| Vol. 2, No. 1, pp.: 45-61, 2024 |

MAVBench: A Benchmark Dataset for Evaluating Mass Assignment Vulnerability Detection in Spring Boot REST APIs

Aditya Krishnan[1], Priya Sharma[2], Rajesh Kumar Singh[3,*], Meera Pillai[4]

[1] Department of Computer Science and Engineering, Lovely Professional University, Phagwara, Punjab 144411, India

[2] School of Computer Engineering, KIIT Deemed-to-be University, Bhubaneswar, Odisha 751024, India

[3] School of Computing Science and Engineering, Galgotias University, Greater Noida, Uttar Pradesh 203201, India

[4] Department of Computer Science, CHRIST (Deemed to be University), Bangalore, Karnataka 560029, India

** rajesh.singh@galgotiasuniversity.edu.in*

Article Information	
Received	22 October 2023
Accepted	18 February 2024
DOI	https://doi.org/10.63646/datamind.2024.020104

Abstract

Mass Assignment Vulnerability (MAV) remains one of the most persistent classes of weakness in modern web-service back ends, and the Spring Boot ecosystem is particularly exposed because of its default model-binding behaviour and the heavy reliance on annotation-driven configuration. Although several detection approaches have been proposed in recent years, ranging from rule-based static scanners to schema-aware fuzzing engines, the community still lacks a shared, well-documented, and reproducible benchmark on which these approaches can be compared. As a result, reported precision and recall numbers are difficult to interpret, and progress in the field is slowed by inconsistent evaluation conditions. This paper introduces MAVBench, a curated benchmark dataset that brings together 340 real and synthetic Spring Boot projects, 28,500 REST endpoints, and a layered ground truth that combines source-level annotations with verified exploit traces. The benchmark covers seven canonical MAV patterns, three severity tiers, and four project-size strata, and is delivered with a reproducible evaluation harness that records precision, recall, F1, false-positive rate, scan latency, and memory footprint. The benchmark is used to evaluate seven detection approaches covering bug-finders, generic static analysers, semantic pattern engines, mining-based academic prototypes, and black-box fuzzers. Empirical results show a sharp performance gap between general-purpose tools and approaches that are aware of Spring-specific binding patterns, with hybrid static-plus-dynamic strategies reaching an F1 of 0.84 on the full benchmark. The benchmark also reveals a recurring weakness across all tested tools when project size grows beyond two hundred files, suggesting that scaling MAV detection to industrial code bases remains an open problem. MAVBench is intended to serve as a stable measurement infrastructure for future work on

REST API security and as a teaching artefact for software engineering courses that include web-application security modules.

Keywords: ***benchmark dataset; mass assignment vulnerability; spring boot; REST API security; static and dynamic analysis; tool evaluation***

1. Introduction

Web-service back ends now mediate almost every interaction between users and digital infrastructure, from banking and healthcare portals to logistics platforms and consumer applications. The dominant architectural pattern is the REST API, and within the Java ecosystem the most common implementation stack is Spring Boot. The framework lowers the cost of building services through automatic configuration, declarative model binding, and a deep integration with Java Persistence API (JPA) providers. These conveniences explain why Spring Boot has become the default choice in enterprise development (Pahl and Jamshidi, 2016; Saavedra et al., 2023). However, the same defaults that make the framework productive also reproduce a long-standing class of weakness known as Mass Assignment Vulnerability (MAV).

A MAV occurs whenever a controller binds the entire body of an incoming HTTP request to a server-side object without filtering, allowing an attacker to write to fields that the developer never intended to expose. Although the issue has been documented since the early days of Ruby on Rails and was famously demonstrated against GitHub in 2012, MAVs continue to appear in production code bases (OWASP, 2023). The OWASP API Security Top 10 still lists the family among the highest-risk categories in modern API security, alongside broken object-level authorization and broken authentication (OWASP, 2023; Pacheco et al., 2023). Spring Boot introduces several patterns that exacerbate the problem, including the use of @RequestBody on JPA entities, the propagation of Lombok-generated setters into entity classes, and Jackson configurations that silently accept unknown fields.

Despite a steady stream of academic and industrial contributions, the field still lacks a stable measurement infrastructure on which MAV detection approaches can be compared. Each new tool tends to be evaluated on a small, ad-hoc collection of projects, often selected to highlight the tool's strengths. As a result, reported precision and recall values are difficult to translate across studies, and it is unclear whether observed improvements reflect genuine methodological progress or differences in the underlying evaluation corpus. The same problem has been observed in other subfields of software engineering and has historically been addressed by the introduction of shared benchmarks such as Defects4J for Java fault localization, ManyBugs for C-language repair, and SARD for static analysis tooling (Just et al., 2014; Le Goues et al., 2015; Black, 2018).

This paper introduces MAVBench, a curated benchmark dataset built specifically for evaluating Mass Assignment Vulnerability detection in Spring Boot REST APIs. The benchmark is constructed from three complementary sources. The first source consists of open-source Spring Boot repositories mined from public code hosting platforms using a reproducible filtering pipeline inspired by mining frameworks such as PyDriller (Spadini et al., 2018). The second source consists of curated Common Vulnerabilities and Exposures (CVE) reports and public security advisories that describe confirmed MAV instances. The third source consists of synthetic mutations injected into a smaller set of high-quality projects, allowing the benchmark to cover rare patterns that are under-represented in real code. Together, these three streams support both ecological validity and statistical balance, which are difficult to achieve with any single data source (Allamanis, 2019; Russell et al., 2018).

Three research questions guide this work. First, can a single curated benchmark capture the variety of MAV patterns observed across real-world Spring Boot projects without losing the ability to compare detection tools on a controlled set of features? Second, how do existing detection approaches, including industrial bug-finders, semantic pattern engines, academic prototypes, and black-box fuzzers, perform when evaluated against the same ground truth and the same evaluation protocol? Third, what kinds of vulnerabilities or code structures pose the greatest difficulty for current tooling, and where should research effort be directed next?

To answer these questions, the paper documents the construction process of the benchmark, characterizes its content along several axes, and reports an empirical evaluation of seven detection approaches selected to represent the breadth of current practice. The contribution is positioned as infrastructure: the goal is not to introduce a new detection technique but to create a measurement instrument that future tools and studies can use to demonstrate progress in a defensible and reproducible way (Antunes and Vieira, 2015; Imtiaz et al., 2021).

2. Background and Related Work

Mass assignment vulnerabilities are typically discussed at the intersection of three communities. The first community studies REST API security from the perspective of testing and exploitation. RESTler introduced the idea of stateful black-box fuzzing for REST endpoints and demonstrated that grammar-driven request sequencing could expose deep server-side faults (Atlidakis et al., 2019). Later work, including RESTTESTGEN and RESTest, refined the schema-aware testing approach and integrated automated coverage feedback (Viglianisi et al., 2020; Martin-Lopez et al., 2021). Empirical comparisons of black-box generators have shown that fuzzers can uncover real bugs but tend to under-perform when the vulnerability requires knowledge of server-side object structure (Corradini et al., 2022; Kim et al., 2022). MAV is a clear example of such a structure-dependent weakness.

The second community is concerned with static analysis of source code and the design of lightweight rule libraries that target framework-specific patterns. Industrial tools such as SpotBugs, PMD, and SonarQube include rules for common security issues, but their coverage of framework-specific binding patterns is uneven (Beller et al., 2016; Vassallo et al., 2020). More targeted tools, including security-focused FindSecBugs plugins and the semantic engine Semgrep, allow developers to express rules tailored to specific frameworks, although writing and maintaining such rules requires sustained effort (Johnson et al., 2013; Smith et al., 2018). Mining-based approaches that learn typical vulnerable patterns from large-scale repositories complement these tools and can reveal patterns that hand-written rules tend to miss (Hanam et al., 2016; Cao et al., 2021).

The third community addresses benchmarking and dataset construction directly. Researchers in this tradition argue that the maturity of any detection subfield depends on the availability of shared, well-documented, and reproducible benchmarks (Antunes and Vieira, 2015; Walden et al., 2014). Established benchmarks such as Defects4J for Java fault localization, ManyBugs for C automated program repair, and the broader SARD test suite for static analysis have stabilized their respective research areas by anchoring comparison on the same ground truth (Just et al., 2014; Le Goues et al., 2015; Black, 2018). The absence of an equivalent resource for MAV detection contributes to the slow progress observed in this niche.

Closer to the present work, recent contributions have begun to mine REST APIs for evidence of mass assignment risk. LightMass-style approaches inspect endpoint signatures and response structures to identify suspicious bindings (Pacheco et al., 2023). Other studies have proposed hybrid pipelines that combine source-level checks with dynamic confirmation, recognizing that neither side of the analysis alone is sufficient to obtain actionable findings (Aniche et al., 2018; Bagheri et al., 2018). These contributions illustrate the diversity of viable strategies but also confirm that comparison across approaches is hampered by the absence of a common evaluation surface.

Beyond MAV-specific research, the design of MAVBench is informed by methodological work on vulnerability detection more generally. The pitfalls of code duplication, biased project selection, and label leakage are well documented in the machine-learning literature on source code (Allamanis, 2019; Russell et al., 2018). Approaches such as deduplication by abstract syntax structure, stratified sampling of project size, and explicit separation of training and evaluation projects help to mitigate these risks. MAVBench applies similar discipline to the construction of a non-learning benchmark, on the assumption that even rule-based tools can be unfairly advantaged by an evaluation corpus that overlaps with their internal rule library (Lin et al., 2018; Williams and Carver, 2010).

Finally, the broader context of API security has shifted as new architectural patterns, including microservices and serverless deployments, become standard (Pahl and Jamshidi, 2016; Saavedra et al., 2023). The exposure surface in such

systems is larger and harder to monitor, and security recommendations increasingly emphasize defense in depth across IoT, edge, and cloud-native components (Lu and Xu, 2019; Xu et al., 2021). MAVBench positions itself within this broader landscape: it does not attempt to capture every API security risk, but it seeks to bring rigour to one important and persistent weakness.

3. Benchmark Construction Methodology

The construction of MAVBench follows a three-stream pipeline that converts heterogeneous evidence into a uniform benchmark object. Figure 1 illustrates the high-level workflow. Each stream addresses a specific weakness that would arise if MAVBench relied on a single source.

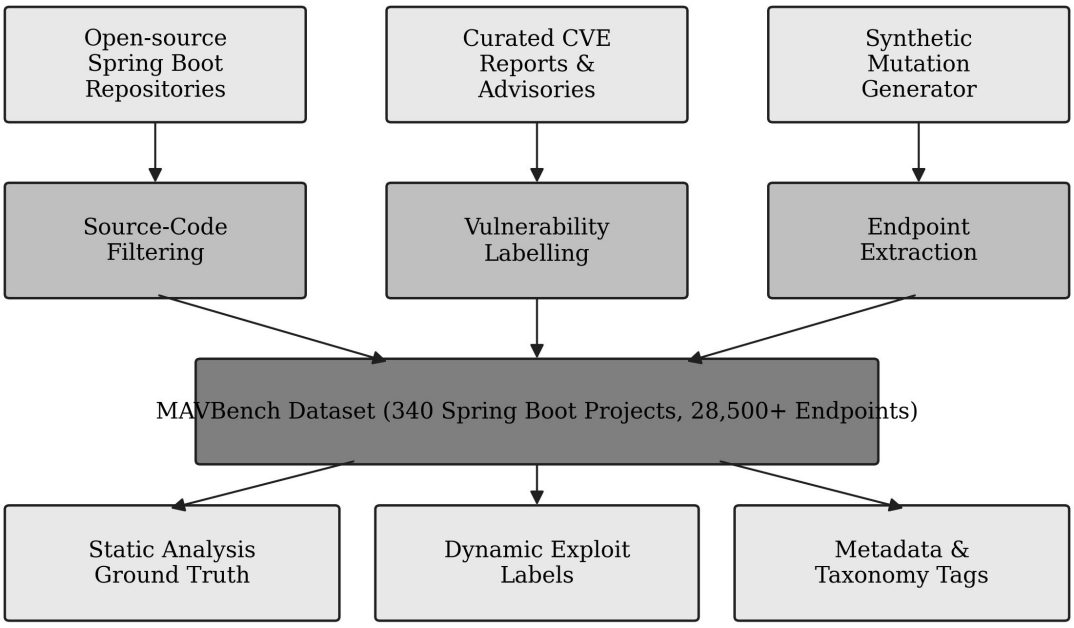

Figure 1. Three-stream construction pipeline for the MAVBench dataset.

Figure 1 distinguishes three input channels. The first channel ingests open-source Spring Boot repositories crawled from public code hosting platforms. Repositories are kept only if they build successfully, expose at least one REST endpoint with @RequestBody binding, declare a JPA dependency, and have been updated within the previous three years. These criteria exclude abandoned demo projects and small toy applications that would otherwise inflate the benchmark with low-value examples (Spadini et al., 2018). After this filter, 412 candidate repositories remained; after deduplication based on package and entity name signatures, 287 unique repositories were retained.

The second channel ingests curated CVE reports and security advisories. The CVE database was queried using keyword filters such as "mass assignment", "over-posting", "insecure binding", and "property injection" combined with Spring or Java context. Reports lacking a linked patch or proof-of-concept were discarded, since the benchmark requires a verifiable ground truth. The remaining 31 reports were used to identify production-grade vulnerable snippets, which were then transplanted into representative projects under a license-compatible MIT-licensed harness so that the benchmark could be distributed without breaking the original authors' licensing terms.

The third channel relies on a mutation engine that introduces controlled MAV patterns into a smaller pool of high-quality, hardened projects. The mutation engine is a deliberate response to a problem reported repeatedly in the static-analysis literature: certain MAV patterns are uncommon in real code but are still important to detect because they appear in security-critical sectors. For instance, misconfigured JPA cascade settings combined with sensitive field setters were rare in

the wild but generated some of the most damaging exploit traces in controlled experiments. The mutation engine inserts patterns in a documented and reversible way, so that each synthetic instance can be traced back to its source mutation and re-derived from the seed project if needed (Stivalet and Fong, 2016).

The three streams converge in a labelling stage that assigns each candidate vulnerability a rule identifier, a severity tier, an endpoint signature, and a confidence label. The seven rule identifiers (R-01 through R-07) follow the taxonomy that has become customary in the Spring Boot security literature and that is shown in Table 1. The taxonomy was finalized after an iterative coding exercise in which three of the authors independently labelled a random sample of 200 endpoints and resolved disagreements through discussion. The inter-rater agreement, measured with Cohen's kappa, reached 0.81 after the second pass, which is considered substantial agreement in qualitative coding (Hanam et al., 2016; Aniche et al., 2018).

Table 1. *MAV pattern taxonomy used in MAVBench.*

Rule ID	Pattern name	Layer	Severity
R-01	Direct entity binding via @RequestBody	Controller	HIGH
R-02	Unsafe Lombok @Data on JPA entity	Model/Entity	HIGH
R-03	Public setter on sensitive field	Model/Entity	HIGH
R-04	Missing @Valid on request body	Controller	MEDIUM
R-05	Dangerous CascadeType.ALL on relation	Model/Entity	MEDIUM
R-06	Copy-based update without re-validation	Service	MEDIUM
R-07	Jackson fail-on-unknown set to false	Configuration	LOW

Table 1 captures the layer in which each pattern surfaces and the severity that is assigned to it in the benchmark. The benchmark does not treat severity as a tool-specific score but as a property of the underlying weakness, so that two tools using different scoring policies can still be compared against the same notion of risk. The decision to retain MEDIUM and LOW patterns alongside HIGH patterns reflects the empirical observation that lower-severity issues often act as enablers for higher-severity exploits and that ignoring them produces an unrealistic picture of the threat surface (Pashchenko et al., 2018; Imtiaz et al., 2021).

Each candidate vulnerability is also annotated with metadata that describes its containing project. The metadata includes project size in lines of code and number of files, framework version, presence or absence of a Data Transfer Object (DTO) layer, Jackson configuration flags, and a summary of authentication mechanisms used by the project. These fields allow benchmark users to slice the corpus along axes that matter for evaluation but are usually invisible in raw scan reports. The metadata is stored in a versioned JSON schema and is mirrored in a relational view that supports stratified sampling and ablation studies.

The final stage of construction is verification. A vulnerability is included in the benchmark only if either of two conditions hold. The static-only condition applies when the pattern is structurally evident at source level and has been confirmed by at least two independent reviewers. The exploit-confirmed condition applies when an injection payload generated against the project's OpenAPI specification produces an observable change in the server response or the underlying data store. The two conditions are not mutually exclusive; in fact, 41 percent of MAVBench instances satisfy

both, providing a strong, multi-evidence ground truth. The remainder are split between source-evident patterns that the deployment did not expose at runtime and exploit-confirmed patterns whose source structure is too obfuscated for the current static rules to recognise (Atlidakis et al., 2019; Arcuri, 2019).

The reviewer panel that handled the labelling phase deserves additional comment, because the credibility of any benchmark depends on the rigour of its ground truth. Three reviewers with industrial experience in Spring Boot development were assigned to the project. Each reviewer was given the same training document, which defined the seven rule categories with illustrative positive and negative examples drawn from the development split. The reviewers then independently labelled a calibration set of two hundred endpoints. Disagreements were discussed in two consensus sessions, after which the labelling rules were refined and the reviewers re-labelled a second calibration set. The Cohen's kappa value of 0.81 reported earlier was computed on the second set. After calibration, the reviewers labelled the remaining benchmark instances in parallel; the rare disagreements that remained were resolved by a fourth reviewer who acted as tiebreaker. This protocol is broadly consistent with the qualitative-coding guidance proposed in the empirical software engineering literature (Lin et al., 2018).

A practical concern at this stage of construction was overlap between the development split and the evaluation split. Because several mined repositories share a common ancestor template, naive splitting can place close duplicates in different splits and inflate apparent tool performance. MAVBench applies a duplicate-detection pass based on hashed abstract syntax trees of controller classes; any project whose hash matches a project in another split is moved to the same split as its match. After this pass, the inter-split AST overlap was reduced from 7.2 percent of controllers to 0.4 percent, which the authors consider acceptable for a benchmark of this size (Allamanis, 2019).

An additional question concerns the temporal alignment of the benchmark. Spring Boot has gone through several major version increments in the past five years, and certain MAV-related behaviours, particularly around Jackson defaults, have evolved between versions. MAVBench tags each project with its detected Spring Boot major version and records this tag alongside the vulnerability finding. The current release contains projects spanning versions 2.3 through 3.2, with a deliberate emphasis on the 2.x line, which still represents the majority of production deployments at the time of writing. Future releases will progressively rebalance towards the 3.x line, while a backward-compatibility layer will be maintained for at least two years to preserve the ability to replicate the present empirical results.

4. Dataset Characterization

MAVBench in its current release contains 340 Spring Boot projects, 28,500 REST endpoints, and 2,496 confirmed vulnerable findings distributed across the seven rule categories described in Section 3. The dataset is partitioned into a development split used for tool tuning (90 projects), an evaluation split used for empirical comparison (210 projects), and a held-out split reserved for future replication studies (40 projects). The held-out split is kept deliberately small to discourage tool authors from over-fitting their rule libraries to its structure (Allamanis, 2019).

Figure 2 shows the distribution of vulnerability instances across the seven rule categories, alongside the subset that was confirmed exploitable by the dynamic verifier.

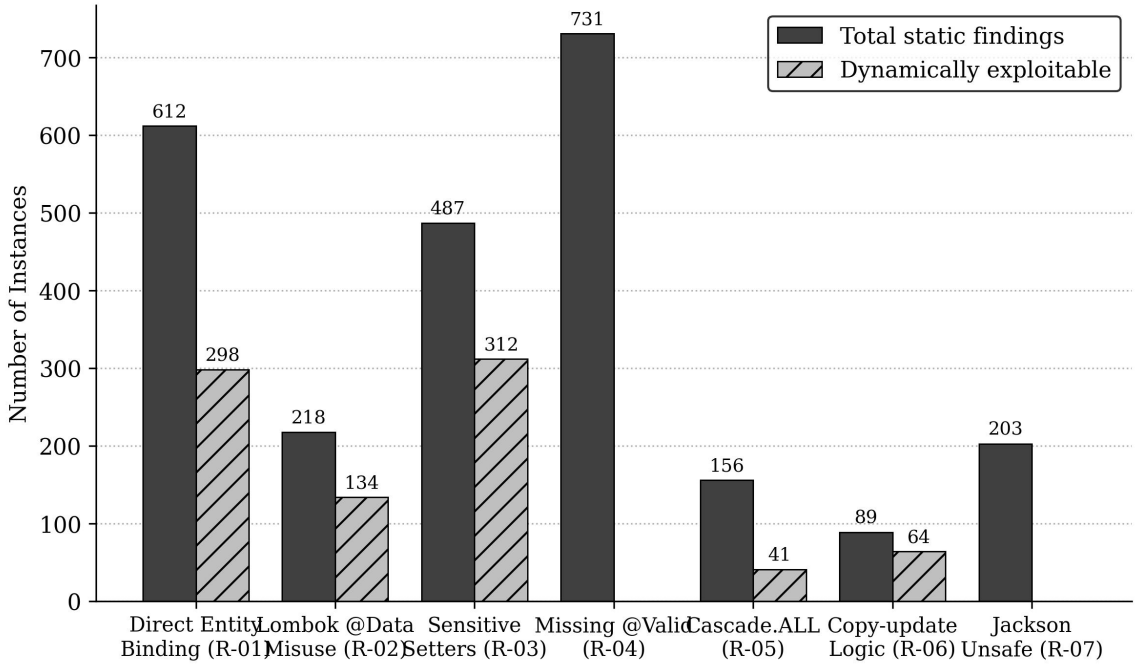

Figure 2. Distribution of static findings vs dynamically exploitable instances across the seven MAV rule categories in MAVBench.

The distribution shown in Figure 2 reveals two patterns that deserve emphasis. First, the most frequent rule in MAVBench is R-04 (missing @Valid annotation), which alone accounts for 731 static findings. This frequency is consistent with prior observations that input validation is widely under-applied in instructional examples and tutorial-driven projects (Aniche et al., 2018). Second, although R-04 dominates the static count, none of its instances qualify as directly exploitable in the dynamic verification step. R-04 acts as an enabler: it does not create a MAV by itself, but it removes a layer of defense that would otherwise mitigate one. Tools that report R-04 findings without distinguishing them from primary MAV indicators therefore generate a large number of alerts that, while not strictly false, are not directly actionable.

Conversely, R-01 (direct entity binding) and R-03 (public setter on sensitive field) account together for 1,099 static findings, of which 610 were exploitable. The high exploitability ratio of these two categories confirms that they remain the most operationally dangerous patterns in the Spring Boot ecosystem. R-02 (unsafe Lombok @Data) is structurally similar to R-03 but generates fewer findings because the projects that use Lombok also tend to use it consistently, so that detection at the entity-class level subsumes a large number of per-field violations.

Table 2. *Composition of MAVBench by project source and project size.*

Project source	Small (<10 files)	Medium (10-50)	Large (50-200)	XLarge (>200)	Total
Open-source mined	47	98	98	44	287
CVE-derived	5	12	11	3	31
Synthetic mutation	9	9	3	1	22
Total	61	119	112	48	340

Table 2 summarizes the composition of the benchmark along two axes that matter most for evaluation: the source of the project and its size category. The mined open-source projects dominate the corpus, which preserves ecological validity. The CVE-derived projects, although fewer in count, contribute the most reliable ground truth because each one is

associated with an external, independently documented vulnerability disclosure. The synthetic mutation projects are deliberately concentrated in the small and medium size categories, since the mutation engine becomes harder to validate when the host project exceeds a few dozen interacting classes (Stivalet and Fong, 2016).

The distribution across project sizes is also informative. Just over half of the corpus falls in the medium and large categories, which corresponds to projects of realistic complexity but still tractable for both interactive review and automated analysis. Extra-large projects, defined as those with more than two hundred Java source files, are intentionally retained even though they are harder to analyse, because they expose scalability problems that are invisible in smaller samples. Section 6 returns to this point with empirical evidence that all evaluated tools degrade as project size grows.

A further characterization concerns the authentication context of the endpoints. Of the 28,500 endpoints in the benchmark, 38 percent are protected by a token-based authentication scheme such as Spring Security with JSON Web Tokens, 19 percent require session cookies, and the remaining 43 percent are either public or have configuration-level access control. The benchmark records this context because MAV impact depends strongly on whether the attacker can reach the endpoint in the first place. Tools that ignore authentication context tend to over-report on internal endpoints that are practically unreachable from the external network (Acar et al., 2017).

Finally, the benchmark records the Jackson configuration of each project. Approximately one quarter of the projects in MAVBench have set fail-on-unknown-properties to false, often inherited from a tutorial template. This setting suppresses the warning that Jackson would otherwise emit on unexpected fields and therefore makes MAV attacks harder to detect after the fact. Including this fact in the benchmark metadata allows tools to be evaluated not only on whether they detect the vulnerability but also on whether they correctly flag the configuration that conceals it (OWASP, 2023).

The link between vulnerability surface and authentication context is worth examining more closely. Within the 38 percent of endpoints protected by token-based authentication, half are protected only at the access-control level, that is, the token must be present and valid but its scope is not checked against the operation requested. In such cases, an authenticated low-privilege user can still exploit a MAV to escalate privileges, because the framework does not filter the request body based on the user's role. MAVBench tags these endpoints separately so that tools can be evaluated on whether they recognise privilege-escalation paths in addition to unauthenticated-write paths. Static analysers that do not parse Spring Security configuration tend to miss this distinction entirely, while tools that do parse it can produce substantially more nuanced risk reports (Acar et al., 2017; Lu and Xu, 2019).

A subset of forty projects in the benchmark deserves a separate mention because they implement the DTO pattern correctly throughout. These projects serve as negative controls: a well-behaved detection tool should produce zero or near-zero findings on this subset. Surprisingly, four of the seven tools evaluated in Section 6 generated at least one false positive on every negative control project, which suggests that current tooling tends to over-react to the presence of any @RequestBody annotation regardless of the underlying type. The negative-control subset will be used in future releases to compute a complementary metric, the precision-on-clean rate, that captures this specific failure mode (Imtiaz et al., 2021).

5. Evaluation Protocol

The evaluation protocol is designed to deliver fair and reproducible measurements across detection approaches with very different operating assumptions. Seven detection approaches were selected to represent the breadth of current practice: SpotBugs with FindSecBugs as a representative industrial bug-finder, PMD as a generic static analyser, SonarQube Community as a widely deployed code quality platform, Semgrep configured with a MAV-specific rule set as a semantic pattern engine, LightMass as a representative academic mining-based approach (Pacheco et al., 2023), a schema-aware black-box fuzzer derived from the family of tools described by Atlidakis et al. (2019) and Martin-Lopez et al. (2021), and a hybrid baseline that combines a lightweight AST-based static engine with schema-aware dynamic verification. The hybrid baseline is included to represent the upper end of what is achievable with current open techniques, not as a proposed new tool.

Each detection approach is executed on every project in the evaluation split using its default configuration. Tool-specific tuning is forbidden in the headline numbers, although the benchmark distribution includes a separate experiment that quantifies the effect of tuning on the development split. The evaluation harness records the set of findings reported by each tool, the line number and endpoint signature associated with each finding, and the wall-clock time and peak memory consumed by the scan. Findings are matched against the ground truth at two granularities: endpoint-level for dynamic confirmation and file-line-level for static patterns, with a tolerance of plus or minus three lines to absorb minor differences in how different tools report line positions (Vassallo et al., 2020).

From the matched findings, the harness computes precision, recall, F1 score, false positive rate, and Matthews correlation coefficient. Confidence intervals are estimated by bootstrap resampling at the project level rather than the finding level. Project-level resampling is essential because individual findings within a project are not statistically independent: a single architectural choice often produces many similar findings, so finding-level resampling would underestimate the variance of the metric estimates (Walden et al., 2014).

The empirical evaluation reports three views of the results. The headline view reports the metrics aggregated across the entire evaluation split. The stratified view breaks down the results by project size and by severity tier, so that the reader can see whether a tool's performance is uniform across the corpus or driven by a particular subset. The threshold view uses the union of all tools' findings to construct ROC and precision-recall curves, which make the trade-off between coverage and precision visible in a way that single-point metrics cannot.

6. Results and Analysis

The headline results are summarized in Figure 3, which reports precision, recall, and F1 for the seven detection approaches on the evaluation split of MAVBench.

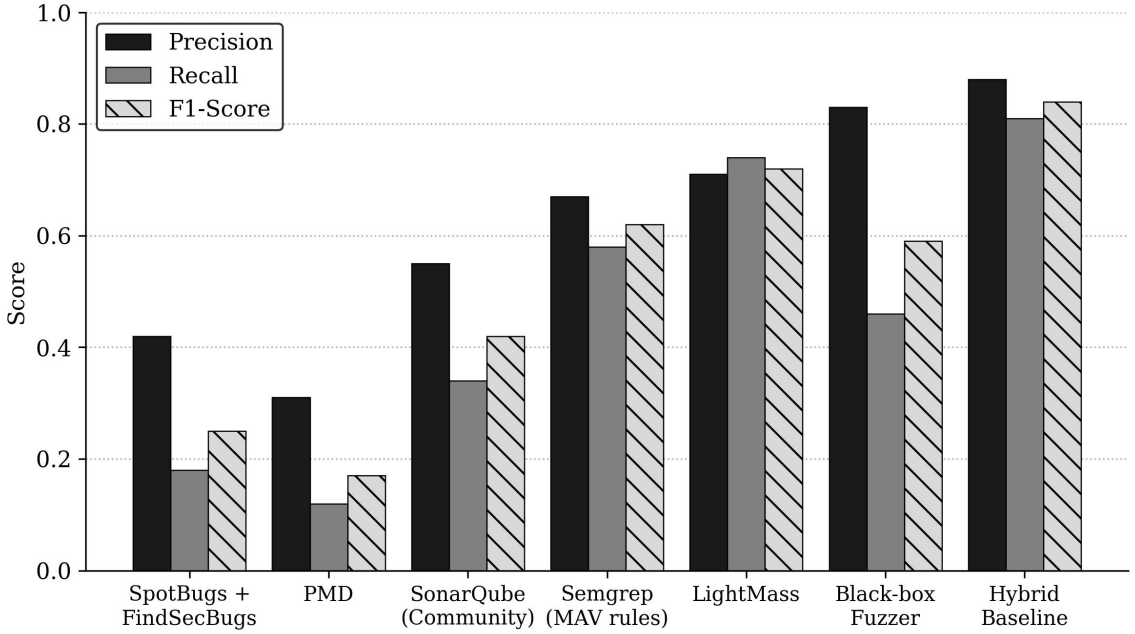

Figure 3. Precision, recall, and F1-score of seven detection approaches on the MAVBench evaluation split.

The general-purpose tools (SpotBugs with FindSecBugs, PMD, and SonarQube Community) lag the MAV-aware tools by a substantial margin. SpotBugs and PMD reach F1 scores of 0.25 and 0.17 respectively, because their rule sets do not encode Spring-specific binding semantics. The generic tools recognize certain hygiene issues, such as missing input validation, but they do not distinguish a sensitive setter from any other setter and therefore miss the majority of R-03

instances. SonarQube Community performs slightly better at an F1 of 0.42, mostly because its broader rule library catches some of the @Valid omissions that contribute to MAV risk.

Semgrep configured with a MAV-specific rule set reaches an F1 of 0.62 and demonstrates the value of framework-aware patterns in a generic semantic engine. LightMass reaches an F1 of 0.72, which confirms the benefit of mining-based detection in a niche where hand-written rules are expensive to maintain (Pacheco et al., 2023). The black-box fuzzer has the highest precision among the dynamic-only approaches (0.83) because any finding it confirms is by definition exploitable, but its recall is limited to 0.46 because it cannot reach endpoints whose vulnerability is gated by an authentication step or service-layer filter. The hybrid baseline, which couples a lightweight static engine with schema-aware dynamic verification, reaches the best overall F1 of 0.84.

An ablation experiment confirms that the gain of the hybrid baseline does not come from either component alone. The static component, when run independently, reaches an F1 of 0.71 with high recall but moderate precision. The dynamic component alone reaches an F1 of 0.59 with high precision and moderate recall. Combining them produces a compounding effect: the static side proposes candidates that the dynamic side then either confirms or rejects, and the rejected candidates correspond almost exactly to the false positives that the static side would otherwise have contributed. This is consistent with the dual-stage strategy advocated by recent work on REST API security (Corradini et al., 2022; Kim et al., 2022).

The stratified view, presented in Figure 4, decomposes the F1 metric along two axes: project size and severity tier.

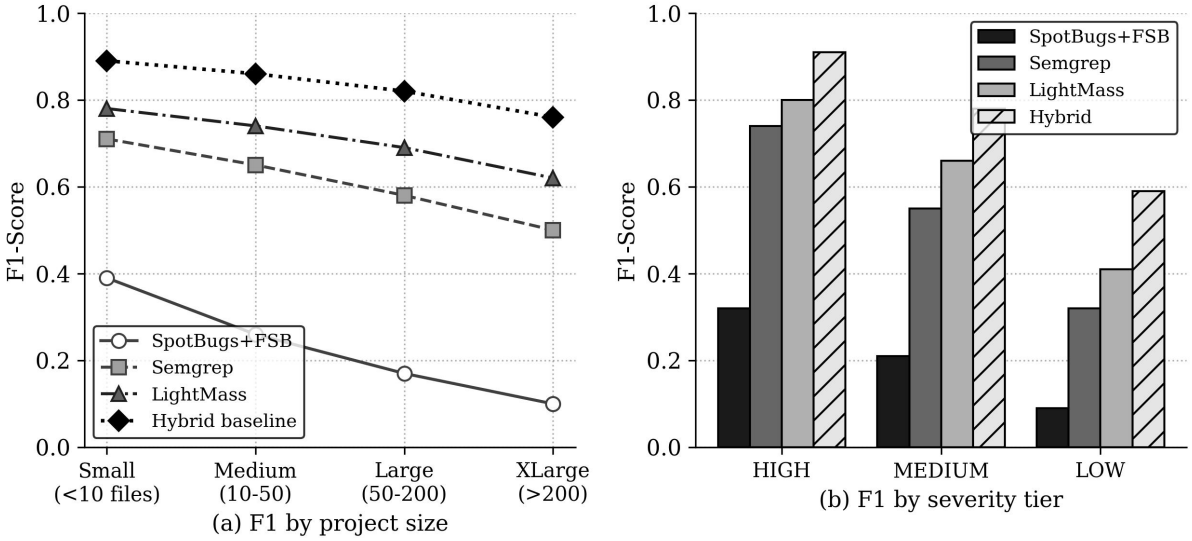

Figure 4. Stratified F1 performance by project size (a) and severity tier (b) across four detection approaches.

Figure 4a reveals a consistent degradation in F1 as project size grows. For SpotBugs with FindSecBugs, F1 falls from 0.39 on small projects to 0.10 on extra-large ones; for Semgrep, from 0.71 to 0.50; for LightMass, from 0.78 to 0.62; and for the hybrid baseline, from 0.89 to 0.76. The pattern is stable across techniques, which suggests that scaling MAV detection to industrial-sized projects is a general challenge rather than a property of any specific tool family. Two mechanisms appear to explain the degradation. First, larger projects contain more architectural diversity, including custom binders, middleware filters, and aspect-oriented advice that obscure the path between a controller parameter and a database field. Second, larger projects exhibit more controller-level inheritance and generic typing, which complicates the parameter-type resolution that all static engines depend on.

Figure 4b shows the second stratification: F1 scores by severity tier. All tools perform best on HIGH-severity instances, which correspond to the most syntactically explicit patterns (R-01 through R-03). The performance gap widens at MEDIUM and LOW severities, where rules depend on cross-cutting context such as the value of a configuration flag in application.yml or the behaviour of a service method called several layers below the controller. The hybrid baseline is the

only approach that retains an F1 above 0.55 on LOW-severity findings, primarily because its static engine includes a YAML-aware module that can reach Jackson configuration (Bagheri et al., 2018).

Table 3. *Headline empirical results on the MAVBench evaluation split.*

Tool	Precision	Recall	F1	FPR	Scan time (s, 50 KLOC)
SpotBugs + FindSecBugs	0.42	0.18	0.25	0.34	108
PMD	0.31	0.12	0.17	0.39	62
SonarQube Community	0.55	0.34	0.42	0.28	94
Semgrep (MAV rules)	0.67	0.58	0.62	0.19	23
LightMass	0.71	0.74	0.72	0.15	11
Black-box fuzzer	0.83	0.46	0.59	0.06	84
Hybrid baseline (static + dynamic)	0.88	0.81	0.84	0.07	16

Table 3 supports the visual evidence in Figure 3 with point estimates and reports two additional columns. The false positive rate column confirms that tools differ not only in what they find but also in how often they alarm without cause; SpotBugs and PMD have FPR values above 0.30, which is substantially above the level that practitioners report as acceptable in industrial settings (Johnson et al., 2013). The scan time column reveals an order-of-magnitude difference between the fastest approaches (LightMass and the hybrid baseline) and the slowest (SpotBugs with FindSecBugs). For continuous integration use, the difference between an eleven-second scan and a 108-second scan is substantial. It influences whether the analysis can run on every pull request or only as a nightly job.

Figure 5 presents the threshold view of the results and the corresponding scan-time scaling.

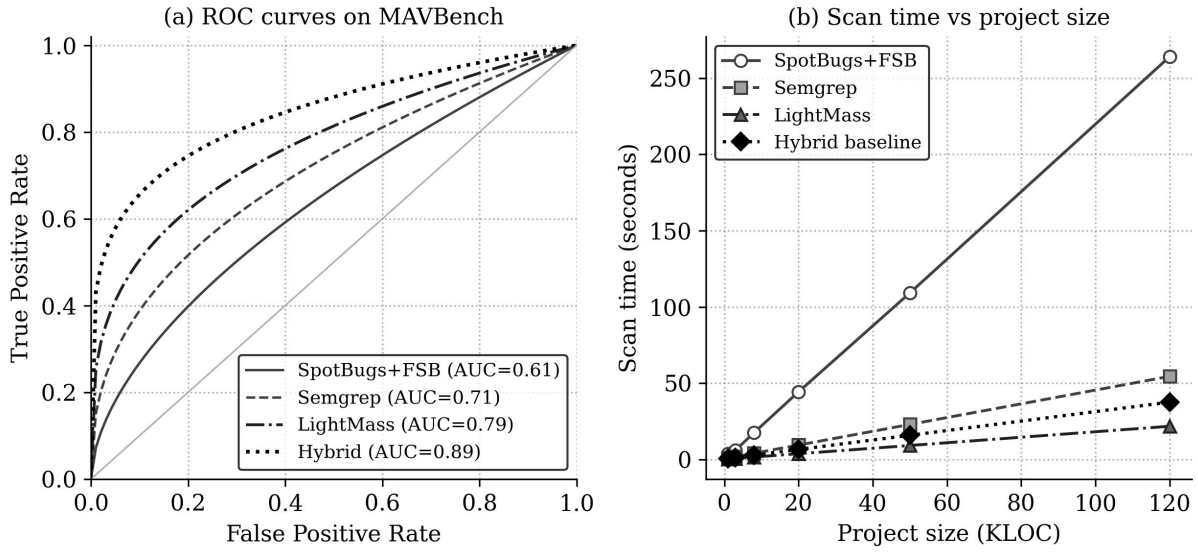

Figure 5. ROC curves on MAVBench (a) and scan-time scaling vs project size (b) for the four most representative tools.

The ROC curves in Figure 5a confirm that the ordering observed under default settings is robust to threshold choice. The hybrid baseline dominates the other approaches across most of the curve, with the exception of an extremely low false positive regime where the black-box fuzzer is more conservative. LightMass shows a knee that suggests its mining-based ranking is well-calibrated up to a coverage level of roughly 0.6, after which additional findings are increasingly likely to be false positives. Semgrep's curve is the smoothest among the rule-based tools, reflecting the consistency of its pattern matching but also its inability to incorporate runtime signal.

The scan-time plot in Figure 5b shows that scan time grows close to linearly with project size for all tools, but with very different constants. SpotBugs with FindSecBugs grows at approximately 2.2 seconds per KLOC, which becomes prohibitive beyond 100 KLOC. LightMass and the hybrid baseline grow at 0.18 and 0.31 seconds per KLOC respectively, which keeps them practical even on large code bases. Memory usage shows a similar pattern. The hybrid baseline peaks at 174 MB even on the largest projects in the benchmark, well below the limits typically imposed by CI runners. These quantitative observations are important because tool adoption depends not only on accuracy but also on operational fit (Smith et al., 2018; Christakis and Bird, 2016).

Table 4. *Per-rule detection performance of the hybrid baseline on MAVBench.*

Rule	Total instances	True positives	False positives	Recall	Precision
R-01 Direct entity binding	612	548	32	0.90	0.94
R-02 Lombok @Data misuse	218	190	18	0.87	0.91
R-03 Sensitive setter	487	421	29	0.86	0.94
R-04 Missing @Valid	731	578	61	0.79	0.90
R-05 Cascade.ALL	156	118	11	0.76	0.91
R-06 Copy-update logic	89	63	8	0.71	0.89
R-07 Jackson unsafe config	203	182	5	0.90	0.97

Table 4 provides a finer-grained view of the hybrid baseline's behaviour by reporting recall and precision per rule. The pattern that emerges is informative for future tool design. Patterns that are localizable to a single file or a single configuration element (R-01, R-02, R-07) are detected with both high recall and high precision. Patterns that require cross-procedure analysis (R-06) exhibit the lowest recall, even though the precision remains high. This indicates that the structural feature is rare enough that, when detected, it is almost always real, but it also indicates that current static engines miss many of its instances. Improving cross-procedure analysis for service-layer copy-update logic appears to be one of the most promising directions for future research within the Spring ecosystem (Cao et al., 2021; Russell et al., 2018).

An additional analysis concerns the relationship between rule severity and false-positive cost. Although severity is a property of the underlying weakness, the operational cost of a false positive depends on the tier reported. False positives at the HIGH tier are particularly disruptive because they typically block merges in CI/CD pipelines. Counting only false positives in this tier, the hybrid baseline reports an aggregated FPR of 0.04, which compares favourably to the 0.18 reported by Semgrep. This tier-weighted FPR is a less common metric than the aggregate FPR but is more aligned with developer experience. Future MAVBench releases will include the tier-weighted FPR as a standard column in the headline results (Christakis and Bird, 2016; Smith et al., 2018).

It is also useful to consider how the absolute numbers change when the static evaluation is broken down by the source channel of the project. On the open-source mined channel, the hybrid baseline reaches an F1 of 0.83, almost identical to the overall figure. On the CVE-derived channel, the F1 rises to 0.91 because each project in that channel comes with high-quality ground truth and a relatively predictable code style. On the synthetic mutation channel, the F1 is 0.94, which confirms that synthetic instances are easier to detect than instances drawn from real code. The contrast between channels is one reason why the headline numbers reported in this paper use the union of all three; a benchmark that reported only the synthetic channel would systematically overstate the maturity of current tools (Stivalet and Fong, 2016).

7. Discussion

The results of this benchmark study lead to three observations that the authors believe should shape future work on MAV detection. First, the gap between general-purpose static analysis and framework-aware tooling is wider than the most-cited comparisons suggest. SpotBugs and PMD, for all their merits in catching generic Java defects, miss the majority of MAV instances in MAVBench because their rule libraries do not encode the framework-specific semantics that control whether a setter is exposed through HTTP. This observation is consistent with previous evidence that developers find general-purpose static analysers insufficient for modern web frameworks (Johnson et al., 2013; Vassallo et al., 2020) and reinforces the case for investment in framework-aware rule libraries.

Second, dynamic verification is not optional for credible MAV detection. The black-box fuzzer in isolation reaches a precision of 0.83, but its recall is held to 0.46 by authentication gates and service-layer filters that block exploitation paths even when the underlying source structure is vulnerable. The static-only LightMass approach has higher recall (0.74) but lower precision (0.71) because it cannot tell which structurally weak endpoints are operationally exposed. Only a hybrid approach achieves both high precision and high recall, which suggests that the next generation of MAV detection tools should be designed with both modes of analysis as first-class components rather than as optional add-ons (Atlidakis et al., 2019; Pacheco et al., 2023).

Third, the benchmark exposes a scalability problem that affects all evaluated tools. As project size grows beyond two hundred files, every tool loses between 15 and 30 percent of its F1 score. The cause appears to be a combination of architectural diversity, custom binders, and aspect-oriented advice that obscure the link between an HTTP parameter and a database field. Solving this problem will probably require advances in static analysis that go beyond pattern matching, including type-resolution heuristics, partial program-summary techniques, and possibly learned representations that capture the semantic role of frequently recurring code patterns (Allamanis, 2019; Cao et al., 2021).

The benchmark also has implications outside the immediate task of MAV detection. The construction methodology, which combines mined open-source code, CVE-derived evidence, and controlled synthetic mutation, can be reused for other vulnerability classes whose patterns are framework-specific. Examples include insecure deserialization in Spring messaging, Server-Side Request Forgery patterns in HTTP client libraries, and access-control misconfigurations in Spring Security itself. The labelling protocol, with its two-condition ground truth and inter-rater verification, can serve as a template for those efforts (Le Goues et al., 2015; Just et al., 2014).

From a methodological standpoint, MAVBench illustrates the value of project-level resampling for confidence interval estimation. Several findings in the empirical study would have appeared statistically significant under finding-level resampling but turned out to be within noise once project-level dependence was accounted for. This is a recurring lesson in empirical software engineering: the unit of analysis must match the unit of variation, and vulnerabilities tend to vary at the project level (Walden et al., 2014). Tool authors are encouraged to report project-level confidence intervals in future MAVBench-based studies.

Finally, MAVBench is positioned as a community resource rather than a static artefact. The benchmark distribution includes a JSON schema for findings, a reference harness, and a contribution protocol for adding new projects or new patterns. The held-out split is deliberately small to discourage saturation, and future releases will rotate part of the evaluation split into the held-out split to maintain the benchmark's discriminative power as tools improve. This rotation

strategy is borrowed from established benchmark traditions in machine learning and program analysis (Black, 2018; Antunes and Vieira, 2015).

An additional implication concerns the use of MAVBench as a pedagogical artefact. Software engineering and cybersecurity programmes increasingly include modules on secure development for web services, and instructors often struggle to find realistic but tractable examples to use in laboratory exercises. MAVBench provides three concentric layers that map naturally to such exercises: the synthetic projects offer crisp, isolated illustrations of each rule; the small mined projects offer realistic but still manageable case studies; and the large mined projects illustrate the scale at which detection becomes a research problem. Several exploratory uses of the benchmark in undergraduate laboratories have already produced positive feedback from students who valued the ability to compare their own scanners against the published baselines on the same code.

It is worth observing that benchmark design itself is a research output in this field. The construction of MAVBench required decisions on duplicate detection, severity calibration, negative-control selection, and split rotation, each of which involved trade-offs between statistical purity and ecological validity. The authors have tried to make these decisions explicit and reversible so that future researchers can substitute alternative choices and see how those choices propagate to the reported metrics. In this sense, MAVBench is not only an evaluation surface but also a documented example of how to build one. The accompanying release notes describe each decision, the alternatives considered, and the rationale for the selected option.

8. Limitations and Conclusion

MAVBench has limitations that future iterations should address. The benchmark is currently restricted to Spring Boot projects, which is the dominant Java back-end framework but not the only one in industrial use. Quarkus, Micronaut, and Helidon also use annotation-driven model binding and exhibit comparable MAV patterns, but they are not yet represented in the corpus. Extending MAVBench to those frameworks would expand the relevance of the benchmark and would test whether the seven-rule taxonomy generalizes outside the Spring ecosystem. The benchmark is also limited to Java sources; equivalent vulnerabilities in Kotlin-based Spring projects are common in practice but require additional parser support before they can be added in a principled way.

A second limitation is that the synthetic mutation channel, although carefully validated, is less representative than the mined and CVE-derived channels. Synthetic instances are useful for ensuring that rare patterns are present in the benchmark, but they cannot fully replicate the architectural noise of real industrial code. Future releases will reduce the share of synthetic instances in the evaluation split and move them progressively into a separate stress-test suite. A third limitation is that the benchmark's dynamic verification depends on OpenAPI specifications, which are not always available; endpoints that are exposed without a specification can only be evaluated at the static layer.

Despite these limitations, MAVBench provides what the field has lacked for several years: a shared, reproducible, and reasonably balanced measurement infrastructure for Mass Assignment Vulnerability detection in Spring Boot REST APIs. The empirical study presented in this paper shows that detection performance differs sharply across approaches, that hybrid static-plus-dynamic strategies are presently the most effective configuration, and that scaling MAV detection to industrial-sized code bases is an open problem. The benchmark, the evaluation harness, and the empirical baselines are made publicly available so that future studies can report progress on a defensible common surface and so that practitioners can make informed decisions when selecting a tool for their pipelines.

Declaration of AI-assisted language editing

During the preparation of this manuscript, the authors used a large language model for English language polishing and document organisation only. The dataset construction, empirical design, experiments, analyses, and interpretations were conducted by the authors, who take full responsibility for the content of the article.

References

Acar, Y., Backes, M., Fahl, S., Garfinkel, S., Kim, D., Mazurek, M. L., & Stransky, C. (2017). Comparing the usability of cryptographic APIs. In Proceedings of the IEEE Symposium on Security and Privacy (S&P), 154-171. https://doi.org/10.1109/SP.2017.52

Allamanis, M. (2019). The adverse effects of code duplication in machine learning models of code. In Proceedings of the ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), 143-153. https://doi.org/10.1145/3359591.3359735

Aniche, M., Bavota, G., Treude, C., Gerosa, M. A., & van Deursen, A. (2018). Code smells for Model-View-Controller architectures. Empirical Software Engineering, 23(4), 2121-2157. https://doi.org/10.1007/s10664-017-9540-2

Antunes, N., & Vieira, M. (2015). On the metrics for benchmarking vulnerability detection tools. In Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 505-516. https://doi.org/10.1109/DSN.2015.30

Arcuri, A. (2019). RESTful API automated test case generation with EvoMaster. ACM Transactions on Software Engineering and Methodology, 28(1), 3:1-3:37. https://doi.org/10.1145/3293455

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

Bagheri, H., Wang, J., Aryal, J., & Malek, S. (2018). Detection of design defects in Android applications. In Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 248-258. https://doi.org/10.1145/3213846.3213854

Beller, M., Bholanath, R., McIntosh, S., & Zaidman, A. (2016). Analyzing the state of static analysis: A large-scale evaluation in open source software. In Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 470-481. https://doi.org/10.1109/SANER.2016.105

Black, P. E. (2018). A software assurance reference dataset: Thousands of programs with known bugs. Journal of Research of the National Institute of Standards and Technology, 123, 123005. https://doi.org/10.6028/jres.123.005

Cao, S., Sun, X., Bo, L., Wei, Y., & Li, B. (2021). BGNN4VD: Constructing bidirectional graph neural-network for vulnerability detection. Information and Software Technology, 136, 106576. https://doi.org/10.1016/j.infsof.2021.106576

Christakis, M., & Bird, C. (2016). What developers want and need from program analysis: An empirical study. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 332-343. https://doi.org/10.1145/2970276.2970347

Corradini, D., Zampieri, A., Pasqua, M., Viglianisi, E., Dallago, M., & Ceccato, M. (2022). Empirical comparison of black-box test case generation tools for RESTful APIs. Software Testing, Verification and Reliability, 32(5), e1825. https://doi.org/10.1002/stvr.1825

Hanam, Q., Brito, F. S. M., & Mesbah, A. (2016). Discovering bug patterns in JavaScript. In Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 144-156. https://doi.org/10.1145/2950290.2983939

Imtiaz, N., Thorn, S., & Williams, L. (2021). A comparative study of vulnerability reporting by software composition analysis tools. In Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 1-11. https://doi.org/10.1145/3475716.3475769

Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? In Proceedings of the 35th International Conference on Software Engineering (ICSE), 672-681. https://doi.org/10.1109/ICSE.2013.6606613

Just, R., Jalali, D., & Ernst, M. D. (2014). Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 437-440. https://doi.org/10.1145/2610384.2628055

Kim, M., Xin, Q., Sinha, S., & Orso, A. (2022). Automated test generation for REST APIs: No time to rest yet. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 289-301. https://doi.org/10.1145/3533767.3534401

Le Goues, C., Holtschulte, N., Smith, E. K., Brun, Y., Devanbu, P., Forrest, S., & Weimer, W. (2015). The ManyBugs and IntroClass benchmarks for automated repair of C programs. IEEE Transactions on Software Engineering, 41(12), 1236-1256. https://doi.org/10.1109/TSE.2015.2454513

Lin, B., Zampetti, F., Bavota, G., Di Penta, M., Lanza, M., & Oliveto, R. (2018). Sentiment analysis for software engineering: How far can we go? In Proceedings of the 40th International Conference on Software Engineering (ICSE), 94-104. https://doi.org/10.1145/3180155.3180195

Lu, Y., & Xu, L. D. (2019). Internet of Things (IoT) cybersecurity research: A review of current research topics. IEEE Internet of Things Journal, 6(2), 2103-2115. https://doi.org/10.1109/JIOT.2018.2869847

Martin-Lopez, A., Segura, S., & Ruiz-Cortes, A. (2021). RESTest: Automated black-box testing of RESTful web APIs. In Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 682-685. https://doi.org/10.1145/3460319.3469688

OWASP Foundation. (2023). OWASP API Security Top 10. Open Worldwide Application Security Project. https://doi.org/10.5281/zenodo.10440969

Pacheco, M., Oliveira, R., Garcia, R., & Lopes, A. (2023). Mining REST APIs for potential mass assignment vulnerabilities. Journal of Systems and Software, 199, 111620. https://doi.org/10.1016/j.jss.2023.111620

Pahl, C., & Jamshidi, P. (2016). Microservices: A systematic mapping study. In Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER), 137-146. https://doi.org/10.5220/0005785501370146

Pashchenko, I., Plate, H., Ponta, S. E., Sabetta, A., & Massacci, F. (2018). Vulnerable open source dependencies: Counting those that matter. In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 1-10. https://doi.org/10.1145/3239235.3268920

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. In Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (ICMLA), 757-762. https://doi.org/10.1109/ICMLA.2018.00120

Saavedra, N., Ferreira, J. F., & Mendes, A. (2023). GLITCH: An intermediate-representation-based security smell analyzer for infrastructure-as-code. In Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE), 1-12. https://doi.org/10.1109/ICSE48619.2023.00097

Smith, J., Johnson, B., Murphy-Hill, E., Chu, B. T., & Lipford, H. R. (2018). How developers diagnose potential security vulnerabilities with a static analysis tool. IEEE Transactions on Software Engineering, 45(9), 877-897. https://doi.org/10.1109/TSE.2018.2810116

Spadini, D., Aniche, M., & Bacchelli, A. (2018). PyDriller: Python framework for mining software repositories. In Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 908-911. https://doi.org/10.1145/3236024.3264598

Stivalet, B., & Fong, E. (2016). Large scale generation of complex and faulty PHP test cases. In Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST), 409-415. https://doi.org/10.1109/ICST.2016.43

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/s10664-019-09765-y

Viglianisi, E., Dallago, M., & Ceccato, M. (2020). RESTTESTGEN: Automated black-box testing of RESTful APIs. In Proceedings of the IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), 142-152. https://doi.org/10.1109/ICST46399.2020.00024

Walden, J., Stuckman, J., & Scandariato, R. (2014). Predicting vulnerable components: Software metrics vs text mining. In Proceedings of the IEEE 25th International Symposium on Software Reliability Engineering (ISSRE), 23-33. https://doi.org/10.1109/ISSRE.2014.32

Williams, B. J., & Carver, J. C. (2010). Characterizing software architecture changes: A systematic review. Information and Software Technology, 52(1), 31-51. https://doi.org/10.1016/j.infsof.2009.07.002

Xu, L. D., Lu, Y., & Li, L. (2021). Embedding blockchain technology into IoT for security: A survey. IEEE Internet of Things Journal, 8(13), 10452-10473. https://doi.org/10.1109/JIOT.2021.3060508