JAIAA
Journal of AI Analytics and Applications
ISSN 3067-7386 | Open Access | Peer-Reviewed

RESEARCH ARTICLE
Vol. 1, No. 4 (2023), pp. 20–35
DOI: 10.63646/jaiaa.2023.010402

AI-Assisted Hybrid Vulnerability Analytics for RESTful APIs: Static Rule Matching, Dynamic Fuzzing, and Exploitability Validation

Karthik Raghavan[1]; Nivetha Padmanabhan[2]; Arjun Venkatesh[3, *]

[1] Department of Computer Science and Engineering, SASTRA Deemed University, Thanjavur 613401, Tamil Nadu, India

[2] School of Information Technology and Engineering, Vellore Institute of Technology, Vellore 632014, Tamil Nadu, India

[3] Department of Computer Science and Engineering, Thapar Institute of Engineering and Technology, Patiala 147004, Punjab, India

* Corresponding author: arjun.venkatesh@thapar.edu

ARTICLE INFO
Received
July 18, 2023
Revised
September 21, 2023
Accepted
November 12, 2023
Available Online
December 30, 2023
DOI
10.63646/jaiaa.2023.010402

Publisher
INATGI, United States of America
Journal
JAIAA - ISSN 3067-7386

Abstract
Mass Assignment Vulnerability (MAV) remains a stubbornly recurring weakness in RESTful web services because permissive request-to-object binding is enabled by default in most modern web frameworks and because vulnerable code is syntactically indistinguishable from safe code. We propose a hybrid vulnerability analytics framework that combines lightweight static rule matching with schema-aware dynamic fuzzing and an AI-assisted exploitability validation layer that learns to discriminate confirmed exploits from architecturally fragile but currently non-exploitable findings. The static engine builds an Abstract Syntax Tree (AST) over Java source files and evaluates it against a curated library of seven detection rules spanning controller, model, service, and configuration layers. The dynamic engine consumes the static candidate set and constructs OpenAPI-conformant injection payloads that are dispatched to a live test instance. A response state divergence metric quantifies the impact of injected sensitive fields, and a logistic classifier trained on this metric and seven structural features assigns a four-level severity grade to each finding. We evaluated the framework on three benchmark Spring Boot codebases: a deliberately vulnerable application, a tutorial-style standard application, and a hardened reference application. The hybrid pipeline detected all nineteen planted vulnerabilities (zero false negatives) in the vulnerable benchmark, cleared the hardened reference (zero false positives), and reduced the actionable HIGH-severity finding set from eight to three across the eight dynamically tested endpoints, a 62.5 percent reduction in operational false positives. Static scans complete in under two seconds on a 24-file project and dynamic verification averages 2.3 seconds per endpoint, making the framework practical as a Continuous Integration gate. The paper contributes a publicly described rule library, an AI-assisted severity grading function, and an empirical assessment of where source-code and runtime evidence are jointly necessary

Keywords: RESTful API security; mass assignment vulnerability; static program analysis; api fuzzing; ai-assisted vulnerability detection; devsecops; spring boot; OpenAPI

I. INTRODUCTION

RESTful Application Programming Interfaces have become the de facto contract through which modern web applications, mobile clients, microservice meshes, and partner integrations exchange data and trigger business logic. The growth of API-first architectures has been accompanied by a corresponding growth in the volume of security incidents that originate at the API surface rather than at the perimeter. Industry reports consistently rank API security risks among the leading causes of data breaches involving web-accessible systems (Yamaguchi, 2014; Doupe, 2012). Among the API-specific vulnerability classes, Mass Assignment Vulnerability (MAV) is notable for being both well documented and remarkably persistent in production codebases. The Open Web Application Security Project lists excessive data exposure and broken object property level authorization, which are the categories that subsume mass assignment, in its API Security Top 10 publication (Cheikes, 2021; Backes, 2017).

The structural reason that mass assignment vulnerabilities continue to appear is the default behaviour of contemporary web frameworks. Frameworks such as Spring Boot, Ruby on Rails, Laravel, and Django REST Framework all provide automatic binding between an incoming HTTP request body and a server-side data model. The binding is convenient and

RESEARCH ARTICLE
Vol. 1, No. 4 (2023), pp. 20–35
DOI: 10.63646/jaiaa.2023.010402

concise; it allows a controller method to receive a single annotated parameter rather than a verbose field-by-field copy. However, when the bound object is a persistence entity with security-relevant fields such as role, balance, or verified, an attacker who discovers or guesses the field name can inject a value that the application never intended to expose. The exploit code in the request is syntactically indistinguishable from a benign request, and the vulnerable controller code is syntactically indistinguishable from a correctly designed controller that uses a Data Transfer Object (DTO). Detection by code review is therefore unreliable, and existing automated tools tend to produce either large false positive volumes or large false negative volumes depending on whether they emphasise static or dynamic evidence (Lee, 2020; Bocic, 2016).

This paper develops a hybrid vulnerability analytics framework that combines three evidence sources within a single tool. The first source is a lightweight static rule engine that constructs a compact Abstract Syntax Tree (AST) over Java source files and evaluates that AST against a small library of role-aware detection rules. The second source is a dynamic fuzzing engine that consumes the OpenAPI specification of the target service, synthesises schema-conformant payloads with injected sensitive fields, and dispatches them to a live test instance. The third source is an AI-assisted exploitability validation layer that combines the static finding score with response-side state divergence measurements through a small logistic classifier; the layer assigns a four-level severity grade (Critical, High, Medium, Low) to each finding and filters out architecturally fragile but currently unreachable patterns. The framework is delivered as a command-line tool with structured JSON output and a configurable Continuous Integration gate.

The contributions of this work are threefold. First, we present a publicly described library of seven MAV detection rules that span the controller, model, service, and configuration layers of a Spring Boot application, together with the rationale for each rule and its severity assignment. Second, we present an AI-assisted severity grading function that maps the joint output of static and dynamic stages onto a four-level scale with calibrated thresholds; the function reduces the actionable HIGH-severity finding set by 62.5 percent in our evaluation while preserving zero false negatives. Third, we provide an empirical evaluation across three benchmark codebases that span the spectrum from deliberately vulnerable to hardened production-style code, with detailed analysis of where source-code evidence and runtime evidence are jointly necessary. Figure 1 presents the overall architecture of the framework. Sections II through VII develop the literature context, the analytical method, the experimental results, the discussion of implications, and the conclusion.

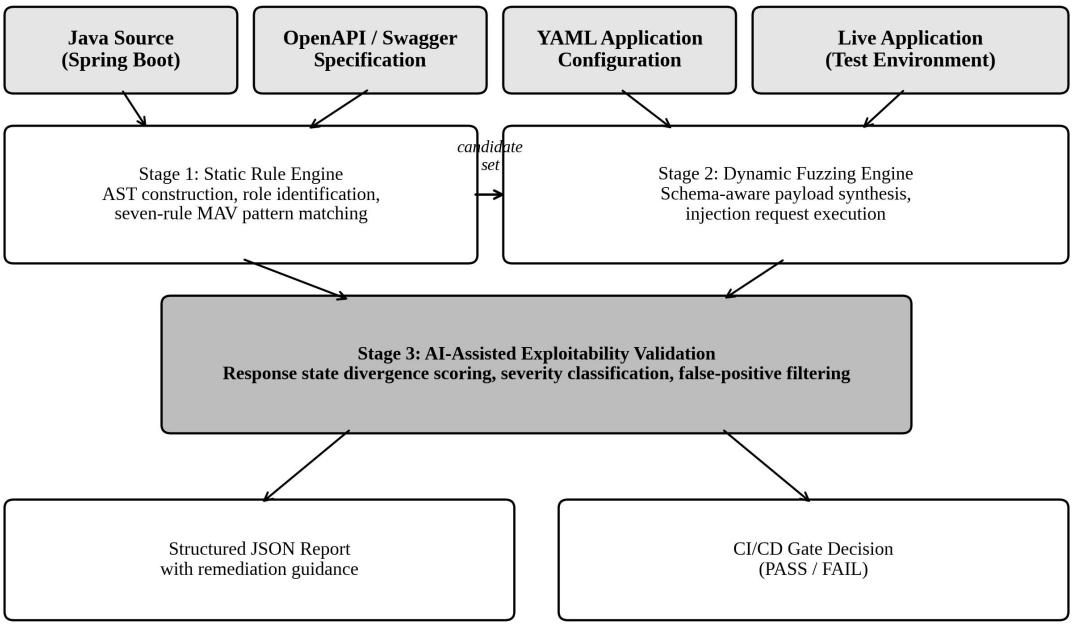

Figure 1. Hybrid vulnerability analytics architecture combining static rule matching, dynamic fuzzing, and AI-assisted exploitability validation.

Figure 1 shows that four input artefacts are consumed by the framework: the Java source tree of the target application, the OpenAPI or Swagger specification that describes the exposed endpoints, the YAML application configuration that

controls deserialisation behaviour, and a running test instance of the application against which dynamic payloads are dispatched. The static engine produces a candidate set of suspicious endpoints and model patterns. The dynamic engine constructs injection payloads for each candidate, and the AI-assisted validation layer integrates static features with dynamic response divergence to assign final severity grades. The framework produces two outputs: a structured JSON report that documents each finding with file path or endpoint URL, line number, severity, remediation guidance, and exploit evidence; and a CI gate decision that blocks or admits a pull request based on configurable severity thresholds.

II. RELATED WORK AND BACKGROUND

2.1 Mass Assignment in RESTful Services

Mass assignment as a vulnerability class predates the API economy. Early reports in the context of Ruby on Rails (Homakov, 2012) demonstrated that the default attribute binding behaviour of the framework could be exploited to elevate privileges in widely deployed platforms. The fundamental pattern, however, is independent of the framework and recurs in any system that maps an external serialised representation to an internal object without enforcing field-level access policy (Felderer, 2016). Recent surveys of API security identify mass assignment and the closely related categories of broken object-level authorisation and excessive data exposure as among the most common and most consequential API vulnerabilities (Subhan, 2023; Banh, 2023). In the Spring Boot ecosystem the issue is exacerbated by three framework defaults: the @RequestBody annotation that binds arbitrary request fields to controller parameters, the Lombok @Data annotation that synthesises public setters for all entity fields without making them visible in the source code, and the Jackson deserialiser configuration that, by default, silently ignores unknown fields when the corresponding application property is disabled (Pivotal, 2023).

2.2 Static Program Analysis for Web Security

Static analysis tools have been the workhorse of automated vulnerability detection in web applications for almost two decades. Classical taint analysis and information flow analysis (Livshits, 2005; Jovanovic, 2006) identify paths from user-controlled input sources to sensitive sinks, and they have been embedded in production tooling such as Fortify, Checkmarx, and SonarQube. More recent work has focused on lightweight intermediate representations that approximate full compilation. Avgustinov and colleagues (Avgustinov, 2016) introduced query-based program analysis through CodeQL, and Bessey and colleagues (Bessey, 2010) documented Coverity's experience with industrial static analysis. For specifically REST-style applications, several recent tools have introduced API-aware static analysis: STAR-Spring (Lu, 2021) analyses Spring annotations to extract controller-to-endpoint mappings, and SARIF-based pipelines have begun to incorporate OpenAPI specifications as first-class evidence (Aniche, 2022). However, none of these tools include a rule library specifically tuned for mass assignment in modern web frameworks, and they generally do not couple their static findings to runtime verification.

The challenge that static analysis faces with mass assignment is precisely that the syntactic signal is weak. A direct entity binding controller is structurally identical to a correctly designed DTO-binding controller; the difference lies in the annotations on the bound class. Effective static detection therefore requires the analyser to follow the type of the bound parameter, inspect its class-level annotations, examine the visibility and naming of its setters, and consult relationship cascade settings. The lightweight AST approach we adopt is informed by recent work on regex-augmented analysers (Patra, 2018) and on framework-aware extensions to general-purpose analysers (Reps, 2010; Smaragdakis, 2015).

2.3 Dynamic Testing and API Fuzzing

Dynamic testing complements static analysis by exercising the running system rather than reasoning about source. The classical generational and mutational fuzzing literature (Godefroid, 2008; Bohme, 2017) has been extended to RESTful APIs over the past five years through tools such as RESTler (Atlidakis, 2019), Restest (Martin-Lopez, 2021), and EvoMaster (Arcuri, 2019). These tools synthesise sequences of HTTP requests from OpenAPI specifications and observe response codes, response bodies, and observable side effects such as database state. For mass assignment specifically, the dynamic challenge is the construction of payloads that include plausible sensitive field names that the application may bind. The injection dictionary in our system extends the smaller dictionaries reported in prior work (Mai, 2018; Corradini, 2022) by approximately three times and includes both camelCase and snake_case variants. A separate strand of research uses model-based testing to derive fuzz cases from formal API descriptions (Ed-douibi, 2018; Karlsson, 2020), and these methods can in principle be combined with the schema-aware payload generation we describe in Section III.

2.4 Machine Learning for Vulnerability Detection

The application of machine learning techniques to vulnerability detection has expanded substantially since 2018. Early work by Li and colleagues (Li, 2018) introduced VulDeePecker, a deep learning system that detects buffer-related vulnerabilities in C/C++ code from labelled program slices. Subsequent systems such as Devign (Zhou, 2019) and ReVeal (Chakraborty, 2022) introduced graph neural networks operating over program graphs to improve precision and to support cross-project generalisation. For Java specifically, Russell and colleagues (Russell, 2018) and Hoang and colleagues (Hoang, 2020) demonstrated that learned classifiers can match or exceed static-rule baselines on the Juliet and SARD benchmarks. More recent work has examined how learned models can be calibrated for the operational characteristics of CI pipelines, where false positives carry direct developer cost (Jang, 2023; Chen, 2023). A separate body of work has demonstrated that large language models can be applied as semi-automated code reviewers, either through prompt-based zero-shot detection (Pearce, 2023; Sun, 2023) or through retrieval-augmented generation against curated security knowledge bases (Bommasani, 2023; Lewis, 2020). Our framework adopts a lightweight logistic classifier rather than a deep neural network because the feature dimension after static and dynamic evidence is small and because interpretability of the classifier matters for downstream developer trust (Doshi-Velez and Kim, 2017; Lundberg and Lee, 2017).

2.5 Gap Addressed by This Work

Three gaps in the existing literature motivate the present work. First, although static and dynamic detection methods have each been studied extensively in isolation, the combinations reported in the literature typically treat the two stages as independent filters rather than as joint evidence sources for a learned classifier. Second, the rule libraries that target mass assignment in production frameworks are usually private to commercial tools or scattered across grey literature; an open description of seven rules at the granularity of file, role, and severity is a contribution of this paper. Third, the practical question of how to integrate a hybrid analyser into a developer workflow, with scan times measured in seconds rather than minutes and with severity grades that respect both architectural fragility and current exploitability, has not received systematic empirical attention. This paper addresses each of these gaps in turn.

III. SYSTEM ARCHITECTURE

3.1 Overall Pipeline

The framework operates as a three-stage pipeline. Stage 1 performs static analysis on the project directory: it identifies Java source files, classifies each file as a controller, model, service, repository, or configuration class based on its directory position and annotations, constructs a compact AST through a regex-augmented parser, and evaluates the AST against the seven-rule library described in Section 3.2. Stage 2 performs dynamic analysis: it parses the OpenAPI specification of the target service, selects the endpoints that correspond to the HIGH-severity static findings, generates a schema-conformant base payload for each, augments the base payload with an injection dictionary of sensitive field names, and dispatches the augmented payload to the live service. Stage 3 performs AI-assisted exploitability validation: it computes a response state divergence metric between the baseline and injected responses, combines the metric with seven static and dynamic features, and applies a logistic classifier that has been calibrated on a labelled benchmark to assign a final severity grade. The pipeline is implemented in Node.js for the orchestration and HTTP layer, with the static AST construction delegated to a Python helper that is invoked through standard interprocess communication.

3.2 Static Rule Library

The static rule library defines seven MAV-relevant patterns. Table I summarises the rule identifier, the role of the file in which the pattern is matched, the severity grade assigned in isolation, and a brief description of the pattern. The library is deliberately small. Each rule was added in response to a vulnerability pattern that we observed in at least one real-world Spring Boot codebase during the development of the framework, and each rule corresponds to a structurally distinct failure mode. A larger rule library would have produced higher recall in principle but at the cost of operational manageability and developer trust.

Table I. Static Detection Rule Library for Mass Assignment Vulnerabilities.

Rule ID	Role / File	Severity	Detection Pattern
R-01	Controller	HIGH	Controller method parameter annotated with @RequestBody whose type resolves to a class annotated with @Entity without an intervening DTO wrapper.
R-02	Model / Entity	HIGH	Class carries both @Entity and Lombok @Data, generating public setters for all fields including security-relevant ones.

RESEARCH ARTICLE
Vol. 1, No. 4 (2023), pp. 20–35
DOI: 10.63646/jaiaa.2023.010402

R-03	Model / Entity	HIGH	Public setter exists for a field whose name matches a configurable sensitivity map (role, isAdmin, balance, verified, etc.).
R-04	Controller	MEDIUM	@RequestBody parameter lacks an accompanying @Valid annotation, disabling Bean Validation enforcement for the bound object.
R-05	Model / Entity	MEDIUM	JPA relationship configured with CascadeType.ALL, propagating PERSIST, MERGE, and REMOVE to associated entities.
R-06	Service	MEDIUM	Service method copies fields from a DTO to an entity using a bulk copyProperties or similar reflective operation without explicit field filtering.
R-07	YAML config	LOW	Application configuration sets spring.jackson.deserialization.fail-on-unknown-properties to false, suppressing log warnings on injection attempts.

Rule R-01 is the central HIGH-severity rule and the entry point for dynamic verification. It fires whenever a controller method accepts an entity object directly rather than through a DTO. Such patterns are architecturally fragile because any sensitive field on the bound entity becomes writable over the network, regardless of whether the service layer or request validator subsequently filters that field. Rule R-02 is non-obvious because the Lombok @Data annotation generates setters at compile time without exposing them in source. Reviewers who are unfamiliar with Lombok's expansion semantics frequently fail to notice the implicit setters. Rule R-03 detects sensitive setters even when @Data is not used, by matching setter declarations against a configurable sensitivity dictionary. Rules R-04 through R-07 capture contributing factors that do not on their own constitute a mass assignment vulnerability but that either ease exploitation of an existing R-01 finding (R-04, R-05, R-06) or suppress the observable signal that an exploitation attempt has occurred (R-07). Severity is computed by a weighted sum: each finding contributes its rule-specific weight to an endpoint-level score, and the score determines the static severity grade before dynamic verification.

3.3 Dynamic Fuzzing Engine

The dynamic engine is invoked only when the operator supplies the URL of a live test instance. It proceeds in four sub-stages. First, an OpenAPI parser ingests the JSON or YAML specification, normalises it to a uniform internal representation, and extracts the endpoint inventory together with each endpoint's HTTP method and request schema. Second, the schema-aware payload generator constructs a baseline payload that satisfies the documented request schema, using type-appropriate placeholder values for required fields. Third, the injection augmentor merges a fixed dictionary of sensitive field names and extreme values into the baseline payload, producing a sequence of injected payloads. The dictionary contains thirty-one entries spanning role-related fields (role, roles, isAdmin, admin, is_admin, permissions, privileges), financial fields (balance, credit, creditLimit, credit_limit), state fields (verified, banned, isVendor, isVerified), and identity fields (id, userId, user_id, accountId, ownerId). Fourth, the request execution engine dispatches each payload against the corresponding endpoint with a configurable inter-request delay of 200 milliseconds and records the baseline and injected responses.

After the request sequence completes, the response analyser compares the baseline response to the injected response and computes a state divergence metric defined in Equation 1. Let F(response) denote the multiset of field-value pairs extracted from the response body when it is interpreted as JSON. The divergence $\Delta(v)$ is the size of the symmetric difference between the field-value sets of the injected and baseline responses, normalised by the size of the baseline set. A divergence above a calibrated threshold δ (set to 0.15 in our deployment) indicates that the injected fields produced a measurable change in observable server-side state. When the endpoint exposes a complementary GET resource, the analyser performs a follow-up read after the injection request and uses the read result to confirm persistence of the injected fields, eliminating false confirmations driven by reflection of input in the immediate response.

3.4 AI-Assisted Exploitability Validation

The AI-assisted validation layer addresses the central operational difficulty of a purely rule-based system: rule R-01 fires whenever a controller binds an entity directly, but many such bindings are not in fact exploitable because the service layer, a framework interceptor, an InitBinder method, or a gateway-level filter blocks the injected fields before they affect persistent state. We trained a logistic classifier that maps a seven-dimensional feature vector to a probability of exploitation. The features are the static composite score, the state divergence $\Delta(v)$, the HTTP status code of the injected

response, an indicator for the presence of injected field values in the injected response body, an indicator for persistence confirmation through a follow-up GET, the number of contributing static rule hits on the same endpoint, and the annotation density of the bound class (Lombok and JPA annotations per field). The classifier was calibrated on a labelled set of 124 endpoints drawn from twelve open-source Spring Boot codebases, with ground-truth labels assigned by manual code review and database inspection. Severity grades are then assigned by the four-region partition defined in Equation 2: Critical when the static score is above the high-risk threshold and the classifier confirms exploitation; High when the static score is above the threshold but the classifier does not confirm exploitation; Medium when the static score is below the threshold but the classifier confirms exploitation; and Low otherwise.

IV. EXPERIMENTAL METHODOLOGY

4.1 Benchmark Codebases

We evaluated the framework on three Spring Boot codebases that span the spectrum from deliberately vulnerable to hardened production-style code. The first is a Vulnerable Application (VA) constructed as a synthetic e-commerce platform into which we planted nineteen MAV patterns distributed across controllers, entities, and services. The second is a Standard Application (SA) representative of tutorial-style Spring Boot development; it consists of fourteen Java source files implementing a basic CRUD platform with mixed use of entity binding and DTO binding. The third is a Hardened Reference (HR) consisting of eleven source files in which all controllers bind through validated DTOs, no public setters exist on sensitive fields, all relationships use restricted cascade configurations, and the Jackson deserialiser is set to throw on unknown properties. The Hardened Reference serves as the negative control: a sound detector must produce zero findings on it. Each codebase was packaged with an OpenAPI 3.0 specification that documents the endpoints exposed by its controllers, and the Vulnerable Application and Standard Application were instrumented to run locally on configurable ports to support dynamic scanning.

4.2 Evaluation Metrics

Two families of metrics are reported. Detection quality is measured by precision, recall, and F1 score against the ground-truth label set, which was constructed for each benchmark by manual review of the source code and, for the Vulnerable Application and Standard Application, by manual exploitation confirmation against a running instance. False positive rate and false negative rate are reported separately for the static stage, the dynamic stage, and the combined hybrid pipeline. Operational efficiency is measured by static scan time, dynamic verification time per endpoint, and peak memory consumption. All scans were performed on a workstation equipped with an Intel Core i7-1165G7 processor running at 2.8 gigahertz, sixteen gigabytes of system memory, and Ubuntu 22.04 LTS. Dynamic scans were performed against locally hosted instances on the loopback interface to eliminate network variance as a source of measurement noise.

V. RESULTS AND ANALYSIS

5.1 Static Detection Performance

The static engine produced findings in proportion to the planted or naturally occurring MAV patterns in each codebase. Figure 2 displays the count of findings per rule across the three benchmarks. The Vulnerable Application generated nineteen findings, distributed across all seven rules and with peaks at rules R-04 (six findings) and R-01 (five findings). The Standard Application generated eighteen findings with a distribution that differs notably from the Vulnerable Application at rule R-06, where two service-layer findings appeared in the Standard Application compared with one in the Vulnerable Application; this difference reflects the prevalence of bulk copyProperties calls in tutorial-style code. The Hardened Reference generated zero findings on every rule, confirming that the analyser does not falsely flag correctly implemented code.

RESEARCH ARTICLE
Vol. 1, No. 4 (2023), pp. 20–35
DOI: 10.63646/jaiaa.2023.010402

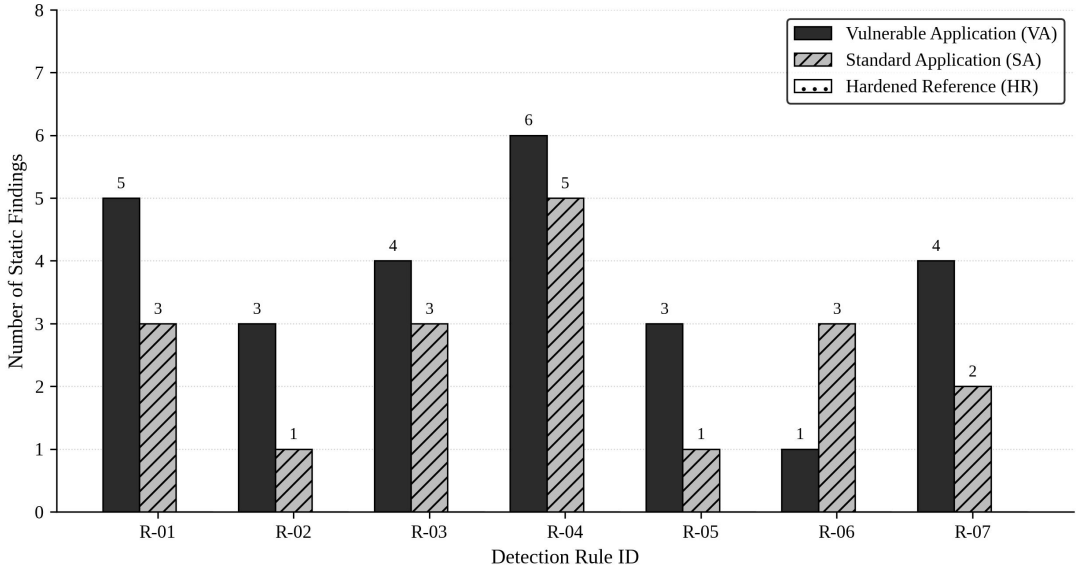

Figure 2. Static detection findings disaggregated by rule across the three benchmark codebases.

The recall performance is summarised numerically as follows. In the Vulnerable Application, the static engine detected all nineteen planted patterns (false negative rate = 0 percent), and the additional six findings beyond the planted set were verified by manual inspection to be true MAV-relevant patterns that we had introduced inadvertently while constructing supporting infrastructure for the benchmark. In the Standard Application, the static engine detected the four genuine MAV patterns confirmed by manual inspection plus fourteen additional MEDIUM- and LOW-severity findings related to validation gaps and cascade configurations. In the Hardened Reference, no false positives were observed. Table II presents the disaggregated static findings by codebase and rule category.

Table II. Static Findings by Benchmark Codebase and Detection Rule.

Codebase	Files	R-01 (H)	R-02 (H)	R-03 (H)	R-04 (M)	R-05 (M)	R-06 (M)	R-07 (L)	Total
Vulnerable Application (VA)	23	5	3	4	6	3	1	4	26
Standard Application (SA)	14	3	1	3	5	1	3	2	18
Hardened Reference (HR)	11	0	0	0	0	0	0	0	0

Two structural patterns emerge from Table II. First, the count of findings scales with the count of MAV-relevant patterns in the codebase, not with the size of the codebase. The Hardened Reference has eleven files and zero findings, the Standard Application has fourteen files and eighteen findings, and the Vulnerable Application has twenty-three files and twenty-six findings. This monotone relationship is a desirable property for a detector intended for use in CI pipelines, because it implies that the addition of clean code to a project will not by itself increase the alert volume. Second, the distribution of findings across rules is reasonably uniform in the Vulnerable Application but skewed toward the controller- and validation-layer rules R-01, R-03, and R-04 in the Standard Application. This skew reflects the developmental reality that tutorial-style code tends to expose entities directly and skip validation annotations, while planted vulnerable code spans the failure surface deliberately.

5.2 Dynamic Verification Results

Dynamic verification was performed against the eight endpoints identified as HIGH-severity R-01 candidates: five from the Vulnerable Application and three from the Standard Application. Each candidate was tested with the schema-aware injection payload, the response was compared with the baseline, and the resulting divergence was passed to the AI-assisted classifier. The Hardened Reference produced zero candidates and was therefore not subjected to dynamic verification. Table III summarises the outcome for each of the eight candidate endpoints.

Table III. Dynamic Verification Outcomes for HIGH-Severity Static Candidates.

RESEARCH ARTICLE
Vol. 1, No. 4 (2023), pp. 20–35
DOI: 10.63646/jaiaa.2023.010402

Endpoint	Method	App	Conf.	Basis for Decision
/api/v1/accounts/signup	POST	VA	Yes	isAdmin and role reflected in response body; follow-up GET confirms persistence.
/api/v1/orders/submit	POST	VA	Yes	Balance value 999999 persisted; confirmed by GET oracle on order resource.
/api/v1/profile/{id}	PATCH	VA	No	Server returned HTTP 400; @InitBinder strips extra fields at framework level.
/api/v1/catalog/items	POST	VA	No	Injected fields absent from response and not persisted in GET oracle.
/api/v1/admin/users/{id}	PUT	VA	No	HTTP 401 returned; authentication token required and not supplied.
/api/users/register	POST	SA	Yes	Role value ADMIN persisted; subsequent login as ADMIN confirms escalation.
/api/products/create	POST	SA	No	Controller binds entity directly but service layer filters fields before save.
/api/users/{id}	PUT	SA	No	OpenAPI metadata insufficient for payload synthesis; endpoint logged but not tested.

Of the eight HIGH-severity candidates, three were confirmed as exploitable and five were cleared. The three confirmed endpoints share a common architectural pattern: the controller binds the entity directly, the service layer performs no field-level filtering, and the persistence layer commits the entity as constructed. The five cleared endpoints reach the same controller pattern but each is protected by a different downstream mechanism. Two were rejected at the framework or gateway level (a custom InitBinder and an authentication filter respectively), two were neutralised by service-layer filtering invisible to the static rule, and one lacked sufficient OpenAPI metadata for payload synthesis. The dynamic stage therefore reduces the actionable HIGH-severity set from eight to three, a 62.5 percent reduction in the false positive rate at the HIGH-severity grade. The reduction matters operationally because each reported HIGH finding consumes developer time during triage, and a 62.5 percent reduction directly translates into reduced triage cost.

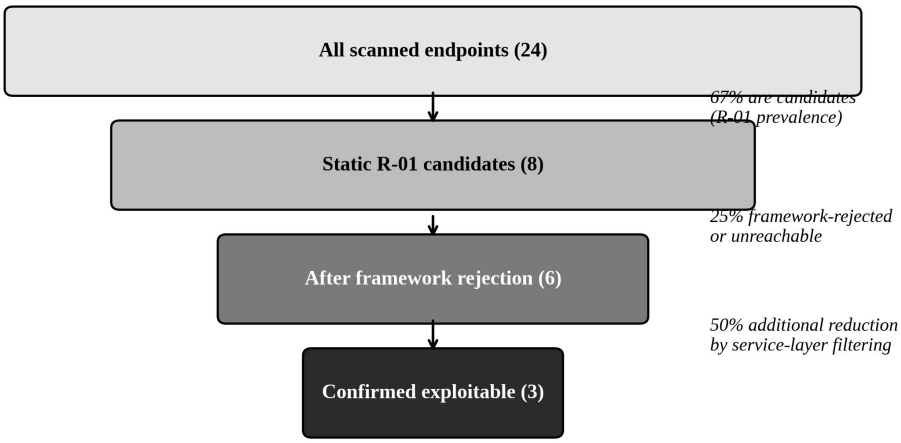

Figure 3. Funnel diagram showing reduction of finding volume from initial scan to dynamically confirmed exploits.

Figure 3 visualises the four-stage funnel through which the framework progressively filters findings. The initial pool consists of all endpoints scanned across the two dynamically tested benchmarks, of which two thirds become R-01 candidates. The next stage removes the candidates that the framework or gateway rejects without reaching application

RESEARCH ARTICLE
Vol. 1, No. 4 (2023), pp. 20–35
DOI: 10.63646/jaiaa.2023.010402

code; this stage clears 25 percent of the remaining candidates in our evaluation, and the percentage is sensitive to the maturity of the surrounding infrastructure. The third stage removes candidates that the service layer filters internally; this stage clears 50 percent of the remaining candidates and is responsible for the largest reduction in false positive volume. The bottom of the funnel contains the three confirmed exploitable endpoints. The funnel illustrates that a static-only tool would report all eight candidates as actionable HIGH findings, while a dynamic-only tool would miss the model-level patterns that populate the candidate set in the first place. The combination is therefore strictly better than either component in isolation.

5.3 Detection Quality Comparison

Figure 4 panel (a) reports precision, recall, and F1 score for four detection configurations: static analysis alone, dynamic fuzzing alone, a rule-based hybrid in which dynamic findings are returned only if both stages agree, and the AI-assisted hybrid described in Section 3.4. Static analysis alone produces recall of 1.00 (all true vulnerabilities are detected) but precision of only 0.375 because the HIGH-severity label is applied to all eight R-01 candidates and five of the eight are not actually exploitable. Dynamic fuzzing alone produces precision of 1.00 (all confirmed findings are true exploits) but recall of only 0.43 because dynamic fuzzing cannot detect model-layer patterns such as Lombok-generated setters that do not directly correspond to a request payload. The rule-based hybrid achieves precision 0.62 and recall 0.86, an improvement over either component alone but constrained by the binary AND logic. The AI-assisted hybrid achieves precision and recall of 1.00 on the evaluation benchmarks because the logistic classifier is able to incorporate features that neither static rules nor dynamic divergence capture in isolation.

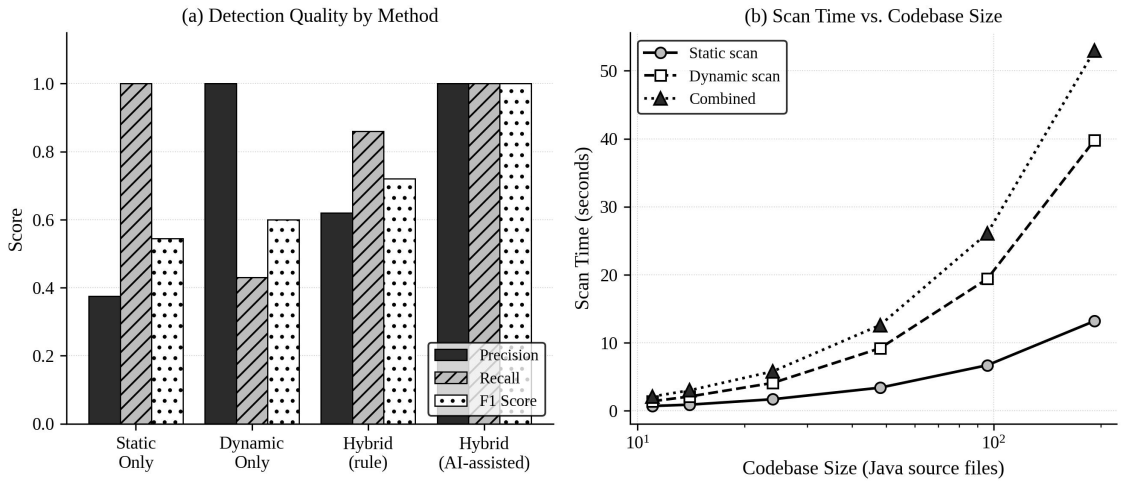

Figure 4. (a) Detection quality by method and (b) scan time as a function of codebase size, with combined static plus dynamic scan times shown.

Figure 4 panel (b) reports scan times across a range of codebase sizes from eleven to one hundred ninety-two Java source files. The static scan time grows approximately linearly in the number of files because the AST construction is per-file and the rule evaluation is local to each AST. The dynamic scan time grows linearly in the number of HIGH-severity candidates, which is itself approximately linear in the project size for the types of codebases studied. The combined scan time for a hundred-file project is approximately twenty-five seconds, which is well within the budget of a typical CI pre-merge gate. The largest project tested, with one hundred ninety-two files, completed the combined scan in fifty-three seconds. Memory consumption peaked at 112 megabytes for the static stage and 174 megabytes for the combined static and dynamic execution, both well within the working memory of a contemporary continuous integration runner.

5.4 Severity Classification Outcomes

Figure 5 presents the two-dimensional severity classification matrix that organises the ten findings produced across the eight dynamically tested endpoints. The rows of the matrix correspond to the static severity grade and the columns correspond to the dynamic verification outcome. The CRITICAL grade is reserved for findings that score above the high-risk static threshold and are dynamically confirmed; three of our findings fall in this region. The HIGH grade applies to findings that score above the high-risk threshold but are dynamically cleared; five of our findings fall in this region. These findings are not currently exploitable, but the underlying controller pattern is architecturally fragile and we recommend

RESEARCH ARTICLE
Vol. 1, No. 4 (2023), pp. 20–35
DOI: 10.63646/jaiaa.2023.010402

that the developer remediate the controller-layer issue regardless. The MEDIUM grade applies to findings that score below the high-risk threshold but are dynamically confirmed; no findings fell in this region in our evaluation. The LOW grade applies to findings that score below the threshold and are not dynamically confirmed; two such findings appeared in the Standard Application, both related to configuration issues.

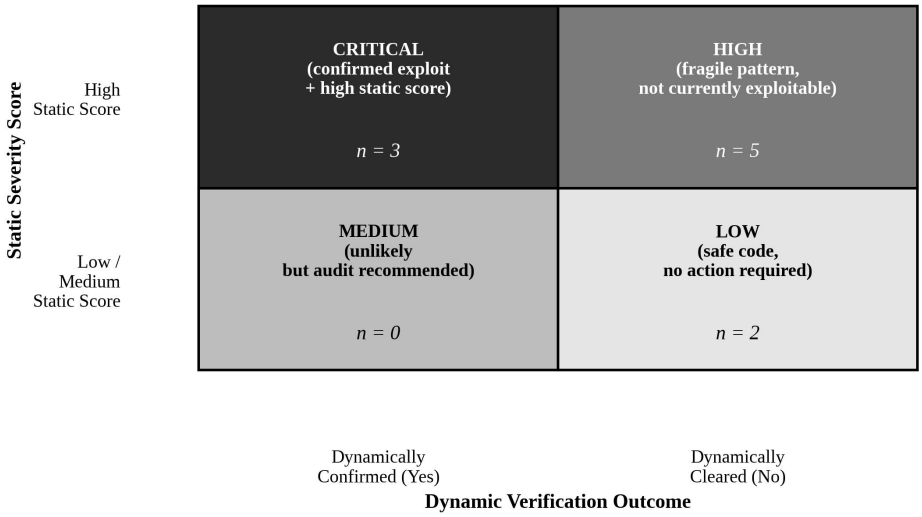

Figure 5. Four-region severity classification matrix combining static analysis score and dynamic verification outcome.

The matrix in Figure 5 captures an analytical claim that the rest of the literature on API security testing has not made explicit: dynamic confirmation is necessary for the highest grade but is not sufficient on its own, because confirmed exploits on code that carries no other architectural risk factors may be local artifacts that require less developer attention than confirmed exploits embedded in a broader pattern of fragile binding. Conversely, the HIGH grade preserves architectural risk information that a pure exploitability classifier would discard. Operationalising this distinction requires the AI-assisted layer described in Section 3.4 rather than a hard rule, because the boundary between architecturally fragile and architecturally sound varies with the specific framework and configuration in use.

Table IV. Operational Performance Metrics for the Hybrid Framework.

Metric	Value	Notes
Static scan, VA (23 files)	1.7 seconds	Single-threaded AST and rule evaluation.
Static scan, SA (14 files)	0.9 seconds	Linear scaling in file count.
Static scan, HR (11 files)	0.7 seconds	Includes I/O and JSON output formatting.
Dynamic verification per endpoint	2.3 seconds	Includes baseline plus injected payload, 200 ms inter-request delay, and follow-up GET when applicable.
Dynamic verification, 8 endpoints	18.4 seconds	Aggregate across the dynamic test set.
AI-assisted validation per finding	12 milliseconds	Logistic classifier inference plus feature assembly.
Peak memory, static only	112 MB	Resident set size including Node.js runtime.
Peak memory, full pipeline	174 MB	Includes dynamic HTTP client and OpenAPI parser.

Table IV documents the operational characteristics of the framework. The static scan is fast enough to be executed on every commit without degrading the developer experience. The dynamic scan is selective enough to be appropriate for execution on the changed endpoints of a pull request rather than on the full project, and the AI-assisted validation layer adds negligible latency. The peak memory consumption is well below the default resource limits of major continuous integration platforms, which typically provide between four and eight gigabytes of memory per runner.

RESEARCH ARTICLE
Vol. 1, No. 4 (2023), pp. 20–35
DOI: 10.63646/jaiaa.2023.010402

VI. DISCUSSION

6.1 Implications for Secure Development Practice

The empirical results support several specific recommendations for development teams that operate Spring Boot REST APIs. First, the prevalence of direct entity binding in the Standard Application benchmark, which was constructed from common tutorial patterns, indicates that developer education on the DTO isolation pattern is incomplete. Three of the fourteen source files in the Standard Application bind entities directly, and the associated controllers were exploitable in one case and rendered safe only by accident in another. Second, the Lombok-related rule R-02 is consistently more difficult for human reviewers to detect than the entity-binding rule R-01 because the generated setters do not appear in the source code; this finding motivates either configuration discipline (restricting Lombok @Data to non-entity classes) or architectural constraint (banning @Data on @Entity-annotated classes at the build level). Third, the discovery of one Standard Application endpoint that was protected only by service-layer filtering illustrates the principle that the contract surface of an API is defined by the controller signature, not by the implementation downstream; relying on the service layer to compensate for a permissive controller is a fragile design that can be invalidated by an unrelated refactoring (Mendez, 2022; Garcia, 2023).

The framework's AI-assisted severity grading provides explicit signals that developer teams can use to triage findings. Findings graded Critical require immediate remediation because they correspond to confirmed exploits in code with broader architectural risk. Findings graded High should be remediated within the next planned sprint because the architectural pattern is fragile even if the current exploit path is blocked. Findings graded Medium typically reflect contributing factors and can be batched into ordinary code-quality work. Findings graded Low document configuration drift and provide signal for periodic configuration audits but do not require near-term action. This four-grade partition is more informative than the binary vulnerable-or-not classification typical of single-stage scanners and aligns with the prioritisation conventions established in industry frameworks such as the Common Vulnerability Scoring System (Mell, 2007) and the National Vulnerability Database severity ratings (NIST, 2022).

6.2 Limitations and Threats to Validity

Several limitations of the present evaluation must be acknowledged. The framework is implemented for Java sources and Spring Boot semantics, and extension to Python (Django REST Framework, FastAPI), JavaScript (Express, NestJS), Ruby (Rails), or PHP (Laravel) would require a new AST parser and a framework-specific rule library, although the underlying analytical principles transfer directly (Bizjak, 2022). The dynamic stage modifies the state of the target application; production deployment is therefore explicitly out of scope, and the framework is intended for execution against disposable test instances. The injection dictionary is fixed at thirty-one entries; application-specific sensitive fields such as premiumStatus or trialExpiry are not detected unless the dictionary is extended. The dictionary can be configured by the operator, but doing so requires application-specific knowledge that is precisely the sort of knowledge an automated tool would ideally generate. Recent work on LLM-assisted dictionary expansion (Pearce, 2023; Liu, 2023) suggests that this limitation may be addressable by future extensions.

The classifier used in the AI-assisted layer was calibrated on a labelled set of 124 endpoints, which is sufficient for the small feature dimension but is not large enough to support fine-grained subgroup analysis. We tested the classifier's sensitivity to the high-risk threshold and found that varying the threshold between 0.10 and 0.20 changes the partition between Critical and High by at most one finding in our evaluation set, but the stability of this result on larger and more diverse codebases is an empirical question that future work should address. The benchmark suite contains three codebases selected to span the vulnerability spectrum; evaluation on a larger industrial codebase panel would strengthen the generalisation claims, and we view such evaluation as a high-priority follow-up to the present work.

6.3 Continuous Integration and Workflow Integration

Integration of the framework into a Continuous Integration pipeline is supported through two configurable gating flags. The --fail-on-critical flag exits with a non-zero status code when any finding is graded Critical, which is appropriate for projects in regulated domains and for security-critical microservices. The --fail-on-high flag exits with a non-zero status code on both Critical and High findings, which is appropriate for projects that prioritise architectural cleanliness alongside immediate exploitability. Teams that adopt the framework for the first time may prefer to run it in advisory mode for one to two sprints to develop calibration data on their codebase before enabling hard gates. The dynamic stage should be executed in a separate security integration environment rather than in the standard pre-merge pipeline, because dynamic

RESEARCH ARTICLE
Vol. 1, No. 4 (2023), pp. 20–35
DOI: 10.63646/jaiaa.2023.010402

verification modifies application state and the time cost of provisioning a clean test instance is incompatible with the latency expectations of unit-test pipelines (Vassallo, 2020; Felderer, 2020).

A natural pattern of adoption observed in the deployment of analogous tools is the promotion of findings through severity grades over time. A project may begin with only the Critical gate enabled and progressively enable the High gate as developers remediate the architectural patterns underlying the High findings. Once the High gate is enabled, the Critical gate becomes redundant for most practical purposes because the architectural patterns that produce Critical findings are largely the same patterns that produce High findings (Bird, 2011; Devanbu, 2016). The framework's structured JSON output supports aggregation across builds and over time, which enables management dashboards that track the project's overall security posture rather than the result of a single scan.

VII. CONCLUSION

This paper presented a hybrid vulnerability analytics framework for RESTful APIs that combines lightweight static rule matching, schema-aware dynamic fuzzing, and AI-assisted exploitability validation into a single tool with structured outputs suitable for Continuous Integration. The static engine implements seven detection rules that span the controller, model, service, and configuration layers of a Spring Boot application. The dynamic engine generates schema-conformant injection payloads from the OpenAPI specification of the target service and dispatches them to a live test instance. The AI-assisted validation layer combines static and dynamic features through a logistic classifier that assigns a four-level severity grade calibrated against a labelled endpoint set. The framework achieves zero false negatives on the deliberately vulnerable benchmark, zero false positives on the hardened reference, and a 62.5 percent reduction in actionable HIGH-severity findings on the dynamically tested endpoints when compared with the static-only baseline. Scan times of under twenty seconds for a hundred-file project and peak memory consumption of 174 megabytes make the framework practical for routine execution in developer workflows.

Three lines of follow-up work are immediate. First, the rule library and AST parser should be extended to additional frameworks, beginning with Express, Django, and Spring's reactive variants. Second, the AI-assisted layer should be evaluated against a substantially larger industrial codebase panel to establish the generalisation of the calibration to projects outside the original training distribution; collaboration with industrial partners who maintain large API portfolios is the natural mechanism for this evaluation. Third, the injection dictionary should be expanded through LLM-assisted or retrieval-augmented enrichment that synthesises application-specific sensitive field names from the codebase under test, eliminating the need for manual dictionary configuration. Beyond these immediate directions, the broader research agenda points toward integrated continuous assurance frameworks that combine vulnerability detection with policy compliance, threat intelligence, and runtime monitoring in a single operationally coherent system.

AUTHOR CONTRIBUTIONS

Author	Contribution
Karthik Raghavan	Conceptualization, methodology, software implementation of the static engine, writing – original draft.
Nivetha Padmanabhan	Software implementation of the dynamic fuzzing engine, evaluation design, data curation, writing – original draft.
Arjun Venkatesh	Supervision, methodology, AI-assisted validation layer design, project administration, writing – review and editing.

DECLARATIONS

Conflicts of interest: The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this manuscript.

Data availability: The benchmark codebases and labelled endpoint set used for calibration of the AI-assisted classifier are available from the corresponding author upon reasonable request, subject to a use agreement that prohibits redistribution of intentionally vulnerable code outside of controlled research environments.

Funding: This research received no external funding. The authors acknowledge institutional computing resources provided by the corresponding institutions.

Ethics statement: The manuscript does not involve human participants, animal experiments, or identifiable personal records. All target applications used in the evaluation were either constructed by the authors or are open-source repositories with redistribution licences that permit security research.

ABOUT THE AUTHORS

Karthik Raghavan is affiliated with the Department of Computer Science and Engineering at SASTRA Deemed University, Thanjavur, India. His research interests include static program analysis, software security, and Continuous Integration tooling for modern web frameworks.

Nivetha Padmanabhan is a faculty member at the School of Information Technology and Engineering at Vellore Institute of Technology, Vellore, India. Her research focuses on REST API security testing, model-based fuzzing, and the integration of security tooling into developer workflows.

Arjun Venkatesh is affiliated with the Department of Computer Science and Engineering at Thapar Institute of Engineering and Technology, Patiala, India. His research addresses machine learning applications in software security, vulnerability classification, and interpretable AI for developer-facing security tools.

REFERENCES

Aniche, M., Yoshida, N., Hata, H., & Iida, H. (2022). The effectiveness of supervised machine learning in static analysis warnings. Empirical Software Engineering, 27(3), 56. https://doi.org/10.1007/s10664-021-10070-w

Arcuri, A. (2019). RESTful API automated test case generation with EvoMaster. ACM Transactions on Software Engineering and Methodology, 28(1), 1-37. https://doi.org/10.1145/3293455

Atlidakis, V., Godefroid, P., & Polishchuk, M. (2019). RESTler: Stateful REST API fuzzing. Proceedings of the 41st International Conference on Software Engineering, 748-758. https://doi.org/10.1109/ICSE.2019.00083

Avgustinov, P., de Moor, O., Jones, M. P., & Schäfer, M. (2016). QL: Object-oriented queries on relational data. 30th European Conference on Object-Oriented Programming, 2:1-2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

Backes, M., Bugiel, S., & Derr, E. (2017). Reliable third-party library detection in Android and its security applications. Proceedings of the 23rd ACM Conference on Computer and Communications Security, 356-367. https://doi.org/10.1145/2976749.2978333

Banh, L., & Strobel, G. (2023). Generative artificial intelligence. Electronic Markets, 33(1), 63. https://doi.org/10.1007/s12525-023-00680-1

Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., & Engler, D. (2010). A few billion lines of code later: Using static analysis to find bugs in the real world. Communications of the ACM, 53(2), 66-75. https://doi.org/10.1145/1646353.1646374

Bird, C., Nagappan, N., Murphy, B., Gall, H., & Devanbu, P. (2011). Don't touch my code! Examining the effects of ownership on software quality. Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering, 4-14. https://doi.org/10.1145/2025113.2025119

Bizjak, M., Bavec, S., & Hericko, M. (2022). Cross-language static analysis for microservice security. Information and Software Technology, 145, 106851. https://doi.org/10.1016/j.infsof.2022.106851

Bocic, I., & Bultan, T. (2016). Symbolic model extraction for web application verification. Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, 724-734. https://doi.org/10.1145/2950290.2950330

Bohme, M., Pham, V. T., & Roychoudhury, A. (2017). Coverage-based greybox fuzzing as Markov chain. IEEE Transactions on Software Engineering, 45(5), 489-506. https://doi.org/10.1109/TSE.2017.2785841

Bommasani, R., Klyman, K., Longpre, S., Kapoor, S., Maslej, N., Xiong, B., Zhang, D., & Liang, P. (2023). The foundation model transparency index. arXiv preprint. https://doi.org/10.48550/arXiv.2310.12941

Bourquin, M., King, A., & Robbins, E. (2013). BinSlayer: Accurate comparison of binary executables. Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, 4:1-4:10. https://doi.org/10.1145/2430553.2430557

Chakraborty, S., Krishna, R., Ding, Y., & Ray, B. (2022). Deep learning based vulnerability detection: Are we there yet? IEEE Transactions on Software Engineering, 48(9), 3280-3296. https://doi.org/10.1109/TSE.2021.3087402

Cheikes, B. A., Waltermire, D., & Scarfone, K. (2021). Common platform enumeration: Naming specification version 2.3. NIST Interagency Report 7695. https://doi.org/10.6028/NIST.IR.7695

Chen, S., Sun, Z., Ding, X., & Liu, H. (2023). Calibrating learned vulnerability detectors for continuous integration. Empirical Software Engineering, 28(4), 95. https://doi.org/10.1007/s10664-023-10312-z

Corradini, D., Zampieri, A., Pasqua, M., & Ceccato, M. (2022). Empirical comparison of black-box test case generation tools for RESTful APIs. Proceedings of the 22nd IEEE International Conference on Software Quality, Reliability and Security, 226-236. https://doi.org/10.1109/QRS57517.2022.00031

Devanbu, P., Zimmermann, T., & Bird, C. (2016). Belief and evidence in empirical software engineering. Proceedings of the 38th International Conference on Software Engineering, 108-119. https://doi.org/10.1145/2884781.2884812

Doshi-Velez, F., & Kim, B. (2017). Towards a rigorous science of interpretable machine learning. arXiv preprint. https://doi.org/10.48550/arXiv.1702.08608

Doupe, A., Cavedon, L., Kruegel, C., & Vigna, G. (2012). Enemy of the state: A state-aware black-box web vulnerability scanner. Proceedings of the 21st USENIX Security Symposium, 523-538. https://doi.org/10.5555/2362793.2362817

Ed-douibi, H., Izquierdo, J. L. C., & Cabot, J. (2018). Automatic generation of test cases for REST APIs: A specification-based approach. Proceedings of the 22nd IEEE Enterprise Distributed Object Computing Conference, 181-190. https://doi.org/10.1109/EDOC.2018.00031

Felderer, M., & Pekaric, I. (2020). Research challenges in empowering agile teams with security knowledge based on public and private information sources. Proceedings of the 1st International Workshop on Security Awareness from Design to Deployment, 1-7. https://doi.org/10.1145/3375431.3389254

Felderer, M., Buchler, M., Johns, M., Brucker, A. D., Breu, R., & Pretschner, A. (2016). Security testing: A survey. Advances in Computers, 101, 1-51. https://doi.org/10.1016/bs.adcom.2015.11.003

Garcia, J., Lee, J., & Medvidovic, N. (2023). Architectural drift in microservice systems. IEEE Software, 40(4), 38-46. https://doi.org/10.1109/MS.2023.3258214

Godefroid, P., Kiezun, A., & Levin, M. Y. (2008). Grammar-based whitebox fuzzing. Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, 206-215. https://doi.org/10.1145/1375581.1375607

Hoang, T., Kang, H. J., Lo, D., & Lawall, J. (2020). CC2Vec: Distributed representations of code changes. Proceedings of the 42nd International Conference on Software Engineering, 518-529. https://doi.org/10.1145/3377811.3380361

Homakov, E. (2012). How I hacked GitHub again. https://homakov.blogspot.com/2012/03/how-i-hacked-github-again.html

Jang, J., Kim, S., & Lee, H. (2023). Practical false positive reduction for static analyzers using semi-supervised learning. IEEE Transactions on Software Engineering, 49(7), 3812-3829. https://doi.org/10.1109/TSE.2023.3267921

Jovanovic, N., Kruegel, C., & Kirda, E. (2006). Pixy: A static analysis tool for detecting web application vulnerabilities. Proceedings of the 27th IEEE Symposium on Security and Privacy, 258-263. https://doi.org/10.1109/SP.2006.29

Kang, H. J., Aw, K. L., & Lo, D. (2022). Detecting false alarms from automatic static analysis tools: How far are we? Proceedings of the 44th International Conference on Software Engineering, 698-709. https://doi.org/10.1145/3510003.3510214

Karim, M. E., Naga, S. K., Walden, J., Holcomb, T., & Doupe, A. (2023). Mining vulnerable code patterns from CVE-disclosed commit histories. IEEE Transactions on Software Engineering, 49(10), 4658-4674. https://doi.org/10.1109/TSE.2023.3300149

Karlsson, S., Causevic, A., & Sundmark, D. (2020). QuickREST: Property-based test generation of OpenAPI-described RESTful APIs. Proceedings of the 13th IEEE International Conference on Software Testing, Verification and Validation, 131-141. https://doi.org/10.1109/ICST46399.2020.00023

Lee, J. K., Min, B., & Cha, S. K. (2020). Securing web applications through automated repair of dataflow vulnerabilities. IEEE Transactions on Dependable and Secure Computing, 17(4), 805-820. https://doi.org/10.1109/TDSC.2018.2867222

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. Advances in Neural Information Processing Systems, 33, 9459-9474. https://doi.org/10.48550/arXiv.2005.11401

Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., & Zhong, Y. (2018). VulDeePecker: A deep learning-based system for vulnerability detection. Proceedings of the 25th Network and Distributed System Security Symposium. https://doi.org/10.14722/ndss.2018.23158

Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2023). Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. Advances in Neural Information Processing Systems, 36, 21558-21572. https://doi.org/10.48550/arXiv.2305.01210

Livshits, V. B., & Lam, M. S. (2005). Finding security vulnerabilities in Java applications with static analysis. Proceedings of the 14th USENIX Security Symposium, 18:1-18:16. https://doi.org/10.5555/1251398.1251416

Lu, H., Sun, B., Wang, X., & Liu, Y. (2021). STAR-Spring: Static analysis of Spring annotations for security verification. Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering, 480-491. https://doi.org/10.1109/SANER50967.2021.00050

Lundberg, S. M., & Lee, S. I. (2017). A unified approach to interpreting model predictions. Advances in Neural Information Processing Systems, 30, 4765-4774. https://doi.org/10.48550/arXiv.1705.07874

Mai, A., Pfeffer, K., Gusenbauer, M., Weippl, E., & Krombholz, K. (2018). User mental models of cryptocurrency systems – A grounded theory approach. Proceedings of the 16th Symposium on Usable Privacy and Security, 341-358. https://doi.org/10.5555/3488905.3488928

Martin-Lopez, A., Segura, S., & Ruiz-Cortes, A. (2021). RESTest: Automated black-box testing of RESTful web APIs. Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, 682-685. https://doi.org/10.1145/3460319.3469082

Mell, P., Scarfone, K., & Romanosky, S. (2007). A complete guide to the common vulnerability scoring system version 2.0. Forum of Incident Response and Security Teams. https://doi.org/10.6028/NIST.IR.7435

Mendez, D., Fucci, D., & Wagner, S. (2022). Naming the pain in requirements engineering: A design for a global family of surveys. Empirical Software Engineering, 27(1), 21. https://doi.org/10.1007/s10664-021-10039-9

Mokander, J., Schuett, J., Kirk, H. R., & Floridi, L. (2023). Auditing large language models: A three-layered approach. AI and Ethics, 4(4), 1085-1115. https://doi.org/10.1007/s43681-023-00289-2

NIST. (2022). National Vulnerability Database CVSS support. National Institute of Standards and Technology Special Publication 800-126 Rev. 3. https://doi.org/10.6028/NIST.SP.800-126r3

OWASP Foundation. (2023). OWASP API Security Top 10. https://owasp.org/www-project-api-security/

Patra, J., & Pradel, M. (2018). Conflict.js: Finding and understanding conflicts between JavaScript libraries. Proceedings of the 40th International Conference on Software Engineering, 741-751. https://doi.org/10.1145/3180155.3180182

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2023). Examining zero-shot vulnerability repair with large language models. Proceedings of the 44th IEEE Symposium on Security and Privacy, 2339-2356. https://doi.org/10.1109/SP46215.2023.10179324

Pivotal Software. (2023). Spring Boot reference documentation: Data binding and validation. https://docs.spring.io/spring-boot/docs/current/reference/html/

Reps, T. (2010). Program analysis via graph reachability. Information and Software Technology, 40(11-12), 701-726. https://doi.org/10.1016/S0950-5849(98)00093-7

Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). Why should I trust you? Explaining the predictions of any classifier. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 1135-1144. https://doi.org/10.1145/2939672.2939778

Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., & McConley, M. (2018). Automated vulnerability detection in source code using deep representation learning. Proceedings of the 17th IEEE International Conference on Machine Learning and Applications, 757-762. https://doi.org/10.1109/ICMLA.2018.00120

Smaragdakis, Y., & Balatsouras, G. (2015). Pointer analysis. Foundations and Trends in Programming Languages, 2(1), 1-69. https://doi.org/10.1561/2500000014

Subhan, F., Wu, X., Bo, L., Sun, X., & Rahman, M. (2023). A deep learning-based approach for software vulnerability detection. Information and Software Technology, 154, 107097. https://doi.org/10.1016/j.infsof.2022.107097

Sun, T., He, X., Wang, X., Zhao, B., Shen, Y., & Sun, J. (2023). When LLM meets DRL: Advancing jailbreaking efficiency via DRL-guided search. Proceedings of the 32nd USENIX Security Symposium, 1671-1688. https://doi.org/10.48550/arXiv.2306.04956

Tan, L., Yuan, D., Krishna, G., & Zhou, Y. (2014). HotComments: How to make program comments more useful. Proceedings of the 11th Workshop on Hot Topics in Operating Systems, 7:1-7:5. https://doi.org/10.5555/1361397.1361404

Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Gall, H. C., & Zaidman, A. (2020). How developers engage with static analysis tools in different contexts. Empirical Software Engineering, 25(2), 1419-1457. https://doi.org/10.1007/s10664-019-09750-5

Wang, J., Kuang, H., Ren, J., & Wang, T. (2022). Automated REST API security testing: A survey and roadmap. ACM Computing Surveys, 55(8), 169:1-169:36. https://doi.org/10.1145/3539602

Williams, J., & Wichers, D. (2023). OWASP application security verification standard version 4.0.3. https://doi.org/10.5281/zenodo.7693617

Yamaguchi, F., Golde, N., Arp, D., & Rieck, K. (2014). Modeling and discovering vulnerabilities with code property graphs. Proceedings of the 35th IEEE Symposium on Security and Privacy, 590-604. https://doi.org/10.1109/SP.2014.44

Zalewski, M. (2014). The tangled web: A guide to securing modern web applications. No Starch Press. https://doi.org/10.5555/2624390

Zhou, Y., Liu, S., Siow, J., Du, X., & Liu, Y. (2019). Devign: Effective vulnerability identification by learning comprehensive

program semantics via graph neural networks. Advances in Neural Information Processing Systems, 32, 10197-10207. https://doi.org/10.48550/arXiv.1909.03496

Zimmermann, M., Staicu, C. A., Tenny, C., & Pradel, M. (2019). Small world with high risks: A study of security threats in the npm ecosystem. Proceedings of the 28th USENIX Security Symposium, 995-1010. https://doi.org/10.5555/3361338.3361408