencies in Terraform at the scale of 62,406 repositories, with statistical analysis of their distribution and correlates.
- A source-based taxonomy of module dependency patterns and their measured prevalence.
- An analysis linking cross-file dependencies to security-sensitive resource types, quantifying the composed configuration surface.
- A controlled, ground-truth evaluation across four scanners, yielding a capability map: cross-file resolution is present in every tool but graded. It is comprehensive in Checkov, limited to the shortest intra-module flows in tfsec and Trivy, and extends to direct module inputs in Terrascan. This corrects the common assumption that scanners uniformly cannot follow cross-file references.
- A public replication package containing all code and derived data.

The negative result is highlighted deliberately. The evaluation was designed expecting to demonstrate a blind spot; the data showed the opposite, and an earlier substring-based analysis that seemed to confirm

the blind spot was traced to a measurement artifact and corrected with a control/treatment design. Reporting this trajectory matters: the commonly repeated assumption, asserting that scanners cannot follow cross-file references, is, for the constructs and tools tested here, inaccurate as a blanket statement: every scanner tested resolves the simplest cross-file mechanisms, and they differ in how many further indirections they follow.

A motivating example, and why it must be tested:

Consider a project split across two files. In variables.tf:

```
variable "bucket_acl" {
  default = "public-read"
}
```

and in main.tf:

```
resource "aws_s3_bucket_acl" "data" {
  acl = var.bucket_acl
}
```

Read in isolation, main.tf contains no insecure literal; the world-readable ACL exists only in the *composition* of the two files. Examples of exactly this shape are routinely offered in the literature as self-evident demonstrations that file-scoped analysis must miss composed misconfigurations. The example is real; the inference is not self-evident. Whether a scanner misses this case is an empirical property of the scanner, not a logical consequence of the file layout. Modern tools may parse a whole directory, build a resource graph, and substitute variable defaults before evaluating rules. This paper therefore treats the example as a hypothesis to test (RQ4), not as a premise, and devotes its first half to the prior question the example presupposes: how much cross-file structure exists in real Terraform to begin with (RQ1 to RQ3).

2 Background and Related Work

2.1 Terraform and the module abstraction

A Terraform configuration is a set of .tf files in the HashiCorp Configuration Language (HCL). Practitioner guidance treats modular decomposition as the default way to structure Terraform at scale [6]. The unit of reuse is the *module*: a directory of configuration instantiated via a module block whose source points to its location, a local path (./modules/network, ../../shared/db), a Terraform Registry reference (hashicorp/consul/aws), or a version-control URL. Files are further coupled by variable references (var.x),

local values, resource-attribute references, and override.tf files whose contents replace earlier definitions under last-wins semantics. The effective configuration of a non-trivial project is thus determined by the composition of many files.

2.2 Security smells and defects in IaC

Rahman et al. established the empirical study of IaC security with seven security smells in Puppet [1, 7], replicated for Ansible and Chef [2], alongside a defect taxonomy [3] and characterizations of defective deployment scripts [8]. A systematic mapping [9] identified security as an under-explored avenue. Sharma et al. and Schwarz et al. examined maintainability smells [10, 11], and a systematic gray-literature review by Kumara et al. distilled practitioner do's and don'ts of infrastructure code [12]. Opdebeeck et al. brought control- and data-flow sensitivity to Ansible smell detection [5, 13], and Rahman and Parnin studied weakness propagation in Puppet [14]. Secret management, a recurrent security concern, was codified into practices by Rahman et al. [15], and co-located insecure coding patterns were characterized by Bhuiyan and Rahman [16]. Quality models have also been defined and validated for Puppet [17], with practical fault detection for the same language explored by Sotiropoulos et al. [18]. Defect prediction from product and process metrics [19] and detection of inconsistencies between code and documentation [20] round out the quality-oriented strand. Throughout, detection and measurement operate on individual scripts, and the multi-file question is deferred.

Beyond smells, a fault-oriented line of work has emerged. Drosos et al. analyzed 360 bugs across Ansible, Puppet, and Chef Ecosystems, providing the first in-depth taxonomy of IaC fault root causes and reproduction requirements, and evaluating the limits of state-of-the-art reliability techniques [21]. Nasiri et al. proposed a complementary taxonomy of Ansible misconfigurations [22], while studies of bug-resolution characteristics across languages give context for how such faults are ultimately fixed [23]. For quality quantification, Begoug et al. released TerraMetrics, an open-source Terraform metrics tool [24], and studied block-level just-in-time defect prediction [25] and module-update practices [26]. These efforts enrich the picture of what can go wrong and how it is measured, but none characterizes the cross-file dependency surface or tests whether scanners resolve it.

2.3 Tooling and the capability assumption

Practical scanning is dominated by rule-based tools such as Checkov, KICS, tfsec, and Terrascan. Academic treatments frequently characterize these as per-file pattern matchers and cite multi-file reasoning as an open problem; GLITCH, the state-of-the-art polyglot detector, analyzes each script against an intermediate representation and lists multi-file analysis as future work [4, 27]. Yet tool documentation indicates that at least Checkov constructs a graph over resources and evaluates variable values. To the author's knowledge no published study has tested, with ground-truth controls, which cross-file constructs such tools actually resolve. The controlled evaluation below provides that test; its outcome contradicts the common characterization.

2.4 Large-scale Terraform studies and corpus bias

Iosif et al. analyzed 8,256 AWS Terraform repositories with three scanners [28]; Verdet et al. studied adoption of security best practices across three clouds [29]. Both note a sampling limitation: filtering GitHub by language:HCL retrieves repositories where HCL dominates, whereas in real projects IaC co-exists with other code and often forms a small fraction of the repository. TerraDS indexes by the presence of Terraform files rather than language dominance, mitigating this bias. Begoug et al. studied module update practices [26]; Iosif et al. also examined cloud-deployment vulnerabilities at scale [28]; and recent work extends the smell lens to sustainability concerns in Terraform [30]. The focus here is the static dependency structure and the tooling that analyzes it.

A parallel strand concerns tool support and threat modeling. Tran et al. proposed TerrARA for automated security threat modeling of Terraform [31]; Sandobalín et al. compared model-driven and code-centric IaC tooling [32]; and Guerriero et al. surveyed industrial adoption and challenges of IaC [33], reporting that tooling maturity and testing remain practitioner pain points [33]. Studies of ecosystem hygiene, semantic-versioning practice in Ansible Galaxy roles [34], evolution and security risks of Helm charts [35], and test quality in Ansible development [36] round out the picture of how IaC artifacts are maintained. None of these evaluates, under controlled conditions, which cross-file constructs a scanner resolves; that is the contribution of RQ4. Tables 1 and 1b position this study against the closest work.

Table 1
Comparison with Closely Related Empirical IaC Studies

Study	Corpus / scale	Unit of analysis	Cross-file	Tool validation
Rahman et al. [1]	1,726 Puppet scripts	Single script	No	—
Rahman et al. [3]	Ansible & Chef	Single script	No	—
Opdebeeck et al. [5]	Ansible (flow)	Role / play	Partial	—
GLITCH [4]	Polyglot, large	Single script	No	Tool (detection)
Iosif et al. [28]	8,256 AWS repos	Resource / file	No	3 scanners (run)
Verdet et al. [29]	≈ 6,071 repos	Policy / file	No	—
This study	**62,406 repos**	**Module graph**	**Yes**	**4 scanners (controlled)**

Table 2 broadens the comparison across the dimensions most relevant to positioning: the IaC technology studied, whether the work targets security, its scale, whether it models any cross-file structure, whether it empirically validates tool behavior, and its core method. The pattern is clear. Security-focused empirical work (Rahman et al.; Iosif et al.; Verdet et al.) operates per file and does not model inter-file structure. Structural and fault studies (Opdebeeck et al.; Drosos et al.) model richer relationships but not the security-relevant cross-file surface at repository scale. Dependency-and-quality tooling (Begoug et al.; TerraMetrics) quantifies modules and updates without linking them to a controlled

security-capability evaluation. No prior study combines large-scale cross-file characterization with a ground-truth test of what scanners resolve, the gap this paper fills.

Table 2
Feature Comparison of Representative IaC Studies

Study	Tech	Security	Scale	Cross-file	Tool valid.	Method
Rahman et al. [1, 3]	Puppet/Ansible/Chef	Yes	1.7k+ scripts	No	No	Smell mining
Rahman & Parnin [14]	Puppet	Yes	Medium	Partial	No	Propagation
Opdebeeck et al. [5]	Ansible	Yes	Large	Partial	No	Data-flow
GLITCH [4, 27]	Polyglot	Yes	Large	No	Tool	IR + rules
Iosif et al. [28]	Terraform/AWS	Yes	8,256 repos	No	3 scanners	Scan mining
Verdet et al. [29]	Terraform/3-cloud	Yes	≈ 6k repos	No	No	Policy adoption
Drosos et al. [21]	Ansible/Puppet/Chef	No	360 bugs	Partial	Tools	Fault study
Begoug et al. [25]	Terraform	No	Large	Modules	No	Defect predict.
This study	**Terraform**	**Surface**	**62,406 repos**	**Yes (primary)**	**4 scanners**	**Graph + controlled eval**

2.5 Empirical evaluation of static analysis tools

Because RQ4 evaluates scanners rather than code, a second literature is relevant: the empirical assessment of static analysis tools. Comparative studies of Java and C analyzers have repeatedly shown that detection capability varies widely between tools on identical inputs [37, 38], that a substantial share of real vulnerabilities is missed entirely [39], and that false positives remain the dominant usability barrier [40, 41]. Developers' expectations of these tools have been surveyed directly [42], and usability criteria catalogued at scale [43]. Particularly relevant to the design used here is the finding that tool *configuration* materially changes measured capability [44]: an evaluation that fixes one configuration may mistake a settings artifact for a capability limit. This motivated the use of matched control, treatment, and inline variants and per-tool signal discovery, so that a tool is judged only against its own baseline behavior.

Two gaps in that literature motivate the contribution made here. First, it targets general-purpose languages; IaC scanners, which reason over declarative configuration and module composition rather

than control flow, are not covered. Second, its unit of evaluation is typically a vulnerability benchmark, which measures *whether a rule exists and fires*, not whether the analyzer can follow a value across the artifacts that define it. Table 3 positions this study against that body of work: the battery used here holds the vulnerability constant and varies only the *delivery construct*, isolating cross-file resolution as the measured property.

Table 3
Comparison with Empirical Tool-Evaluation Studies

Study	Target	What is varied	Measured property
Li et al. [37]	Java SAST	Benchmark programs	Detection rate, precision
Lipp et al. [38]	C analyzers	Vulnerable programs	Detection effectiveness
Charoenwet et al. [39]	SAST in review	Real commits	Issue-identification support
Mordahl & Wei [44]	Taint analysis	Tool configuration	Configuration sensitivity
Guo et al. [40]	SAST warnings	Warning sets	False-positive mitigation
This study	**IaC scanners**	**Cross-file delivery construct**	**Cross-file value resolution**

3 Methodology

Figure 1 summarizes the pipeline: (1) source repositories from TerraDS; (2) reconstruct a module-level dependency graph per repository; (3) classify dependencies by a source-based taxonomy; (4) link cross-file dependency targets to security-sensitive resource types; (5) compute statistics answering RQ1 to RQ3. RQ4 is addressed separately by the controlled experiment reported under Results.

3.1 Data source

TerraDS [45] is a public dataset of open-source Terraform repositories, released with an accompanying dataset paper. It stores extracted metadata rather than raw HCL: repositories, their modules (directory path, providers, and a structured list of module calls), and the resources each module declares. Table 4 reports corpus scale. Because TerraDS indexes on the presence of Terraform files, it includes the IaC-minority projects under-represented in prior corpora [28, 29].

Table 4
Scale of the TerraDS corpus analyzed in this study

Entity	Count
Repositories	62,406
Modules	279,344
Resources	1,773,991
Modules with ≥ 1 module call	97,893 (35.0%)
Modules declaring a provider	260,259 (93.2%)

Table 5 characterizes the corpus along provider, popularity, size, and recency. The corpus is genuinely multi-cloud: AWS is the most common provider (29.1% of module-provider occurrences) but Azure (10.5%), Google Cloud (8.0%), Kubernetes (3.7%), and Helm (2.1%) are all well represented. Crucially, the corpus is not skewed toward popular projects, median stars and forks are both zero, so it reflects everyday practice rather than a curated set of flagship repositories, the population most vulnerable to the HCL-majority sampling bias noted in prior work. Repositories are recent and active: creation years concentrate in 2021 to 2024 and 89% record their latest commit in 2022 or later, with only 3.7% archived.

Table 5
Corpus characteristics (distributions).

Dimension	Distribution
Top providers (by module)	AWS 29.1%, Azure 10.5%, GCP 8.0%, Kubernetes 3.7%, Helm 2.1%
Repository size (KB)	median 27, Q3 120, p90 967 (mean 1,536; long-tailed)
Stars	median 0, p90 3, max 4,429
Forks	median 0, p90 3, max 4,396
Creation year	concentrated 2021 to 2024 (peak 2023: 15,069)
Latest commit	89% in 2022 or later; 3.7% archived
Modules per repository	median 2, Q3 3, p90 8, max 1,189

3.2 Dependency graph construction

For each repository r a directed graph is built $G(r) = (M, E)$: nodes M are its modules (identified by directory path); an edge $(m_i, m_j) \in E$ exists when m_i declares a module call whose source resolves to m_j. For a call with local source s from a module at directory d_i, the target is the normalized join of d_i and s; if it matches a module directory in the same repository the edge is resolved, otherwise retained as unresolved cross-file (its target lies outside the indexed set). Registry, VCS, HTTP, and bucket sources are labeled external and never marked cross-file. Algorithm 1 states the procedure in full for reproducibility. Across the corpus, 95.5% of local cross-file edges (154,494 of 161,749) resolve to an in-

dataset module; the 4.5% unresolved are reported explicitly (see Methods), and a 200-edge audit sample accompanies the replication package for manual precision checking.

Algorithm 1

Cross-file dependency resolution

Input : repository r with modules M (each: path, module_calls)

Output: edge set E, each with cross_file flag and resolution status

index ← { normalize(m.path) : m for m in M }

for each module m in M:

for each call c in m.module_calls:

kind ← classify(c.source)

if kind in {local_subdir, local_traversal}:

target ← normalize(join(m.path, c.source))

if target in index:

add edge (m → index[target]); cross_file=true; resolved=true

else:

add edge (m → external); cross_file=true; resolved=false

else: // registry, vcs, http, bucket

add edge (m → external); cross_file=false; resolved = n/a

return E

The classifier at line 4 inspects the source prefix; normalize collapses “.” and “..” path segments and strips a leading “./”. The procedure is linear in the number of module calls and fully deterministic.

3.3 Dependency pattern taxonomy (RQ2)

Every module dependency is classified by its source string into one of seven categories (Table 6). Two, directory-traversing and subdirectory local references, constitute the intra-repository cross-file class. The classification is fully deterministic: each category is defined by a syntactic property of the source string (its prefix or form), evaluated by a fixed rule with no human judgment. Consequently the taxonomy Because the classification is deterministic and involves no human coding, inter-rater agreement is not

applicable; accordingly, statistics such as Cohen's kappa [46, 47] are not reported., so no inter-rater agreement assessment applies, and none is reported, following the conventions summarized by Cohen [46], interpreted by conventional agreement bands [47], would provide. The classifier is released in full for inspection.

Table 6
Source-based taxonomy of module dependencies.

Category	Example source	Cross-file?
local_traversal	../../shared/db	Yes
local_subdir	./modules/network	Yes
registry	hashicorp/consul/aws	No (external)
vcs_remote	git::https://…/mod.git?ref=v1	No (external)
http_remote	https://…/mod.zip	No (external)
cloud_bucket	s3::https://…	No (external)
other	(unrecognized form)	—

3.4 Security linkage (RQ3)

A resource type is designated as *security-sensitive* if it governs one of four control domains whose misconfiguration is a recognized cloud-breach vector: identity and access management, network exposure, data storage, and encryption/key management. The specific types (Table 7) are drawn from resources covered by established IaC security-smell catalogs [1, 2] and by cloud configuration benchmarks such as the CIS Foundations Benchmarks, which define control objectives for exactly these four domains; the replication package ships the per-resource mapping to the corresponding benchmark control so the selection can be audited. Two properties of the set should be stated openly. It is provider-asymmetric, nine AWS types against one Azure type, which reflects both the corpus composition (AWS accounts for 29.1% of provider occurrences against Azure's 10.5%, Table 5) and the greater maturity of AWS-focused smell catalogs, not a judgment that Azure resources matter less. And it is deliberately conservative: restricting the set to well-established control domains means the RQ3 counts are lower bounds, since security-relevant resources outside these domains are not counted. Per repository, each cross-file dependency target is tested for the declaration of such a resource. The construct measured is precisely *reachability of modules declaring security-sensitive resources*: it quantifies the composed surface, and deliberately does not assert that any reached resource is misconfigured. Whether tools can evaluate that surface is exactly RQ4.

Table 7
Security-sensitive resource types, by control domain.

Control domain	Resource types
Identity & access	aws_iam_policy, aws_iam_policy_document, aws_iam_role, aws_iam_role_policy_attachment
Network exposure	aws_security_group, aws_security_group_rule
Data storage	aws_s3_bucket, aws_db_instance, azurerm_storage_account
Encryption / keys	aws_kms_key

Of the 161,749 local cross-file edges, 7,255 (4.5%) do not resolve to an in-dataset module. Categorizing them shows the residual is a dataset-coverage effect, not a resolver defect: 6,153 (84.8% of unresolved) point to a directory that is a valid relative target but has no corresponding module row in TerraDS (for example a helper directory the dataset did not index as a module), and 1,102 (15.2%) are parent traversals whose normalized target climbs above the indexed module set for that repository. No unresolved edge stems from a malformed source. These are retained as cross-file for prevalence (they do cross the calling file's boundary) while excluding them from resolved-target security linkage.

The resolver is validated in two ways. First, on a 200-edge sample of resolved edges, every resolved target is structurally consistent (a normalized relative path with no residual traversal, distinct from the calling directory). This is a structural consistency check, not a manual semantic validation, and it is deliberately not reported as a precision figure: confirming that each resolved target is the module the author intended would require manual inspection and labeling, which has not been performed. The sample is released so that such a validation can be carried out and the estimate extended. Second, on synthetic repositories with planted ground truth, known-resolvable module calls mixed with decoys pointing to non-existent directories, the resolver recovers all planted resolvable edges and resolves none of the decoys across 300 trials. This second check confirms implementation correctness (the path logic neither misses valid targets nor invents matches); it is a soundness check on the algorithm, not a claim about resolution accuracy on unseen real-world path idioms, which the first check addresses at sample scale.

3.5 Controlled tool evaluation (RQ4)

Because tool capability is version- and configuration-dependent, the exact environment is recorded by the experiment itself: each run writes a manifest (tool versions, resolved binary paths, platform, Python version, invocation, timestamps) alongside structured results and the captured stdout/stderr and exit code of every tool execution. A container definition in the replication package pins the exact versions that produced these results (Checkov 3.3.13, tfsec v1.28.14, Terrascan v1.19.9, Trivy 0.74.0), and the emitted manifest records them for every run. Table 8 summarizes the configuration. All scanners run inside a single Linux container image on the generated directory tree; each is invoked on the *root*

directory (not per module) so that module resolution is available to any tool that implements it, and each runs in its documented default mode without custom rule sets, suppression files, or tuning flags. No terraform init is executed and no provider schemas are downloaded, so all tools operate purely on static configuration; this is the mode in which such scanners are typically deployed in pre-apply CI checks. All capability statements about scanner behavior should therefore be read as holding *under the evaluated versions and default documented configurations*; a different release or a tuned configuration could yield different verdicts, and the released generator makes re-running the battery against any such configuration straightforward.

Table 8
Evaluation environment for the RQ4 battery.

Item	Setting
Scanners	Checkov 3.3.13, tfsec v1.28.14, Terrascan v1.19.9, Trivy 0.74.0; versions are pinned in the released Dockerfile and confirmed in the emitted run manifest
Invocation	Default directory scan on the repository root; JSON output; no custom rules or suppressions
Module handling	Tools free to resolve local modules themselves; no external module download
Terraform	No terraform init; no provider schemas fetched; static analysis only
Platform	Single Linux container image; identical inputs across all tools
Oracle	Per-tool signal check discovered from that tool's own inline positive control

The capability experiment uses a matched control/treatment/inline design. For each of seven cross-file constructs three repositories are generated that are identical except for one value: the *control* delivers a secure value (private) through the construct; the *treatment* delivers an insecure value (public-read) through the same construct; the *inline positive control* writes the insecure value directly in the resource file with no indirection. All three variants are scanned, and the verdict is decided by a four-part oracle rather than by raw output comparison: (i) the signal check identifier must match the one the tool itself fires on its inline positive control; (ii) the finding must attach to one of the resource addresses the property is expected to flag, compared after stripping module-path prefixes, so that a finding on an unrelated resource cannot be mistaken for a hit; (iii) the treatment run must differ from the matched control run on that signal, isolating the effect of the injected value; and (iv) the inline control must fire, confirming the rule exists and is reachable. A construct is judged only when all four conditions are evaluable. The verdict vocabulary is kept deliberately separate and never collapsed: RESOLVED, NOT RESOLVED, INCONCLUSIVE (the tool has no applicable rule, so its inline control does not fire), ERROR (the tool failed to execute or returned unparseable output), and N/A (the tool is absent). Distinguishing ERROR and INCONCLUSIVE from NOT RESOLVED matters: an execution failure silently recorded as a missing finding would manufacture a blind spot that does not exist. Generated configurations undergo syntax validation with terraform fmt -check before scanning, so a malformed test case is reported as

ERROR rather than scored as a clean run with no findings; semantic validation requiring provider initialization is outside the static-analysis execution mode the scanners run in, and is provided as a separate opt-in pass in the replication package. This guards against the two failure modes that a naive set comparison admits: incidental differences in unrelated checks, and absent rules being read as blind spots. The decision rule: if treatment reproduces inline and differs from control, the construct is *resolved*; if treatment equals control while inline differs, the construct is a genuine *blind spot*; if inline equals control, the probe is inconclusive (no rule fires at all). The inline arm is essential, it separates "no rule exists" from "the rule exists but the cross-file value is not followed". An earlier version of this experiment filtered findings by an ‘acl’ substring, missed that Checkov’s public-read check is named CKV_AWS_20, and wrongly suggested a blind spot; the control/treatment design reported here supersedes it, and the correction is documented as a methodological caution for similar studies.

4 Results

4.1 RQ1: prevalence

Of 62,406 repositories, 30,377 (48.7%) declare at least one module dependency and 21,665 (34.7%) contain at least one intra-repository cross-file dependency. The median repository declares two modules; the mean is 4.48 and the maximum 1,189. Among repositories with at least one cross-file edge (n = 21,665 with metadata available), the edge count has mean 7.47, median 2, and maximum 2,039, an extremely skewed distribution (skewness 25.0, excess kurtosis 1,105). Figure 2 shows the distribution and its complementary CDF.

Table 9
Prevalence summary (RQ1).

Measure	Value
Repositories analyzed	62,406
… with ≥ 1 module dependency	30,377 (48.7%)
… with ≥ 1 cross-file dependency	21,665 (34.7%)
Modules per repo (median / mean / max)	2 / 4.48 / 1,189
Cross-file edges per repo, among > 0 (median / mean / max)	2 / 7.47 / 2,039
Cross-file edges resolved to an intra-repo module	154,494 (95.5%)

The skew has a practical reading. Because half of all repositories declare only one or two modules, cross-file coupling is absent or trivial for the median project; a naïve summary by the mean would therefore mislead. The action is in the tail. The 10% of cross-file repositories above the 90th percentile carry a disproportionate share of all cross-file edges, and it is in these large, densely-coupled projects that a single mis-set variable propagates furthest and is hardest to trace by eye. This concentration matters for both tooling and study design: tools that ignore cross-file structure lose the most precisely

where configuration is most entangled, and corpora that exclude such projects (for example by requiring HCL to dominate the repository) systematically remove the very population where the phenomenon is strongest. The use of TerraDS, which does not impose that requirement, is what makes the tail observable.

4.2 RQ2: dependency patterns

Figure 3 and Table 10 report the taxonomy over all 264,352 classified dependencies. Directory-traversing local reference is the single most frequent pattern (101,819; 38.5%). Together with subdirectory references (59,930; 22.7%), intra-repository cross-file dependencies account for 161,749 edges, 61.2% of all module dependencies, more than every external category combined (Fig. 3b). The dominant idiom by which practitioners compose Terraform is, structurally, a cross-file idiom.

Table 10
Dependency pattern distribution (RQ2).

Pattern	Count	Share	Cross-file
local_traversal	101,819	38.5%	Yes
registry	64,001	24.2%	No
local_subdir	59,930	22.7%	Yes
vcs_remote	37,171	14.1%	No
other	920	0.4%	—
cloud_bucket	435	0.2%	No
http_remote	76	0.0%	No

Table 11 grounds the two cross-file categories in practice. Traversals typically reference shared modules kept up the directory tree; subdirectory references instantiate project-local modules under a modules/ folder. Both are deliberate, maintainability-oriented idioms, and both place referenced configuration in a different file from the reference.

Table 11
Cross-file patterns as they appear in practice.

Pattern	Representative source	Typical intent
local_traversal	../../modules/vpc	Shared module up the tree
local_traversal	../network	Sibling-directory reuse
local_subdir	./modules/app_instance	Project-local module
local_subdir	./servercert	Local helper module

Why does traversal dominate? The prevalence of ../ references reflects a common repository layout in which environments (for example prod, staging) live in sibling directories and share modules kept one or two levels up. Each environment reaches up and across to the shared module, generating traversal edges in proportion to the number of environments times the number of shared modules, an interaction that grows quickly and helps explain the heavy tail reported above. Subdirectory references, by contrast, reflect self-contained projects that vendorize their modules under a local modules/ tree. Both layouts are explicitly recommended in community guidance, so the cross-file structure measured here is not a smell to be eliminated but the intended shape of maintainable Terraform. That is precisely why the capability question of RQ4 matters: if the dominant, recommended idiom were opaque to scanners, security analysis would be systematically blind on well-structured code. The controlled evaluation shows it is largely not, a relevant finding given how common these layouts are.

4.3 RQ3: the targeted security surface

In 9,035 repositories, or 41.7% of those with at least one cross-file dependency, such a dependency targets a module that declares at least one security-sensitive resource. The measure is stated precisely: this establishes that the dependency's target module contains security-relevant configuration, not that the dependency itself carries a security-relevant value into that resource. Establishing the latter would require attribute-level data-flow analysis over raw HCL, which is outside the scope here and is identified as future work. Figure 4 breaks the targeted instances down by type: IAM policy documents (13,851) and security-group rules (12,897) lead, followed by role-policy attachments, roles, and security groups. Aggregated by control domain (Table 13), identity and access management accounts for 45,930 instances (58.5%) and network access control for 24,173 (30.8%). The resource categories governing who may act and what may connect are therefore the ones most often located in modules that other modules depend on. Whether a given dependency actually influences those resources' settings is a separate, finer-grained question not answered here.

Table 12
Security-sensitive resource types most often declared in targeted modules (RQ3).

Resource type	Instances in targeted modules
aws_iam_policy_document	13,851
aws_security_group_rule	12,897
aws_iam_role_policy_attachment	12,485
aws_iam_role	11,648
aws_security_group	11,276
aws_iam_policy	7,946
aws_s3_bucket	4,160
azurerm_storage_account	1,674
aws_kms_key	1,373
aws_db_instance	1,143

Table 13
Targeted security-sensitive resources by control domain. The rollup by domain.

Control domain	Instances in targeted modules	Share
Identity & access (IAM)	45,930	58.5%
Network access control	24,173	30.8%
Data storage	6,977	8.9%
Encryption / key management	1,373	1.8%

4.4 Statistical analysis

What correlates with cross-file coupling (RQ1)

Over all 62,406 repositories, the cross-file edge count correlates strongly with module count (Spearman $\rho = 0.752$; Pearson on logs $r = 0.773$) and moderately with repository size ($\rho = 0.336$), but only weakly with stars ($\rho = 0.113$) and forks ($\rho = 0.109$). All coefficients are significant, which is unremarkable at this sample size; the magnitudes are what separate the factors. Cross-file coupling scales with structural complexity, not popularity, consistent with these dependencies being a by-product of modular decomposition rather than community attention (Table 14).

Table 14
Correlation of cross-file edge count with repository characteristics (n = 62,406; 95% CI via Fisher z).

Factor	Spearman ρ	95% CI	Pearson r (log)
Number of modules	0.752	[0.748, 0.755]	0.773
Repository size (KB)	0.336	[0.329, 0.343]	0.335
Stars	0.113	[0.105, 0.120]	—
Forks	0.109	[0.101, 0.117]	—

Is the distribution heavy-tailed? (RQ1)

To substantiate the heavy-tail characterization, two candidate models are fitted to the positive cross-file counts (n = 21,665) and compared by two criteria that, informatively, disagree. A discrete power law (zeta distribution, maximum-likelihood exponent $\alpha = 1.643$ at $x_{min} = 1$) and a log-normal ($\mu = 1.08$, $\sigma = 1.09$) are compared first by Kolmogorov–Smirnov distance, which favors the log-normal decisively ($D = 0.181$ vs 0.459), and second by a Vuong likelihood-ratio test, which favors the power law ($V = 8.31$, $p < 10^{-15}$). The disagreement is expected and worth stating plainly: the likelihood criterion is dominated by the dense low-count bulk, which a power law captures well point-by-point, whereas the KS statistic reflects the overall cumulative shape, which the log-normal captures better. Both tests, however, decisively reject a thin-tailed (e.g. geometric or Poisson) description, and the raw moments, skewness 25.0, excess kurtosis 1,105, place the distribution firmly in the highly right-skewed, long-tailed regime. Only what both criteria support is therefore claimed: the distribution is highly right-skewed and long-tailed, with neither a pure power law nor a log-normal providing a fully adequate fit, and no single generative model is asserted.

What predicts targeting a security-sensitive module (RQ3)

Over the 21,665 repositories with a cross-file dependency (9,035 positives), a logistic regression predicts whether any such dependency targets a security-sensitive module, from standardized log-transformed structural features. Table 15 reports the model. Two findings deserve emphasis, and the second is a negative result stated plainly. First, the strongest predictor is the cross-file edge count itself (odds ratio 1.41 per standard deviation): a higher cross-file edge count is more likely to target a module declaring security-relevant configuration. Second, module count contributes no positive weight once cross-file edge count is controlled (OR = 0.95), popularity is inert (stars OR = 1.05), and overall discrimination is weak (AUC = 0.615). Complementary tests agree: repositories that target such modules have significantly higher cross-file counts (Mann–Whitney U, $p \approx 10^{-150}$, rank-biserial 0.203, a small-to-moderate effect), and above-median module count associates with the outcome only weakly ($\chi^2 = 341.5$, $\varphi = 0.126$). Structural metrics are therefore not claimed to predict this outcome well; they do not. What the analysis establishes is narrower: cross-file edge count, not size or popularity, is the structural property most associated with targeting such modules, while the weak discrimination shows that

whether a given project's coupling targets such modules is largely determined by factors these features do not capture.

Table 15
Logistic regression for targeting a security-sensitive module (n = 21,665; 9,035 positives; 95% CI via 400-sample bootstrap).

Predictor (per + 1 SD, log-scaled)	OR	95% CI	Reading
Cross-file edge count	1.41	[1.35, 1.48]	Strongest positive
Repository size (KB)	1.14	[1.11, 1.18]	Weak positive
Stars	1.05	[1.02, 1.08]	Effectively none
Number of modules	0.95	[0.91, 1.00]	None once density controlled
Model discrimination (AUC)	0.615	[0.607, 0.622]	Weak overall

Controlled evaluation: what do scanners actually resolve? (RQ4)

The preceding results establish that local module dependencies are prevalent and that their targets frequently declare security-relevant resources. The capability question is whether scanning tools can follow configuration values along the composition those dependencies create. Using the control/treatment/inline design described in Methods, seven constructs are probed across *four* widely used open-source scanners, Checkov, tfsec, Terrascan, and Trivy, and across *two independent security properties*: a public-read S3 ACL (P1) and an open security-group ingress rule (P2). All four scanners expose a signal check for both properties, so both support a full cross-tool comparison and each is an independent replication of the other. A third property, unencrypted storage, was attempted but is excluded from the results: expressing "no encryption" through a cross-file count leaves the encryption block present in the static source, so the variant is not a clean unencrypted-bucket test; it is reported as an open item for future work rather than as evidence. Each tool’s signal check is discovered from its own inline positive control rather than hard-coded, so the comparison is fair across differing rule identifiers.

Two levels of composition. The seven constructs deliberately span two distinct levels, and the distinction matters for how the results should be read. Four constructs test *intra-module multi-file resolution*: the value reaches the resource from a different file within the same module directory (a variable default, a local value, a terraform.tfvars entry, an override.tf merge). Three constructs test *inter-module value propagation*: the value crosses a module boundary through a module input, an output chained into a second module, or two levels of nesting. Only the second group corresponds directly to the module dependencies measured in RQ1 to RQ3; the first group characterizes the file-level composition that occurs *within* each such module. Both are necessary for a complete picture, a tool that follows module boundaries but not in-module file composition, or the reverse, would be misdescribed by either level alone, but the prevalence results speak to the inter-module level specifically. Table 16 reports the security-group property (P2), grouped by level.

Table 16
Resolution across four scanners under property P2 (open security-group ingress).

Construct (level / delivery mechanism)	Checkov	tfsec	Terrascan	Trivy
INTRA-MODULE MULTI-FILE RESOLUTION				
C1 Variable default (separate file)	RESOLVED	RESOLVED	RESOLVED	RESOLVED
C2 Local value (separate file)	RESOLVED	RESOLVED	RESOLVED	RESOLVED
C3 terraform.tfvars	RESOLVED	NOT	NOT	NOT
C7 override.tf (last-wins merge)	RESOLVED	RESOLVED	NOT	RESOLVED
INTER-MODULE VALUE PROPAGATION				
C4 Module input (root → module)	RESOLVED	NOT	RESOLVED	NOT
C5 Module output chaining	RESOLVED	NOT	NOT	NOT
C6 Nested modules (two levels)	RESOLVED	NOT	NOT	NOT

Table 16: Capability is graded rather than binary. All four scanners resolve variable defaults and locals held in a separate file; only Checkov resolves terraform.tfvars; inter-module propagation is fully resolved only by Checkov, partially by Terrascan (direct module input), and not at all by tfsec or Trivy. "NOT" abbreviates NOT RESOLVED under the evaluated versions and default configurations.

The four scanners are not arbitrary. They are widely used open-source Terraform security scanners: each is actively maintained, distributed as a CI-ready container or binary, and appears routinely in practitioner guidance and IaC pipeline templates. No quantitative market-share claim is made, but this set covers the open-source scanners a Terraform team is most likely to encounter. They also span the relevant architectural spectrum: Checkov builds an internal graph over resources and variables, whereas tfsec, Terrascan, and Trivy are predominantly rule-based per-resource evaluators. Covering both designs means the observed pattern is consistent with Checkov’s graph-based evaluation model; this study does not, however, isolate architecture as the sole causal factor, since the tools also differ in rule engines, implementations, and versions. Proprietary scanners are excluded (no reproducible access) and KICS (which the released generator supports but which was not included in this run); the generator auto-discovers each tool's signal, so adding scanners is mechanical.

Table 17 repeats the battery under the ACL property (P1). Because all four scanners also expose a signal here, P1 is a genuine independent replication on a different resource type and a different rule, not merely a supplementary check. Its verdicts reproduce Table 16 exactly except for override files, which no tool resolves under P1 but three of four resolve under P2, showing that even a single construct’s handling can depend on the property, so tool *behavior* must be characterized per construct and per property rather than summarized in a single verdict.

Table 17
Resolution under the S3 public-read ACL property (P1)

Construct	Checkov	tfsec	Terrascan	Trivy
C1 Variable default (intra)	RESOLVED	RESOLVED	RESOLVED	RESOLVED
C2 Local value (intra)	RESOLVED	RESOLVED	RESOLVED	RESOLVED
C3 terraform.tfvars (intra)	RESOLVED	NOT	NOT	NOT
C7 override.tf (intra)	NOT	NOT	NOT	NOT
C4 Module input (inter)	RESOLVED	NOT	RESOLVED	NOT
C5 Module output chaining (inter)	RESOLVED	NOT	NOT	NOT
C6 Nested modules (inter)	RESOLVED	NOT	NOT	NOT

The result is a capability *gradient*, not a binary split. Three tiers emerge, and they replicate across both properties. First, a tier of *universal support*: every scanner resolves a value delivered through a variable default or a local value defined in a separate file (C1, C2). The common assertion that IaC scanners cannot follow values across files is therefore false for the two most frequent intra-module mechanisms. Second, a tier where support *diverges*: values supplied through terraform.tfvars (C3) are resolved only by Checkov, consistently in both properties, even though the same tools resolve the variable default that tfvars overrides. Third, a tier of *inter-module propagation* where only Checkov resolves all three constructs; Terrascan resolves the simplest case, a value passed directly as a module input (C4), but not values chained through a module output or two levels of nesting; tfsec and Trivy resolve none. Override files (C7) sit outside this ordering. No tool resolves them under P1, while three of four do under P2.

Across the seven constructs tested, coverage narrows as the number of indirections between a value's definition and its use grows. This is a pattern within the battery used here, not an established law. All four scanners follow a value one hop across a file boundary; only Checkov follows it through a file-precedence mechanism such as tfvars; and only Checkov follows it across successive module boundaries, with Terrascan managing a single hop. This ordering tells a practitioner more than a per-tool pass or fail verdict would. It says which compositions their chosen scanner can actually see.

Two secondary observations refine the picture. First, override.tf behaves inconsistently: three of four tools resolve the last-wins merge under P2, yet none does under P1. Override handling is therefore neither a uniformly resolved construct nor a single confirmed gap; it is inconsistent and property-dependent, and should not be relied upon in either direction. Second, the constructs of full cross-tool agreement (C1 and C2) are the shortest paths from definition to use, a single hop within one module, so that agreement marks the floor of capability rather than its ceiling.

The corrected conclusion is more actionable than a blanket claim in either direction. Cross-file structure is pervasive, accounting for 61.2% of dependencies, and it frequently reaches modules declaring security-sensitive resources in 9,035 repositories. Whether a scanner can evaluate that structure

depends heavily on which scanner. A team using Checkov inherits resolution across every construct tested, bar override files under one property. A team using tfsec, Terrascan, or Trivy is covered for the simplest and most common mechanisms, a variable default or a local in a separate file, but not for tfvars-supplied values, and, with the partial exception of Terrascan on direct module inputs, not for values crossing module boundaries. This is stated as a result about the tested constructs, not as a general claim about every cross-file security value. This is a concrete, tool-specific hazard, not a diffuse limitation, and it motivates either multi-tool scanning or a plan-time pass that evaluates fully-resolved configuration. On scope: two properties carry the main result with all four tools signaling, a third was explored but excluded for the reason given above, and the released generator auto-discovers each tool's signal so the battery extends mechanically to more tools, checks, and constructs.

A worked trace

To make the mechanism concrete, consider construct C5 (module output chaining), the most indirect construct. The root module passes the insecure value into module cfg as an input; cfg re-exports it as an output; the root wires that output into module storage, which applies it to the resource. The value therefore crosses three module boundaries and four files. In the treatment run *Checkov* raises the signal check on module storage identical, after stripping the module-path prefix, to the inline positive control, while the secure control raises nothing: Checkov has constructed the inter-module value flow and substituted the effective value. Run on the *same three repositories*, tfsec, Terrascan, and Trivy raise nothing in either the treatment or the control: their verdicts are independent of the chained value, because they never propagate it across the module boundary. The single experiment thus separates the tools cleanly, one reconstructs the flow, and three do not, on a construct that is entirely ordinary Terraform. For C1 and C2, where the value is one hop away in a sibling file, all four fire identically; those are the constructs on which their capabilities coincide.

5 Discussion

5.1 Implications for research

The results invite a more careful premise. Papers motivating multi-file or graph-based IaC analysis frequently begin from the assertion that existing scanners cannot follow cross-file references; the evidence here suggests this assertion should be qualified rather than stated flatly, since no tool tested is wholly unable to follow values across files: all four resolve the simplest intra-module mechanisms, and they differ in how far beyond that they reach. The defensible motivations that remain are precise: override semantics, statically-unknowable values, cross-repository composition, and, the substrate this paper quantifies, analyses that require the dependency graph itself as an object of study. The characterization (RQ1 to RQ3) shows that graph is large, skewed, and security-relevant, which is motivation enough without relying on an untested capability assumption.

5.2 Implications for practice

For practitioners, the picture is partly reassuring and partly a caution, and which applies depends on the scanner. Splitting configuration across files within a module does not, by itself, hide a security-relevant value: every scanner tested followed a variable default and a local value defined in a separate file. Beyond that, coverage narrows. Values supplied through terraform.tfvars were resolved only by Checkov, and values crossing module boundaries were resolved fully only by Checkov, with Terrascan handling the direct module-input case. A team that factors security-relevant configuration into shared modules and relies on tfsec, Terrascan, or Trivy alone should not assume the resulting composition is analyzed end to end. Override-file handling was property-dependent in this evaluation: no scanner resolved the ACL override, whereas Checkov, tfsec, and Trivy resolved the security-group override. Teams should therefore not assume either reliable resolution or systematic pre-override analysis; a plan-time check provides a safer complementary control. For tool builders, both the tfvars gap and inconsistent override merging are concrete, implementable targets.

5.3 Relation to prior findings

The results here *agree* with Iosif et al. on which resources carry the most security weight, they found modules and security groups among the most vulnerability-dense resources in AWS Terraform [28], and RQ3 here shows security groups and IAM constructs are precisely those most often reached across module boundaries. The agreement is not incidental: the resources that are heavily modularized are the ones whose configuration is dispersed, and dispersion is what the cross-file measure captures.

The results here *differ* from the prevailing capability assumption, and the reason is methodological. Where GLITCH's authors list multi-file analysis as future work [4] and much of the smell-detection literature evaluates per script [1, 2, 5], resolution is tested directly with ground-truth controls and find that at least one production tool (Checkov) already performs it across module boundaries, while others do not. The blanket assumption that scanners cannot resolve cross-file references is therefore false as stated; the accurate claim is narrower and tool-specific, which is exactly what a capability map provides.

The results here *extend* the fault- and quality-oriented literature. Drosos et al. characterize IaC faults and note that misconfigurations are a substantial fault class whose fixes are often small [21]; RQ4 here adds that, for cross-file misconfigurations of the mainstream kind, existing scanners can already surface the responsible value, so the bottleneck is less detection than the specific unresolved constructs (override files, statically-unknowable inputs). Verdet et al. measured per-file adoption of security policies [29]; because the analyzable surface extends across files for most constructs, per-file adoption measurements likely understate what tools can verify once cross-file resolution is credited. Relative to dependency-and-quality tooling such as TerraMetrics and module-update studies [24, 25, 26], which quantify structure without a security-capability test, the contribution here is the controlled link between the two.

5.4 Practical implications for DevSecOps and CI/CD

For organizations running IaC scanning in delivery pipelines, the capability map translates into concrete pipeline decisions. First, *tool choice is a security control, not a preference*. A pipeline gated on tfsec or

Trivy alone should not assume that values supplied through terraform.tfvars, or propagated across module boundaries, are analyzed end to end. Terrascan resolved the direct module-input construct but not module-output chaining or nested modules, while Checkov resolved every tested construct except the ACL override. Teams that have standardized on a single scanner should either add a scanner demonstrated to resolve the constructs they depend on, or introduce a plan-time complementary control; the choice should be explicit rather than implicit. Override-file handling is property-dependent and should not be assumed reliable in either direction; a plan-time check provides a safer complementary control.

Second, *layer static scanning with plan-time evaluation*. Scanning the rendered plan evaluates fully-resolved configuration and therefore sidesteps both cross-file resolution limits and override-file semantics. A practical arrangement is a fast per-commit static scan for immediate feedback, plus a plan-time scan at the merge or pre-apply gate where the full value flow is available. Third, *treat modularization depth as a review signal*: the regression shows coupling density, not repository size, is what tracks reaching security-relevant configuration, so changes that add cross-module wiring around IAM or security-group resources warrant closer review than their diff size suggests. Finally, teams can regenerate this battery against their own toolchain, the released generator discovers each tool's signal automatically, turning "does this scanner see it?" from an assumption into a test that runs in minutes.

Three factors support cautious generalization. First, the corpus is multi-cloud and not popularity-biased (Table 5), so the prevalence and pattern results are unlikely to be artifacts of a narrow provider or of flagship projects. Second, the RQ4 verdicts rest on Terraform's own documented semantics, variable defaults, tfvars precedence, module I/O, and override merging behave identically regardless of which resource type carries the value, so the resolution results should transfer beyond the S3-ACL probe to other resources governed by the same constructs. Third, cross-tool agreement on the override gap indicates the boundary is a property of the constructs, not of one tool's implementation. The principal threats to generalization are the single signal check per tool and property, and the four-tool scope; both are addressed by the released generator, which reproduces the battery for any tool and any check the user supplies.

5.5 Actionable recommendations

Do not motivate multi-file IaC analysis by asserting that scanners cannot resolve cross-file values; verify capability against the current tool, and cite the specific unresolved constructs. Index corpora on the presence of IaC rather than language dominance to include the complex, cross-file-heavy projects the heavy-tailed distribution highlights. The module dependency graph is a productive unit of analysis; attribute-level flows over raw HCL are the natural refinement.

Continue modularizing; scanning follows the mainstream constructs. Treat override files as a scanning hazard: add a plan-time scan wherever they are used, since the pre-merge configuration a scanner sees can differ from the configuration Terraform applies.

Improve and test override-file merge semantics, whose resolution was property-dependent in this evaluation, and document construct-level capabilities explicitly so that researchers and users need not rediscover capability boundaries experimentally.

A concrete pipeline recommendation follows from the cross-tool result. Because tfsec, Terrascan, and Trivy did not resolve the tested cross-boundary values supplied through terraform.tfvars variables or module boundaries, while Checkov did, teams whose security gate depends on one of the former may leave important classes of cross-file value propagation unanalyzed. A low-cost mitigation is to include Checkov or to complement static scanning with a terraform plan-time scan that evaluates the fully-resolved configuration Terraform will actually apply, closing both the override gap and any tool-specific input-resolution gap in one step. This is intended as a pragmatic layered approach: static scanning catches the mainstream cross-file cases cheaply and early, and a plan-time pass covers the residual constructs this experiment isolates.

6 Threats to Validity

Construct validity. Cross-file coupling is operationalized through module-call dependencies in TerraDS metadata; finer couplings (individual variable or attribute references) are not counted, so RQ1 prevalence is a conservative lower bound. RQ3 measures reachability of security-sensitive modules, not misconfiguration, and the paper's claims are scoped accordingly. RQ4 evaluates one tool-specific signal check for each of the two security properties tested; other checks may exhibit different resolution behavior; constructs could in principle behave differently for other checks, which is why the generator is released for extension.

Path resolution may err on unusual constructions; 7,255 cross-file edges (4.5%) did not resolve to an in-dataset module and are reported as such, with a 200-edge audit sample released for manual precision assessment. Both the pattern taxonomy and the resolver verdict are objective. Each has a single correct answer, determined by the source string and the repository's directory structure. Validating them is therefore a precision check against ground truth, not a subjective coding task, so no inter-rater agreement coefficient applies and none is reported. The RQ4 experiment was corrected once during development. An initial substring-based analysis produced a spurious blind-spot signal, and the published control/treatment design eliminates that class of error. The correction is disclosed deliberately.

TerraDS is built by indexing GitHub for Terraform files, so it inherits any bias in what GitHub exposes and in the indexing criteria; repositories behind authentication, on other forges, or without recognizable Terraform files are absent. The corpus characterization (Table 5) mitigates concern, it is multi-cloud, recent, and not popularity-skewed (median stars zero), but generalization beyond public GitHub Terraform should remain cautious. The 4.5% of unresolved edges further reflect incomplete module indexing within the dataset itself (see Methods), a coverage limit rather than a sampling bias, but one readers should weigh when interpreting absolute counts.

The corpus is open-source Terraform from TerraDS; proprietary codebases, other IaC languages, and styles absent from GitHub may differ. Several ecosystem factors bound generalization further. TerraDS captures public GitHub repositories, so private and enterprise Terraform, often following different modularization and security conventions, are unobserved, and repositories removed or made private after indexing are absent. The corpus predates OpenTofu's divergence from Terraform; whether OpenTofu's parsing or module semantics shift cross-file prevalence or tool resolution is untested and a natural extension. Terraform syntax has evolved across major versions, and newer constructs may change the pattern distribution; the taxonomy is version-agnostic at the module-source level but does not model version-specific features. The tool evaluation covers four scanners (Checkov, tfsec, Terrascan, Trivy); others (KICS, Semgrep, Regula) may resolve different construct sets, and extending the battery is mechanical with the released generator, which auto-discovers each tool's signal check. Two security properties carry the reported capability results across all four tools. A third property was explored but excluded because its static formulation did not provide a semantically clean test. TerraDS extracts metadata rather than raw HCL and inherits that extraction's limitations.

The RQ4 result is a capability map over a representative set of the most common cross-file constructs in Terraform, not an exhaustive audit of any scanner. Seven constructs across two composition levels cannot characterize a tool completely, and no such claim is made: a scanner shown to resolve these constructs may still miss others not tested, and one shown to fail them may resolve constructs outside this battery. The claim is deliberately bounded, these particular, widely-used constructs, delivered through these properties, resolved or not by these tools, and its value is comparative and diagnostic rather than a certification of any tool's overall correctness. The conclusions are held to that scope throughout.

Corpus statistics are computed deterministically over the full dataset, so no sampling error applies within it; where distributions are skewed medians, tails, and effect sizes are reported rather than means alone, and weak model performance (AUC = 0.615) is stated plainly rather than foregrounding statistical significance.

7 Conclusion and Future Work

This paper characterized cross-file dependencies across 62,406 Terraform repositories and tested, under controls, whether contemporary scanning resolves them. The structure is pervasive: present in 34.7% of repositories, highly right-skewed and long-tailed, dominated by directory-traversing references (38.5%), tracking structural complexity rather than popularity, and targeting modules that declare security-sensitive resources, chiefly IAM and network access, in 9,035 repositories. The capability result is a map rather than a verdict: capability is graded: every scanner resolves a variable default or a local value defined in a separate file, only Checkov resolves terraform.tfvars, and inter-module propagation is resolved fully only by Checkov and partially by Terrascan. The common assumption that scanners cannot follow cross-file references is thus inaccurate as stated for every tool tested; what varies is how

far each one follows, which is precisely why capability should be measured per construct rather than assumed.

The conclusions are bounded in three deliberate ways. The study measures *module-level* cross-file dependencies, meaning those expressed through module calls, not every form of inter-file coupling in Terraform. Individual variable or attribute references that also cross files form a finer surface that is not enumerated, so the prevalence figures are lower bounds. The capability findings hold for the *constructs and tools tested*: seven constructs spanning two composition levels, four scanners, and two reported security properties. The claim is only that, for these mainstream constructs, resolution is complete across the tested constructs only in Checkov, partial in Terrascan, and limited to the shortest intra-module flows in tfsec and Trivy. Within these bounds the evidence is direct and reproducible; beyond them it motivates, rather than settles, the questions below.

Future work follows the boundary located here. First, widening the capability battery, more scanners, more signal checks, more constructs (remote state, dynamic blocks, generated overrides), turns this paper's probe into a capability map of the ecosystem. Second, attribute-level flow analysis over raw HCL would refine the module-level surface measured here. Third, the property-dependent handling of override files invites focused work on implementing and evaluating consistent override-aware scanning. Beyond these specifics, the broader lesson is a design and evaluation guideline: Cross-file dependency resolution should be treated as a first-class capability of IaC analysis tools and evaluated explicitly rather than assumed present or absent; the evidence shows sufficient variation across tools that only direct measurement can establish the capability of a given scanner.

Declarations

Author contributions: Moatasem M. Draz designed the study, implemented the analysis pipeline and the controlled experiment, wrote and revised the manuscript, and approved the final version. The author read and approved the manuscript.

Funding: The author received no specific funding for this work.

Competing interests: The author declares no competing interests.

Data availability: The study analyzes TerraDS, publicly archived on Zenodo (https://zenodo.org/records/14217386). The replication package contains the dependency-graph builder, taxonomy classifier, security-linkage and statistical scripts, the per-repository dataset (62,406 rows), the 200-edge resolver audit sample, generators for all controlled-experiment variants, and scripts that produce every figure and table. All of these are available in the GitHub repository [https://github.com/moatasem2005/Cross-File-Dependencies-in-Terraform-Prevalence-Patterns-and-How-Much-Scanners-Actually-Resolve] and permanently archived at Zenodo (https://doi.org/10.5281/zenodo.22216732).

Code availability: All code used in this study is publicly available at the GitHub repository [https://github.com/moatasem2005/Cross-File-Dependencies-in-Terraform-Prevalence-Patterns-and-How-Much-Scanners-Actually-Resolve] and permanently archived at Zenodo (https://doi.org/10.5281/zenodo.22216732).

References

1. Rahman, A., Parnin, C. & Williams, L. The Seven Sins: Security Smells in Infrastructure as Code Scripts. ICSE 2019, pp. 164–175. (2019). https://doi.org/10.1109/ICSE.2019.00033
2. Rahman, A., Rahman, M. R., Parnin, C. & Williams, L. Security Smells in Ansible and Chef Scripts: A Replication Study. *ACM TOSEM*. **30** (1), 3:1–331. https://doi.org/10.1145/3408897 (2021).
3. Rahman, A., Farhana, E., Parnin, C. & Williams, L. Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts. ICSE 2020, pp. 752–764. (2020). https://doi.org/10.1145/3377811.3380409
4. Saavedra, N. & Ferreira, J. F. GLITCH: Automated Polyglot Security Smell Detection in Infrastructure as Code. ASE 2022, pp. 1–12. arXiv:2205.14371 (2022).
5. Opdebeeck, R., Zerouali, A. & De Roover, C. Control and Data Flow in Security Smell Detection for Infrastructure as Code: Is It Worth the Effort? MSR 2023, pp. 534–545. (2023). https://doi.org/10.1109/MSR59073.2023.00079
6. Brikman, Y. Terraform: Up and Running - Writing Infrastructure as Code, 3rd edition. O'Reilly Media (2022).
7. Rahman, A. & Williams, L. Different Kind of Smells: Security Smells in Infrastructure as Code Scripts. *IEEE Secur. Priv.* **19** (3), 33–41. https://doi.org/10.1109/MSEC.2021.3065190 (2021).
8. Rahman, A. & Williams, L. Characterizing Defective Configuration Scripts Used for Continuous Deployment. ICST 2018, pp. 34–45 (2018).
9. Rahman, A., Mahdavi-Hezaveh, R. & Williams, L. A Systematic Mapping Study of Infrastructure as Code Research. *Inf. Softw. Technol.* **108**, 65–77 (2019).
10. Sharma, T., Fragkoulis, M. & Spinellis, D. Does your configuration code smell? MSR 2016, pp. 189–200 (2016).
11. Schwarz, J., Steffens, A. & Lichter, H. Code Smells in Infrastructure as Code. QUATIC 2018, pp. 220–228 (2018).
12. Kumara, I. et al. The Do's and Don'ts of Infrastructure Code: A Systematic Gray Literature Review. *Inf. Softw. Technol.* **137**, 106593 (2021).
13. Opdebeeck, R., Zerouali, A. & De Roover, C. Smelly Variables in Ansible Infrastructure Code: Detection, Prevalence, and Lifetime. MSR 2022, pp. 61–72 (2022).
14. Rahman, A. & Parnin, C. Detecting and Characterizing Propagation of Security Weaknesses in Puppet-based Infrastructure Management. *IEEE TSE*. **49** (6), 3536–3553 (2023).
15. Rahman, A., Barsha, F. L. & Morrison, P. Shhh! 12 Practices for Secret Management in Infrastructure as Code. IEEE SecDev 2021 (2021).

16. Bhuiyan, F. A. & Rahman, A. Characterizing Co-located Insecure Coding Patterns in Infrastructure as Code Scripts. ASE 2020 Workshops, 27–32 (2020).
17. Bent, E., Hage, J., Visser, J. & Gousios, G. How Good is Your Puppet? An Empirically Defined and Validated Quality Model for Puppet. SANER 2018, 164–174 (2018).
18. Sotiropoulos, T., Mitropoulos, D. & Spinellis, D. *Practical Fault Detection in Puppet Programs. ICSE 2020* 26–37 (ACM, 2020).
19. Dalla Palma, S., Di Nucci, D., Palomba, F. & Tamburri, D. A. Within-Project Defect Prediction of Infrastructure as Code Using Product and Process Metrics. *IEEE TSE*. **48** (6), 2086–2104 (2022).
20. Borovits, N. et al. FindICI: Using Machine Learning to Detect Linguistic Inconsistencies between Code and Natural Language Documentation in Infrastructure as Code. *Empir. Softw. Eng.* **27**, 178 (2022).
21. Drosos, G-P., Sotiropoulos, T., Alexopoulos, G., Mitropoulos, D. & Su, Z. When Your Infrastructure Is a Buggy Program: Understanding Faults in Infrastructure as Code Ecosystems. Proc. ACM Program. Lang. 8(OOPSLA2), Article 359, 2490–2520. (2024). https://doi.org/10.1145/3689799
22. Nasiri, R., Kumara, I., Tamburri, D. A. & van den Heuvel, W-J. Toward a Taxonomy of Infrastructure as Code Misconfigurations: An Ansible Study. Symposium and Summer School on Service-Oriented Computing (SummerSOC), Springer, 83–103 (2024).
23. Sotiropoulos, T., Chaliasos, S., Mitropoulos, D. & Spinellis, D. *A Study of Bug Resolution Characteristics in Popular Programming Languages* (IEEE TSE, 2020).
24. Begoug, M., Chouchen, M. & Ouni, A. TerraMetrics: An Open Source Tool for Infrastructure-as-Code (IaC) Quality Metrics in Terraform. ICPC 2024, 450–454 (2024).
25. Begoug, M., Chouchen, M., Ouni, A., Alomar, E. A. & Mkaouer, M. W. Fine-Grained Just-in-Time Defect Prediction at the Block Level in Infrastructure-as-Code (IaC). MSR 2024, 100–112 (2024).
26. Begoug, M. et al. How Do IaC Practitioners Update Their Dependencies? An Empirical Study of Terraform Module Updates. MSR 2025 (2025).
27. Saavedra, N., Gonçalves, J., Henriques, M., Ferreira, J. F. & Mendes, A. Polyglot Code Smell Detection for Infrastructure as Code with GLITCH. ASE 2023, pp. 2042–2045 (2023).
28. Iosif, A-C., Gasiba, T. E., Zhao, T., Lechner, U. & Pinto-Albuquerque, M. *A Large-Scale Study on the Security Vulnerabilities of Cloud Deployments. UbiSec 2021 (Revised Selected Papers)* pp. 171–188 (Springer, 2022). https://doi.org/10.1007/978-981-19-0468-4_13
29. Verdet, A., Hamdaqa, M., Da Silva, L. & Khomh, F. Assessing the adoption of security policies by developers in Terraform across different cloud providers. *Empir. Softw. Eng.* https://doi.org/10.1007/s10664-024-10610-0 (2024).
30. Krause, A. et al. Smells-sus: Sustainability Smells in Infrastructure as Code. MSR 2025, IEEE/ACM (2025).
31. Tran, A-D., Sion, L., Yskout, K. & Joosen, W. TerrARA: Automated Security Threat Modeling for Infrastructure as Code. ACM CODASPY 2024, 269–280 (2024).

32. Sandobalín, J., Insfran, E. & Abrahão, S. On the Effectiveness of Tools to Support Infrastructure as Code: Model-Driven vs. *Code-Centric IEEE Access.* 8 (2020).
33. Guerriero, M., Garriga, M., Tamburri, D. A. & Palomba, F. Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. ICSME 2019 (2019).
34. Opdebeeck, R., Zerouali, A., Velázquez-Rodríguez, C. & De Roover, C. On the Practice of Semantic Versioning for Ansible Galaxy Roles: An Empirical Study. *Inf. Softw. Technol.* **141**, 106703 (2021).
35. Zerouali, A., Opdebeeck, R. & De Roover, C. Helm Charts for Kubernetes Applications: Evolution, Outdatedness and Security Risks. MSR 2023 (2023).
36. Hassan, M. M. & Rahman, A. As Code Testing: Characterizing Test Quality in Open Source Ansible Development. ICST 2022 (2022).
37. Li, K. et al. Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java. ESEC/FSE 2023, ACM (2023).
38. Lipp, S., Banescu, S. & Pretschner, A. An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection. ISSTA 2022, ACM, 544–555 (2022).
39. Charoenwet, W., Thongtanunam, P., Pham, V-T. & Treude, C. An Empirical Study of Static Analysis Tools for Secure Code Review. ISSTA 2024, ACM. (2024). https://doi.org/10.1145/3650212.3680313
40. Guo, Z. et al. Mitigating False Positive Static Analysis Warnings: Progress, Challenges, and Opportunities. *IEEE TSE*. **49** (12), 5154–5188 (2023).
41. Mehrpour, S. & LaToza, T. D. Can Static Analysis Tools Find More Defects? *Empir. Softw. Eng.* **28** (1), 5. https://doi.org/10.1007/s10664-022-10232-4 (2023).
42. Christakis, M. & Bird, C. What Developers Want and Need from Program Analysis: An Empirical Study. ASE 2016, ACM, 332–343 (2016).
43. Nachtigall, M., Schlichtig, M. & Bodden, E. A Large-Scale Study of Usability Criteria Addressed by Static Analysis Tools. ISSTA 2022, ACM, 532–543 (2022).
44. Mordahl, A. & Wei, S. *The Impact of Tool Configuration Spaces on the Evaluation of Configurable Taint Analysis for Android. ISSTA 2021* 466–477 (ACM, 2021).
45. Chouchen, M., Begoug, M. & Ouni, A. TerraDS: A Dataset for Terraform HCL Programs. MSR 2025, IEEE/ACM (2025).
46. Cohen, J. A Coefficient of Agreement for Nominal Scales. *Educ. Psychol. Meas.* **20** (1), 37–46 (1960).
47. Landis, J. R. & Koch, G. G. The Measurement of Observer Agreement for Categorical Data. *Biometrics* **33** (1), 159–174 (1977).

Figures

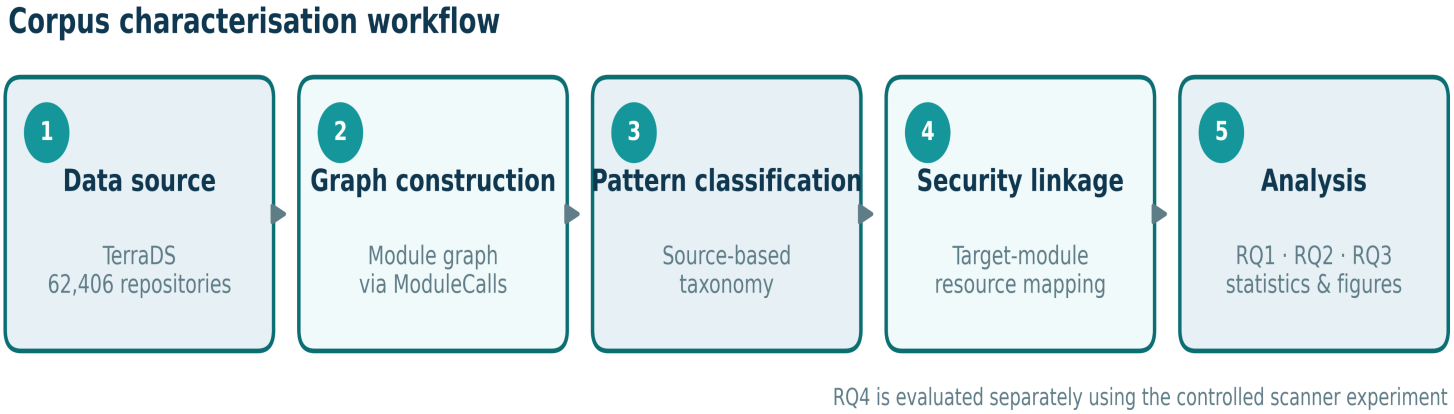


Figure 1

The five-stage corpus pipeline (RQ1 to RQ3)

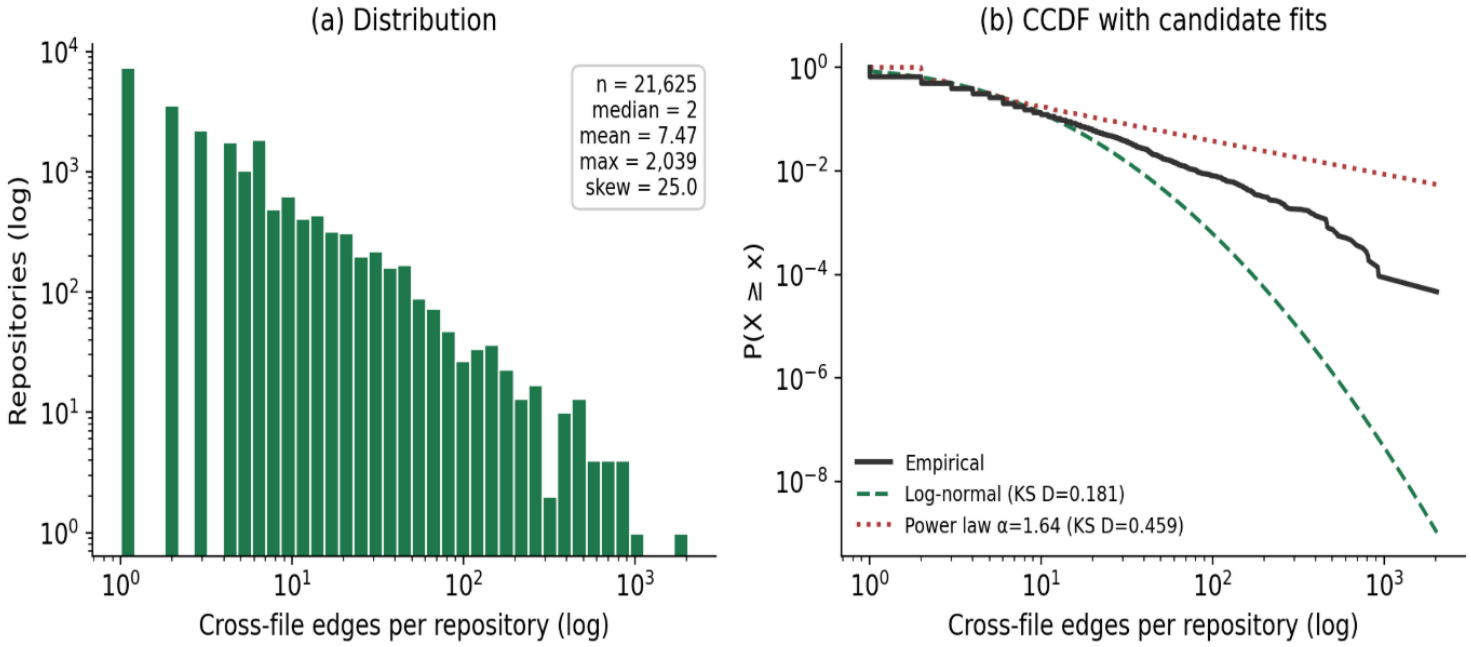


Figure 2

Cross-file dependencies per repository over the full corpus.

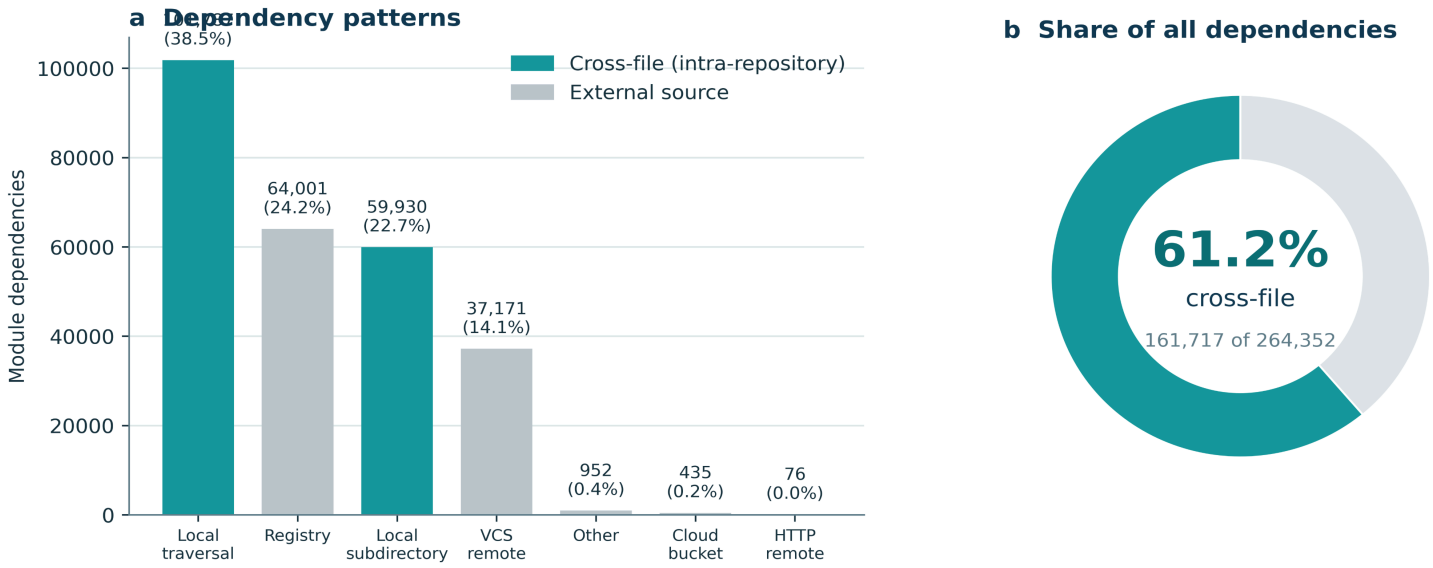


Figure 3

(a) Distribution of module dependencies across the seven taxonomy

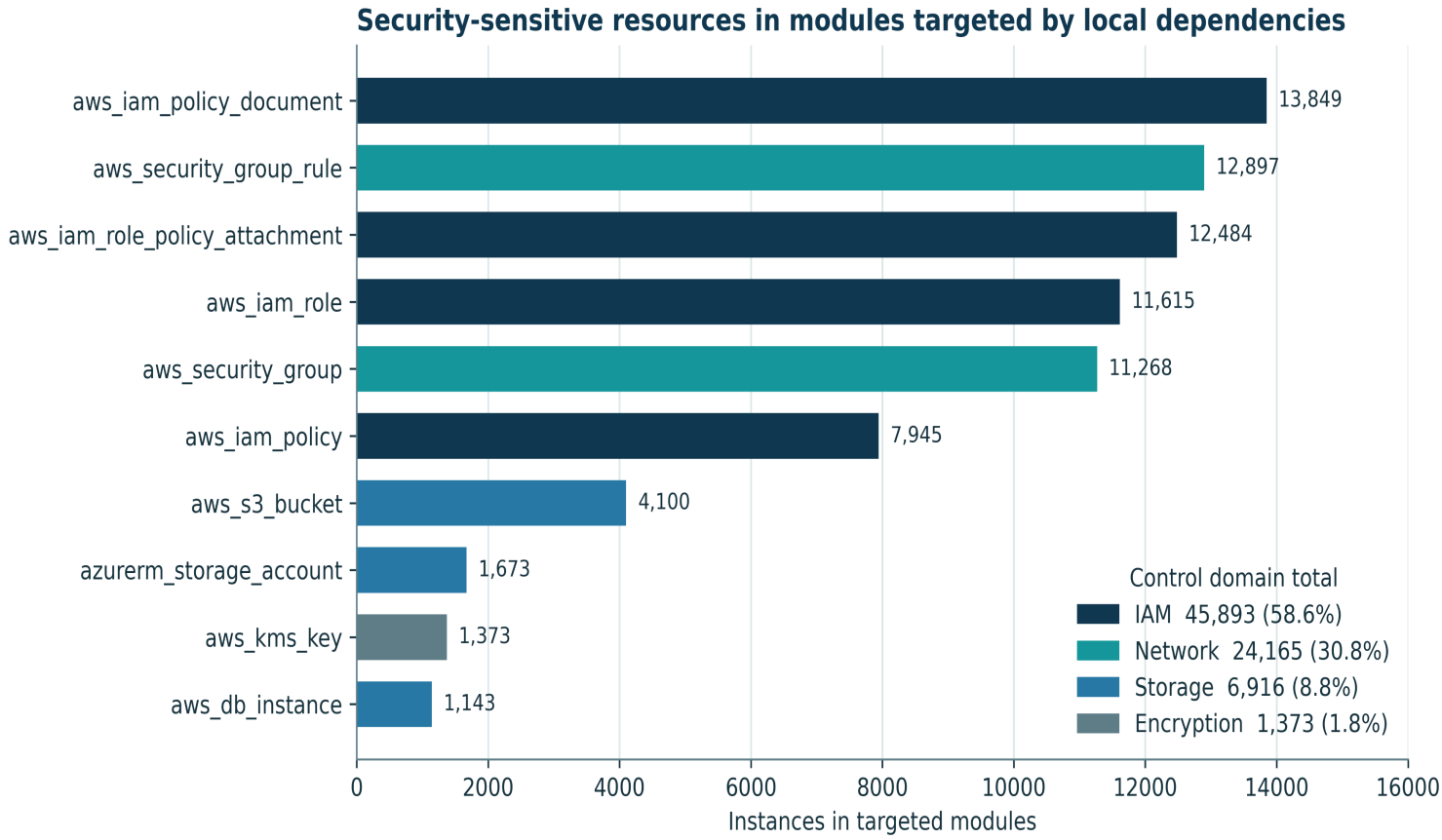


Figure 4

Security-sensitive resource types declared in modules targeted by a local dependency

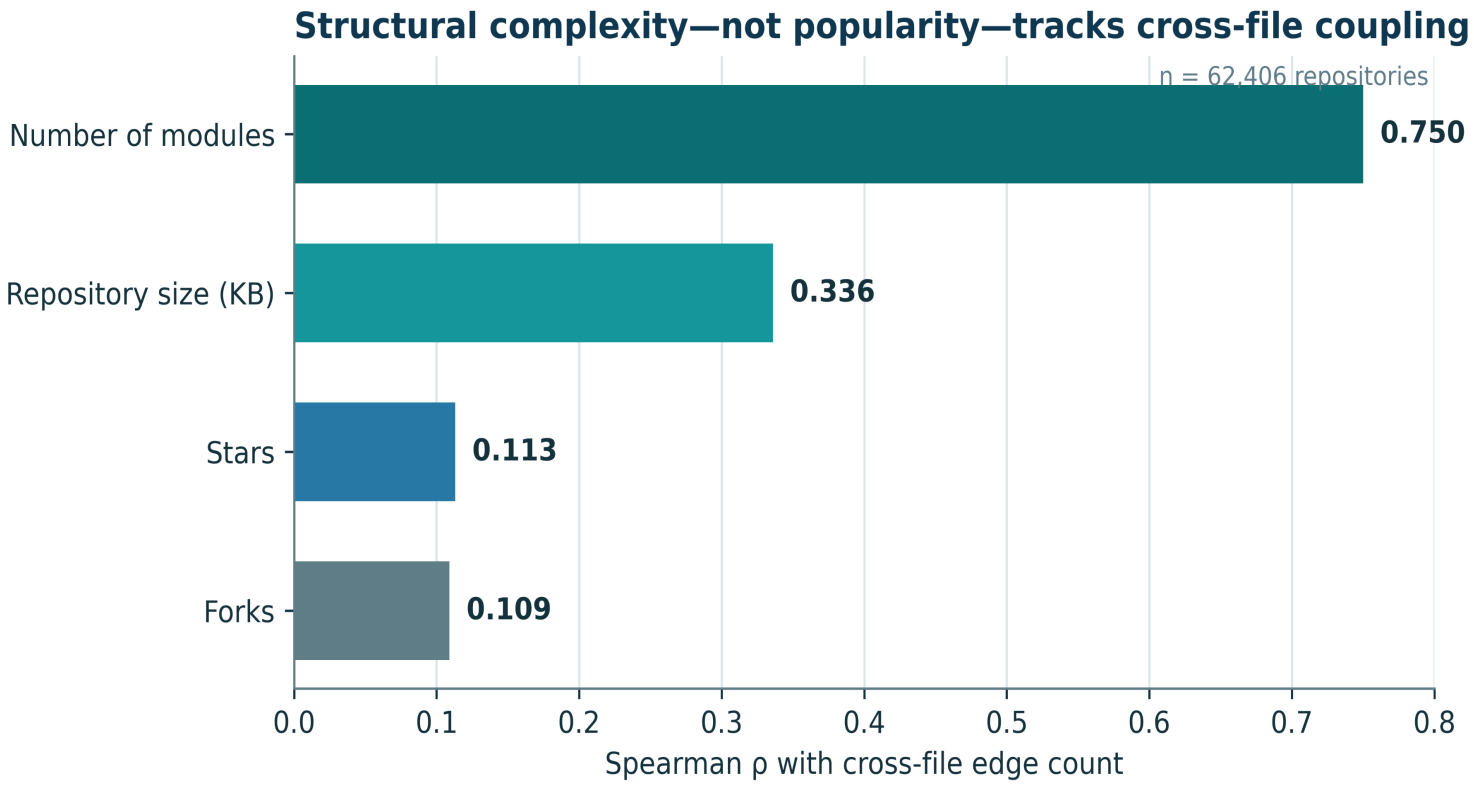


Figure 5

Rank correlation of cross-file edge count with repository characteristics.