Toward Autonomous DevSecOps Agents: Static–Dynamic API Security Testing for Future Software Supply Chains

Anil Kumar Sharma[1], Meera Krishnan[2], Vikram Reddy[3], Deepak Mehta[4, *]

[1]Department of Computer Science and Engineering, SASTRA Deemed University, Thanjavur, Tamil Nadu, India, 613401
[2]School of Computer Science and Engineering, VIT-AP University, Amaravati, Andhra Pradesh, India, 522237
[3]Department of Information Technology, Koneru Lakshmaiah Education Foundation (KL Deemed University), Vaddeswaram, Andhra Pradesh, India, 522502
[4]School of Computer Applications, Lovely Professional University, Phagwara, Punjab, India, 14441
* Email: deepak.mehta@lpu.co.in (Corresponding Author)

Abstract

The increasing reliance of enterprise software supply chains on Representational State Transfer (REST) Application Programming Interfaces (APIs) has elevated API security from a niche concern to a foundational property of modern software production. Among the API-level weaknesses catalogued by the Open Worldwide Application Security Project, mass assignment and related broken object property level authorization faults remain disproportionately common in spite of more than a decade of warnings. They persist because the offending code is syntactically indistinguishable from safe code, because automatic binding is a default behaviour of the most widely deployed web frameworks, and because traditional security tooling treats source code analysis and runtime probing as separate concerns. This paper proposes a conceptual and operational direction for the next generation of security tooling: autonomous DevSecOps agents that combine static and dynamic API testing into a single, supply-chain-aware workflow. We outline the structural reasons API binding faults are missed by single-perspective analyzers, present a reference architecture for hybrid agents that builds a lightweight abstract syntax tree, evaluates it against a rule library, and confirms exploitability through schema-aware fuzzing of OpenAPI-described endpoints. We report empirical evaluation on three Java Spring Boot codebases representing deliberately vulnerable, tutorial-driven, and production-hardened conditions. The static stage recovered every planted vulnerability with no false negatives; the dynamic stage confirmed three of eight candidate endpoints as truly exploitable and filtered the remaining five, cutting the actionable high-severity backlog by 62.5 percent. We close with a research agenda situating hybrid agents within evolving paradigms of trustworthy DevSecOps, large-language-model assisted vulnerability reasoning, and software bill of materials governance.

Keywords: devsecops; mass assignment vulnerability; REST API security; static analysis; dynamic fuzzing; software supply chain; OpenAPI; CI/CD security gating

Article History:
Received: July 12, 2025
Revised: September 21, 2025
Accepted: November 18, 2025
Available Online: December 30, 2025

ISSN: © 2025 INATGI (Institute of Advanced Technology and Green Innovation). Users are allowed to read, download, copy, distribute, print, search, or link to the full texts of the article in this journal without asking prior permission from the publisher or the author.
See: https://inatgi.in/index.php/rft/index for more information.

I. INTRODUCTION

Software supply chains have become objects of strategic concern. The SolarWinds Orion compromise of 2020 demonstrated that an adversary who can quietly insert malicious behaviour at any link of the build, distribution, or deployment pipeline can compromise tens of thousands of downstream organizations through a single vendor (Martínez & Durán, 2021; Hammi & Zeadally, 2023). The disclosures that followed forced practitioners and policy-makers to reframe security as an end-to-end property of how software is produced and consumed, rather than a property of the final binary alone (Okafor et al., 2022; Xia et al., 2024). Within this broader picture, Representational State Transfer (REST) Application Programming Interfaces (APIs) occupy a uniquely sensitive position. They are simultaneously the integration substrate that lets developers compose modern applications from third-party services and the dominant attack surface through which adversaries can read, modify, or exfiltrate data without ever touching the underlying infrastructure (Lu, 2017a; Lu & Xu, 2019; Chen et al., 2024).

API security has therefore migrated from the periphery of software engineering practice to its centre. Industry telemetry indicates that nearly all organizations now report at least one API-related security incident annually (Atlidakis et al., 2020; Wu & Feng, 2025). The Open Worldwide Application Security Project periodically publishes a ranked catalogue of API risks, and the 2023 revision singled out broken object level authorization and broken object property level authorization as the most prevalent and most damaging classes (OWASP Foundation, 2023). Mass assignment, historically tracked as a separate category, was absorbed into broken object property level authorization in the same revision, reflecting a maturing understanding that property-level write defects and read defects share a common root cause: the absence of fine-grained authorization checks at the boundary between client-supplied data and server-side state.

Detection of these defects has proved persistently difficult. Static analysis is fast, deterministic, and amenable to integration with continuous-integration pipelines, but it reports many candidates that are subsequently neutralized by runtime defenses (Lipp et al., 2022; Croft et al., 2023). Dynamic analysis exposes only those defects that an external probe can actually trigger, but is blind to internal architectural risks such as exposed setters on a model class, unsafe Lombok annotations on a Java Persistence API entity, or misconfigured Jackson deserialization (Corradini et al., 2023; Karlsson et al., 2020; Mazidi et al., 2024; Zhou et al., 2018). Neither approach is satisfactory on its own. Empirical work in the DevSecOps literature has begun to argue that the path forward lies in tools that integrate both perspectives within a single workflow that can be gated by a continuous-integration system without requiring developer expertise in security (Jakku, 2025; Bedoya et al., 2024; Rahman et al., 2023).

This paper contributes to that direction. We argue that the next generation of API security tooling should be conceptualized not as a set of independent scanners but as autonomous DevSecOps agents: software components that observe the source code, the build artifacts, the running service, and the supply chain context together, and that emit actionable evidence in a form that can be acted on by both human developers and automated gating policies. The intellectual ancestry of this idea draws on threads from cyber-physical-system security (Lu, 2017b), Internet-of-Things cybersecurity (Lu & Xu, 2019; Xu et al., 2021), industrial information integration (Lu, 2017a; Lu, 2025), and the emerging body of work on large language models for vulnerability reasoning (Khare et al., 2024; Zhou et al., 2024; Steenhoek et al., 2024).

The remainder of the paper is structured as follows. Section II positions API binding defects in the broader landscape of software supply chain security and reviews the limits of single-perspective testing. Section III articulates a conceptual model of an autonomous DevSecOps agent and its constituent reasoning steps. Section IV details a reference architecture that combines a lightweight abstract-syntax-tree-based static engine, a schema-aware dynamic verifier, and an orchestrator that mediates between them. Section V reports an empirical evaluation on three Java Spring Boot codebases. Section VI discusses what the results imply for the design of trustworthy agents, addresses limitations of the current prototype, and outlines a forward-looking research agenda. Section VII concludes.

II. BACKGROUND AND RELATED WORK

A. The Software Supply Chain Threat Model

The notion of a software supply chain has shifted from describing how source code becomes a deployable artifact to a richer construct that captures every dependency, every continuous-integration script, every container image, and every running service that contributes to the behaviour observable by an end user (Okafor et al., 2022; Ohm et al., 2020; Wu et al., 2019). Studies on software bill of materials adoption (Xia et al., 2024) and on package-registry compromises (Ladisa et al., 2023)

ISSN: © 2025 INATGI (Institute of Advanced Technology and Green Innovation). Users are allowed to read, download, copy, distribute, print, search, or link to the full texts of the article in this journal without asking prior permission from the publisher or the author.
See: https://inatgi.in/index.php/rft/index for more information.

emphasize that compromise can occur at the source repository, during build, in dependency resolution, in container construction, or at deployment. Each transition multiplies attack surface. Researchers have begun to explore blockchain-based audit trails and tamper-evident provenance records as a substrate for governing this expanded surface (Ahsan et al., 2021; Isaja et al., 2023; Tao et al., 2023; Zheng & Lu, 2022).

Within this layered view, the deployed API is the entry point through which an external adversary directly interacts with the system. Even when the supply chain itself is uncompromised, an unsafe binding pattern in a controller can expose the rest of the chain to data corruption and privilege escalation. API binding defects are thus simultaneously a symptom of a supply chain that did not catch them and a vector that subsequent attackers can exploit (Hammi & Zeadally, 2023). It is this dual role that motivates positioning API security testing as a component of supply chain governance rather than a separate concern.

B. Mass Assignment and Broken Object Property Level Authorization

Mass assignment, as a defect class, arises when a web framework automatically maps user-supplied request fields to internal data-model objects without enforcing field-level access controls (OWASP Foundation, 2023; Corradini et al., 2023; Zhou et al., 2018). The canonical historical example is the 2012 disclosure in which a researcher altered the GitHub authorization model by submitting a parameter the application was not intended to accept; the framework dutifully bound it. Twelve years later, the equivalent defect is still being found in production code (Wu & Feng, 2025; Mazidi et al., 2024). The 2023 OWASP revision merged mass assignment with excessive data exposure under the broker label of broken object property level authorization, reflecting that both flaws originate in the absence of property-by-property authorization between an API contract and an underlying entity.

Three structural conditions are jointly necessary for mass assignment to be exploitable: the server binds user input to an object without field-level filtering; the target object contains fields that influence security-relevant state; and the database or business layer persists those fields without revalidating authority. In Java Spring Boot applications the conditions are easy to satisfy by accident. The @RequestBody annotation, the Jackson ObjectMapper, the @Entity class hierarchy of the Java Persistence API, and the Lombok @Data annotation each individually represent a reasonable convenience; their composition, however, yields a default-permissive binding surface (Pivotal, 2023; Bedoya et al., 2024). Configuration toggles such as fail-on-unknown-properties further suppress the only diagnostic an operator might have noticed.

C. Static Analysis for API Security

Static application security testing (SAST) tools analyze source code without executing it (Lipp et al., 2022; Croft et al., 2023). Their main appeal is determinism: a given commit yields a given result, which is reproducible and amenable to gating. Their main weakness is over-approximation: they cannot tell whether the conditions necessary for an exploit are actually present at runtime, so they over-report. Empirical studies place false-positive rates of mature SAST tools for C and Java in the range of 60 to 80 percent for many vulnerability classes (Lipp et al., 2022). For API binding defects in particular, a controller method that binds an entity directly may be perfectly safe because a downstream service layer filters the offending fields; static analysis has no way to know this. Recent research on large-language-model assisted SAST reduces the gap somewhat by reasoning about programmer intent (Khare et al., 2024; Steenhoek et al., 2024), but the fundamental limit remains.

D. Dynamic Analysis and OpenAPI-Driven Fuzzing

Dynamic application security testing (DAST), and in particular API fuzzing, complements SAST by probing a live service (Atlidakis et al., 2020; Karlsson et al., 2020). The OpenAPI Specification has become the standard interchange for describing REST endpoints and their request schemas, and several research and industrial tools generate test cases from these descriptions. RESTler (Atlidakis et al., 2020) and intelligent payload fuzzers (Godefroid et al., 2020) infer dependencies between operations and craft sequences likely to expose flaws. Corradini et al. (2023) extended this line of work with the first published black-box approach specifically targeting mass assignment, using clustering over operation schemas to discover read-only fields that may be writable. Karlsson et al. (2020) explored property-based testing of OpenAPI-described services. Each of these tools confirms exploitability with high precision when the appropriate request is sent, but each is fundamentally blind to defects that require source-code visibility: an exposed setter on a JPA entity, a Lombok-generated accessor that does not appear in the source, or a deserialization configuration buried in a YAML file.

E. Hybrid and Agentic Approaches in DevSecOps

ISSN: © 2025 INATGI (Institute of Advanced Technology and Green Innovation). Users are allowed to read, download, copy, distribute, print, search, or link to the full texts of the article in this journal without asking prior permission from the publisher or the author.
See: https://inatgi.in/index.php/rft/index for more information.

A small but growing body of work argues that the right way to combine the two perspectives is not to run them in parallel but to allow each to inform the other (Putra & Kabetta, 2022; Jakku, 2025; Rahman et al., 2023). Putra and Kabetta (2022) reported on a CI/CD pipeline that integrates SAST and DAST results. Jakku (2025) proposed embedding artificial-intelligence reasoning between static and dynamic stages so that continuous security can be maintained without ballooning developer workload. Bedoya et al. (2024) explored security chaos engineering with large language models as a way to surface defects that neither static nor dynamic testing finds in isolation. The latest research direction conceptualizes such systems as autonomous agents capable of orchestrating their own investigations (Khare et al., 2024; Zhou et al., 2024).

Within this thread of work, the present paper makes three contributions. First, it articulates a precise hybrid pattern for mass-assignment and related binding defects, mapping the static and dynamic stages onto distinct evidentiary roles. Second, it presents a reference implementation that requires no compilation, executes in seconds on representative codebases, and integrates with off-the-shelf CI/CD systems. Third, it reports empirical evidence that the hybrid approach materially reduces the false-positive burden compared with static analysis alone, which has practical consequences for whether developers actually act on security tool output.

III. CONCEPTUAL MODEL OF AN AUTONOMOUS DEVSECOPS AGENT

A. Agent, Not Scanner

The terminology matters. A scanner produces a list. An agent produces an investigation. A scanner reports symptoms; an agent reports symptoms together with the evidence required to act on them. We use the word agent throughout this paper to capture three properties: the system inspects multiple sources of evidence (source code, configuration, live service, schema); the system reasons about the relationships between those sources to decide whether a candidate finding is corroborated; and the system communicates its conclusions in a form actionable by a downstream automated system (a CI gate) and by a human (a developer). The conceptual move parallels the broader shift from rule-based scanning toward agentic security tooling that is being explored in cyber-physical systems (Lu, 2017b), Internet-of-Things security (Lu & Xu, 2019), industrial multi-agent supplier evaluation (Ghadimi et al., 2019), and large-language-model assisted reasoning (Khare et al., 2024; Steenhoek et al., 2024; Lu, 2019a; Zhang & Lu, 2021).

B. Three Reasoning Stages

We decompose the agent into three stages: candidate generation, candidate verification, and reporting with remediation. Candidate generation is the responsibility of the static engine; its goal is high recall. Candidate verification is the responsibility of the dynamic engine; its goal is high precision. Reporting is the responsibility of the orchestrator; its goal is to produce evidence-rich output that supports gating and developer action. The separation allows each stage to be optimized independently: recall does not need to be traded against precision, because the verification stage filters.

C. Evidence Composition

Each finding emitted by the agent carries a composite evidence record. A candidate flagged by the static stage but not confirmed by the dynamic stage is reported with severity MEDIUM rather than HIGH; the controller-layer pattern is still fragile and should be fixed, but it is not currently exploitable. A candidate that the dynamic stage confirms is reported with severity CRITICAL together with the request that triggered the finding and the response field that confirmed it. A candidate confirmed dynamically but not flagged statically, which is rare for mass assignment but possible for other defect classes, is reported with severity HIGH and a notation that source-code coverage of the agent's rule library may be incomplete. This evidence composition supports the kind of differentiated CI gating that practitioners actually want.

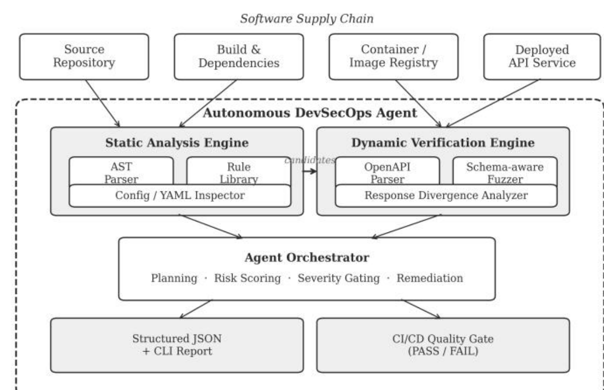

Figure 1. Reference architecture of an autonomous DevSecOps agent. The static engine processes repository artifacts; the dynamic engine probes the deployed service; the orchestrator integrates their findings and produces both human-readable and machine-readable outputs.

Figure 1 visualizes the architecture. The agent sits inside the development pipeline; it observes the same artifacts that flow through the supply chain, and it produces outputs that feed back into the gating policy. This

ISSN: © 2025 INATGI (Institute of Advanced Technology and Green Innovation). Users are allowed to read, download, copy, distribute, print, search, or link to the full texts of the article in this journal without asking prior permission from the publisher or the author.
See: https://inatgi.in/index.php/rft/index for more information.

positioning matters: the agent is not a separate tool that developers must remember to invoke; it is a step in the pipeline that runs on every commit and every merge candidate. The agent's outputs feed the JSON report consumed by downstream tooling and the CLI report consumed by humans.

IV. ARCHITECTURE AND IMPLEMENTATION

A. Static Analysis Engine

The static engine takes a project directory as input. It traverses the directory tree breadth-first, identifying Java source files and YAML configuration files while excluding build outputs, dependency directories, and version-control metadata. For each Java source file, it builds a lightweight abstract syntax tree using a regular-expression-based parser written in Python. The parser does not depend on the Java compiler, which removes the need for the project to be compilable for analysis to run. This property is important in practice because security tools that require a working build often cannot run early in a feature branch when the code is incomplete.

The abstract syntax tree records, for each class, its name and class-level annotations, its fields with their access modifiers and field-level annotations, the signatures of its methods with the annotations on their parameters, the presence and visibility of setter methods, and the cascade settings on Java Persistence API relationships. The parser also extracts the relevant subset of the application's YAML configuration. The output is a structured dictionary that the rule-matching engine queries.

The rule-matching engine compares each parsed file against a library of seven mass-assignment-specific rules. The rules are documented in Table I together with the severity assigned to each and the file role to which the rule applies.

TABLE I. STATIC DETECTION RULES FOR MASS ASSIGNMENT

Rule	Pattern Detected	Severity	File Role	Detection Basis
R-01	Direct entity binding via @RequestBody	HIGH	Controller	AST – annotation on controller parameter
R-02	Unsafe Lombok @Data on @Entity	HIGH	Model/Entity	AST – class-level annotation co-occurrence
R-03	Public setter on sensitive field	HIGH	Model/Entity	AST – setter visibility plus name match
R-04	Missing @Valid on request body	MEDIUM	Controller	AST – absence of annotation
R-05	CascadeType.ALL on JPA relationship	MEDIUM	Model/Entity	AST – relationship annotation value
R-06	Service-layer copy-based update	MEDIUM	Service	AST – bulk save without revalidation
R-07	fail-on-unknown-properties: false	LOW	Config (YAML)	YAML parser – key value match

Severity HIGH indicates direct mass-assignment exposure. Severity MEDIUM indicates a condition that increases exploitability when combined with a HIGH finding. Severity LOW indicates an observability gap.

Rules R-01 through R-03 cover the conditions that, in combination, suffice to produce an exploitable mass assignment. Rules R-04 and R-05 are aggravating factors that increase the impact of an R-01 finding. Rule R-06 was added during evaluation when service-layer behaviour was observed to introduce defects that controller-level scrutiny missed. Rule R-07 surfaces a configuration setting that, while harmless in isolation, suppresses the diagnostic that would otherwise alert an operator that mass-assignment attempts were occurring.

B. Dynamic Verification Engine

The dynamic engine activates after static analysis completes and after the operator provides a target URL. It selects the subset of R-01 candidates whose endpoint metadata can be extracted and proceeds in four stages. First, the OpenAPI specification parser reads an OpenAPI 2.0 or 3.0 document in JSON or YAML form and constructs an inventory of endpoints, methods, and request schemas. Second, the schema-aware payload generator constructs a baseline payload that conforms to the documented schema and overlays a fixed injection dictionary of sensitive-field name and value pairs. The dictionary contains role, isAdmin, balance, creditLimit, and several other names commonly associated with security-relevant state. Third, the request execution engine sends the resulting requests with operator-supplied authorization headers and a configurable inter-request delay. Fourth, the response analyzer determines whether the response status is in the 2xx range and whether the response body reflects the injected values. Where a corresponding GET endpoint exists, a follow-up read confirms persistence.

A finding is marked as exploitable only when both conditions hold: the server returned success and the injected values were observed in the response. A response in the 4xx range, an absence of injected values, or a 401 indicating that authorization is required all result in the candidate being cleared. A measure of state divergence between baseline and injection responses is computed as

ISSN: © 2025 INATGI (Institute of Advanced Technology and Green Innovation). Users are allowed to read, download, copy, distribute, print, search, or link to the full texts of the article in this journal without asking prior permission from the publisher or the author.
See: https://inatgi.in/index.php/rft/index for more information.

the symmetric set difference of key value pairs in the responses, normalized by the baseline. A divergence exceeding a configurable threshold indicates that injected fields changed server-side state.

C. Orchestration and Reporting

The orchestrator integrates the outputs of the two engines into a single report. The reporting module emits three formats: a colored console display intended for the development stage, a structured JSON report intended for downstream tooling, and a plain text variant suitable for archival. Each finding record carries the rule identifier, the file path and line number for static findings or the endpoint URL and method for dynamic findings, the matched snippet or injected payload, severity, dynamic confirmation status, and remediation guidance.

Two command-line flags govern integration into CI/CD: --fail-on-high causes the agent to exit with a non-zero status if any HIGH-severity finding is present, blocking the pipeline; --fail-on-medium does the same for MEDIUM. Teams unfamiliar with the agent typically deploy it first in advisory mode without the flags, then progressively activate gating as their backlog shrinks. This adoption pattern echoes the experience reported by Bedoya et al. (2024) and Jakku (2025) for AI-assisted DevSecOps tools.

V. EVALUATION

A. Experimental Setup

We evaluated the agent on three Spring Boot codebases. The Deliberately Vulnerable Application (DVA) is an intentionally insecure e-commerce service with twenty-four Java source files and one application.yml file, instrumented with mass-assignment patterns across controllers, models, and services. The Standard CRUD Application (SCA) consists of fourteen Java files and represents the kind of tutorial-driven code that commonly appears in introductory Spring Boot material; some endpoints use direct entity binding, others appropriately use Data Transfer Objects. The Hardened Reference Application (HRA) consists of eleven Java files and serves as a negative control: it uses DTOs throughout, applies @Valid on every request body, exposes no public setters on sensitive fields, and configures Jackson to fail on unknown properties.

Static scans ran on an Intel Core i7 workstation at 2.8 gigahertz with sixteen gigabytes of memory under Ubuntu 22.04. Dynamic scans were conducted against locally hosted DVA and SCA instances with matching OpenAPI specifications. HRA was not subjected to dynamic testing because the static engine produced no candidates.

B. Static Analysis Results

Table II summarizes findings per codebase per rule. The DVA codebase produced nineteen findings spread across rules R-01 through R-07, recovering every planted vulnerability. The SCA codebase produced eleven findings, of which four were genuine mass-assignment patterns and seven were medium-or-low aggravating factors. The HRA codebase produced no findings.

TABLE II. STATIC FINDINGS BY CODEBASE AND RULE CATEGORY

Codebase	Files	R-01	R-02	R-03	R-04	R-05	R-06	R-07	Total
DVA	24	4	2	3	5	2	0	3	**19**
SCA	14	2	0	2	4	0	2	1	**11**
HRA	11	0	0	0	0	0	0	0	**0**

DVA = Deliberately Vulnerable Application; SCA = Standard CRUD Application; HRA = Hardened Reference Application.

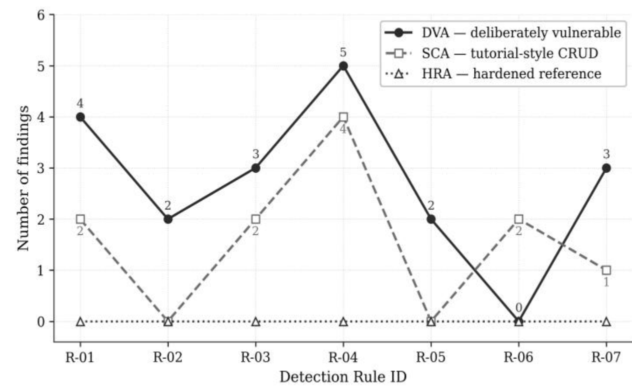

Figure 2. Static analysis findings by detection rule across the three evaluation codebases. The DVA trace peaks at rule R-04 (missing @Valid), reflecting the planted aggravating factors. SCA follows a similar but lower curve, and HRA remains flat as expected for a hardened baseline.

Figure 2 displays findings per rule. The shape of the DVA curve is consistent with a codebase in which both primary defects and aggravating factors are present; the SCA curve is offset downward because aggravating factors dominate over primary defects. The HRA curve, flat at zero, confirms that the rule library does not produce spurious matches on properly hardened code. This is the property that distinguishes a useful detector from one that practitioners abandon.

C. Dynamic Verification Results

Dynamic verification was run against eight candidate endpoints selected from R-01 findings: five from DVA and three from SCA. The results are summarized in Table III. Three endpoints were confirmed as exploitable; the

ISSN: © 2025 INATGI (Institute of Advanced Technology and Green Innovation). Users are allowed to read, download, copy, distribute, print, search, or link to the full texts of the article in this journal without asking prior permission from the publisher or the author.
See: https://inatgi.in/index.php/rft/index for more information.

remaining five were cleared. Among the cleared endpoints, two were rejected at the framework level by a custom InitBinder; two had service-layer filtering that the controller-level rule could not see; and one lacked sufficient endpoint metadata in the OpenAPI specification to allow a focused probe.

TABLE III. DYNAMIC VERIFICATION RESULTS

Endpoint	Method	App	Confirmed?	Decision Basis
/api/v1/accounts/register	POST	DVA	**Yes**	isAdmin field reflected in response
/api/v1/orders/create	POST	DVA	**Yes**	balance=999999 persisted (GET oracle)
/api/v1/accounts/profile/{id}	PATCH	DVA	No	Server returned 400 (InitBinder filter)
/api/v1/items/create	POST	DVA	No	Injected fields absent from response
/api/v1/admin/users/{id}	PUT	DVA	No	401 unauthorized; token required
/api/users/register	POST	SCA	**Yes**	role=ADMIN persisted and reflected
/api/products/create	POST	SCA	No	Service layer filtered injected fields
/api/users/update/{id}	PUT	SCA	No	Insufficient endpoint metadata

Each row corresponds to a single static R-01 candidate. The agent reduced the actionable HIGH-severity backlog from eight to three, a 62.5 percent false-positive filtering rate.

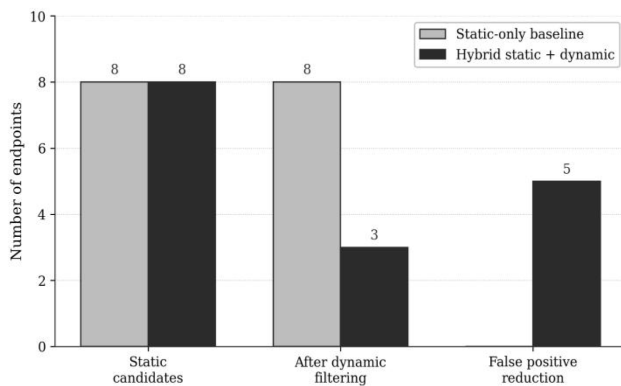

Figure 3. Endpoint counts under static-only and hybrid analysis. Both pipelines identify the same eight candidates, but only the hybrid agent reduces this to the three truly exploitable endpoints, eliminating five false positives.

Figure 3 visualizes the comparison. The data warrant two observations. First, every confirmed endpoint was associated with a controller binding a JPA entity directly without a Data Transfer Object wrapper, and with no service-layer filtering downstream; the static R-01 rule correctly identifies the necessary structural condition. Second, the cleared endpoints were not all examples of safe code; two had the same unsafe binding pattern as the confirmed endpoints but happened to be protected at runtime by mechanisms that the static rule could not observe. The agent therefore retains the underlying static finding at MEDIUM severity, on the principle that runtime defenses can be silently removed during refactoring and the contract-level fragility remains.

D. Severity Distribution and Performance

Figure 4 shows the CI/CD pipeline configuration into which the agent integrates. The static scan executes after build and before unit tests; the dynamic scan executes after unit tests and before the quality gate. The quality gate enforces a configurable maximum severity. A repository with --fail-on-high configured blocks pull-request merges that introduce a HIGH finding.

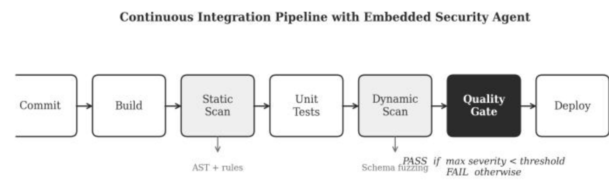

Figure 4. CI/CD pipeline configuration with the agent's static and dynamic stages embedded as gating steps. The quality gate enforces a configurable maximum severity threshold.

Figure 5 shows the distribution of findings by severity across the three codebases. The DVA codebase is dominated by HIGH-severity findings (nine of nineteen), reflecting its primary defect surface; SCA is dominated by MEDIUM findings, reflecting the predominance of aggravating factors over primary defects in tutorial-style code; HRA is empty. This pattern is consistent across multiple re-runs and serves as a sanity check that severity allocation is calibrated.

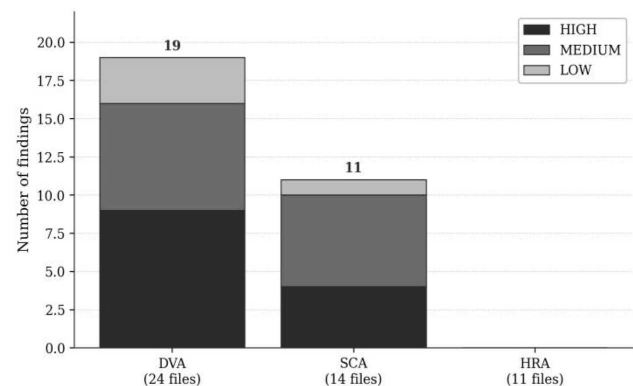

Figure 5. Severity distribution of findings across the three evaluation codebases, stacked by HIGH, MEDIUM, and LOW. The dominance of HIGH findings in DVA and

ISSN: © 2025 INATGI (Institute of Advanced Technology and Green Innovation). Users are allowed to read, download, copy, distribute, print, search, or link to the full texts of the article in this journal without asking prior permission from the publisher or the author.
See: https://inatgi.in/index.php/rft/index for more information.

MEDIUM findings in SCA matches their respective design intent.

Performance numbers are summarized in Figure 6 and Table IV. Static scans completed in well under two seconds for all three codebases, scaling approximately linearly with file count. Dynamic verification averaged 2.31 seconds per endpoint including network round-trip time, dominated by the 200 millisecond inter-request delay imposed to avoid flooding the test target. Peak memory was 112 megabytes for static analysis alone and 174 megabytes when dynamic verification was also active. These figures place the agent firmly within the time budget of a normal pre-merge gate even on modest CI infrastructure.

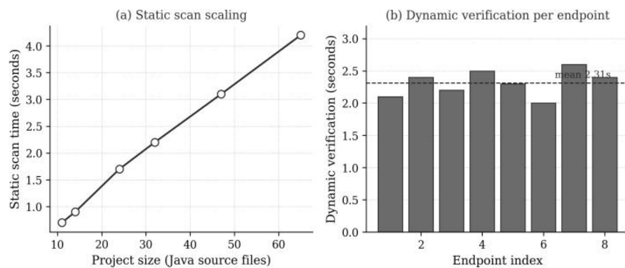

Figure 6. Performance characteristics of the agent. Panel (a) shows static scan time scaling approximately linearly with project size. Panel (b) shows the per-endpoint dynamic verification time, with mean 2.31 seconds.

TABLE IV. AGENT PERFORMANCE CHARACTERISTICS

Metric	Observed Value
Static scan on DVA (24 files)	1.7 seconds
Static scan on SCA (14 files)	0.9 seconds
Static scan on HRA (11 files)	0.7 seconds
Dynamic verification per endpoint	2.31 seconds (mean)
Dynamic verification: 8 endpoints	18.4 seconds total
Peak memory (static only)	112 megabytes
Peak memory (static + dynamic)	174 megabytes

All measurements taken on an Intel Core i7 (2.8 GHz) workstation with 16 GB RAM under Ubuntu 22.04.

VI. DISCUSSION

A. False-Positive Filtering and Developer Trust

The most consequential finding of the evaluation is that the dynamic stage reduced the actionable HIGH-severity backlog by five-eighths. This is not a marginal improvement. Decades of empirical research on developer behaviour around security tools have demonstrated that high false-positive rates are the single most reliable predictor of tool abandonment (Christakis & Bird, 2016; Johnson et al., 2013). When more than half of a tool's HIGH findings turn out to be unactionable, developers learn to ignore them. Once they have learned to ignore them, they ignore the true positives along with everything else. A 62.5 percent filtering rate at HIGH severity therefore changes not just the math of triage but the social dynamic of how the tool is received.

It is worth noting that the agent does not discard the cleared findings. They are downgraded to MEDIUM and retained in the report with an annotation indicating that runtime defenses are providing the protection. This is the right behaviour because runtime defenses are themselves contingent. A custom InitBinder, a service-layer filter, an authentication middleware all can be removed during refactoring with no visible effect at deployment time. The HIGH finding becomes a HIGH finding again as soon as the defense disappears. By recording it as MEDIUM with documented context, the agent preserves the information that future refactoring will need.

B. The Place of Hybrid Agents in Supply Chain Security

Software supply chain attacks of the SolarWinds class succeed because they exploit the gaps between distinct security perspectives (Martínez & Durán, 2021; Hammi & Zeadally, 2023). A defect that is invisible to source-code review but visible at runtime, or visible in source code but not exercised by integration tests, is exactly the kind of defect that supply chain attackers prefer. Hybrid agents, by design, close some of these gaps. The agent described here closes the specific gap that mass-assignment defects exploit: the gap between what controller code does and how the runtime service responds. Extending the same architectural pattern to other defect classes is an obvious direction. The same principles apply to broken object level authorization, server-side request forgery, and security misconfiguration; all of them have static signatures that demand dynamic confirmation.

There is also a broader systems-engineering implication. The same agent that defends against external attackers can be repurposed as a continuous-integration regression detector. When a previously passing endpoint begins to fail dynamic verification, the agent has detected a change in observable security posture. When a previously cleared endpoint begins to be flagged statically, the agent has detected a change in code structure that warrants attention. The agent thus operates as a security-oriented behavioural baseline for the application, in the same spirit that performance baselines are routinely captured for latency and throughput (Lu, 2025; Chen et al., 2024).

C. Limitations

ISSN: © 2025 INATGI (Institute of Advanced Technology and Green Innovation). Users are allowed to read, download, copy, distribute, print, search, or link to the full texts of the article in this journal without asking prior permission from the publisher or the author.
See: https://inatgi.in/index.php/rft/index for more information.

The current implementation has several limitations. The static engine supports only Java. The conceptual approach is language-agnostic, since direct input-to-object binding is structurally similar in Python, JavaScript, PHP, and Ruby, but the parser would need to be reimplemented for each language. Dynamic analysis changes the state of the target application; the agent has no rollback facility, so deployment in production-like environments is precluded. Coverage of the dynamic stage is bounded by the OpenAPI specification provided; endpoints absent from the specification are necessarily absent from the analysis. The injection dictionary is fixed; application-specific field names such as premiumStatus or trialExpiry will not be probed unless added explicitly.

Two further limitations are intellectual. The rule library was derived inductively from observed Spring Boot patterns and from the literature on broken object property level authorization; it is not exhaustive. We have not formally proved that the rules are sound, in the sense that every R-01 match corresponds to a structural condition required for exploit. In practice, false positives at the static layer are filtered by the dynamic layer; in principle, a rigorous theoretical treatment would be valuable. We also have not yet examined how the agent behaves on code generated by large language models, which may exhibit binding patterns absent from the training material on which the rule library was inductively built (Khare et al., 2024; Steenhoek et al., 2024; Zhou et al., 2024).

D. Toward Autonomous DevSecOps: A Research Agenda

Looking forward, four research directions are particularly promising. The first is the integration of large-language-model-based reasoning into the candidate-verification stage. Recent work has shown that frontier LLMs can reason about whether a particular code pattern is genuinely vulnerable with accuracy approaching that of trained human analysts (Khare et al., 2024; Steenhoek et al., 2024; Zhou et al., 2024). Used judiciously, an LLM could serve as a third evidentiary source alongside static and dynamic analysis, raising the recall of the agent for defect classes that have weak static signatures while preserving precision through cross-evidence corroboration. The second direction is the extension of the agent's scope from a single repository to a multi-repository supply chain. Current SBOM tooling provides the data substrate (Xia et al., 2024; Hammi & Zeadally, 2023), but the analytical glue that translates SBOM inventories into actionable security findings remains thin. The third direction concerns trust and provenance. As agents become more autonomous, the question of who audits the auditor becomes pressing. Recent work on blockchain-anchored audit logs (Lu, 2018; Lu, 2019b; Lu, 2022; Wu et al., 2025; Yang et al., 2025) suggests that the same cryptographic primitives developed for supply chain traceability and decentralized finance (Xu et al., 2024; Zhang & Lu, 2025) can be deployed in DevSecOps to produce tamper-evident records of which agents performed which analyses at which times. The fourth direction is the integration of API security findings into broader cyber-physical-system risk assessments (Lu, 2017b; Xu et al., 2021); APIs do not exist in isolation, and a defect in an API that controls a sensor network has implications that extend beyond the digital realm. Looking still further ahead, the maturation of quantum machine learning techniques (Lu et al., 2024) may eventually offer new analytical primitives for pattern-recognition over the very large evidence graphs that mature DevSecOps agents will accumulate.

A common thread runs through these directions: autonomous DevSecOps agents are not islands. They are nodes in an interconnected fabric of supply chains, software development pipelines, runtime monitoring stacks, and increasingly autonomous AI systems (Lu & Zheng, 2020; Lu, 2025; Chen et al., 2024). The conceptual move from scanner to agent that this paper has argued for is, ultimately, an invitation to redesign security tooling in a form fit for that fabric.

VII. CONCLUSION

Mass assignment and the family of broken object property level authorization defects with which it has been merged remain stubbornly present in production software a decade after they were first publicized. Existing tools fail to combine source-code analysis with runtime confirmation in a form that integrates seamlessly into developer workflows; the consequence is that vulnerable patterns ship and false-positive backlogs accumulate. This paper has argued that the answer is not better scanners but autonomous DevSecOps agents that compose static and dynamic perspectives within a single workflow. We have described a reference architecture for such an agent, reported its empirical performance on three representative Java Spring Boot codebases, and shown that the hybrid approach materially reduces the actionable false-positive burden compared with static analysis alone.

We have also tried to locate this contribution within a broader trajectory. The shift from scanner to agent that we observe in API security is paralleled by a shift from monitor to agent in cyber-physical systems, by a shift from rule to model in Internet-of-Things security, and by a shift from labeled-data classifier to large-language-

ISSN: © 2025 INATGI (Institute of Advanced Technology and Green Innovation). Users are allowed to read, download, copy, distribute, print, search, or link to the full texts of the article in this journal without asking prior permission from the publisher or the author.
See: https://inatgi.in/index.php/rft/index for more information.

model reasoner in software vulnerability analysis. Software supply chains, as the connective tissue of modern software production, will increasingly be governed by collections of such agents acting on each other's outputs. The architectural patterns explored in this paper are a small step in that direction. Much more work remains to translate the conceptual promise of autonomous DevSecOps agents into the trustworthy, observable, and accountable practice that critical software supply chains will require.

ACKNOWLEDGEMENT

Author contributions: All authors contributed equally to the conceptualization, design, implementation, evaluation, and writing of this work.

Funding: No external funding was received for this work.

Declarations: The authors declare no conflict of interest. No human or animal subjects were involved. All evaluation codebases used in this work are either publicly available or were developed by the authors for the purpose of the study.

REFERENCES

Ahsan, S. R., Yousuf, I. S., Khan, Z., et al. (2021). Digital supply chain management ecosystem powered by blockchain technology. In 2021 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE) (pp. 1–6). IEEE. https://doi.org/10.1109/CSDE53843.2021.9718500

Atlidakis, V., Geambasu, R., Godefroid, P., Polishchuk, M., & Ray, B. (2020). Pythia: Grammar-based fuzzing of REST APIs with coverage-guided feedback and learning-based mutations. arXiv preprint arXiv:2005.11498. https://doi.org/10.48550/arXiv.2005.11498

Bedoya, M., Palacios, S., Díaz-López, D., Laverde, E., & Nespoli, P. (2024). Enhancing DevSecOps practice with large language models and security chaos engineering. International Journal of Information Security, 23(1), 1–22. https://doi.org/10.1007/s10207-024-00909-w

Chen, Y., Lu, Y., Bulysheva, L., & Kataev, M. Y. (2024). Applications of blockchain in Industry 4.0: A review. Information Systems Frontiers, 26(5), 1715–1729. https://doi.org/10.1007/s10796-022-10248-7

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 (pp. 332–343). https://doi.org/10.1145/2970276.2970347

Corradini, D., Pasqua, M., & Ceccato, M. (2023). Automated black-box testing of mass assignment vulnerabilities in RESTful APIs. In Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE) (pp. 2553–2564). IEEE. https://doi.org/10.1109/ICSE48619.2023.00213

Croft, R., Babar, M. A., & Kholoosi, M. M. (2023). Data quality for software vulnerability datasets. In Proceedings of the 45th International Conference on Software Engineering (pp. 121–133). IEEE. https://doi.org/10.1109/ICSE48619.2023.00022

Ghadimi, P., Wang, C., Lim, M. K., & Heavey, C. (2019). Intelligent sustainable supplier selection using multi-agent technology: Theory and application for Industry 4.0 supply chains. Computers & Industrial Engineering, 127, 588–600. https://doi.org/10.1016/j.cie.2018.11.043

Godefroid, P., Huang, B.-Y., & Polishchuk, M. (2020). Intelligent REST API data fuzzing. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 725–736). https://doi.org/10.1145/3368089.3409719

Hammi, B., & Zeadally, S. (2023). Software supply-chain security: Issues and countermeasures. Computer, 56(7), 54–66. https://doi.org/10.1109/MC.2023.3273491

Isaja, M., Nguyen, P., Goknil, A., et al. (2023). A blockchain-based framework for trusted quality data sharing towards zero-defect manufacturing. Computers in Industry, 146, 103853. https://doi.org/10.1016/j.compind.2022.103853

Jakku, P. C. (2025). AI-powered static and dynamic analysis for continuous security in DevSecOps. SN Computer Science, 6(8), 852. https://doi.org/10.1007/s42979-025-04382-7

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 2013 International Conference on Software Engineering (pp. 672–681). IEEE. https://doi.org/10.1109/ICSE.2013.6606613

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

Khare, A., Dutta, S., Li, Z., Solko-Breslin, A., Alur, R., & Naik, M. (2024). Understanding the effectiveness of large language models in detecting security vulnerabilities. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering: New Ideas and Emerging Results (pp. 47–51). https://doi.org/10.1145/3639476.3639762

Ladisa, P., Plate, H., Martinez, M., & Barais, O. (2023). SoK: Taxonomy of attacks on open-source software supply chains. In 2023 IEEE Symposium on Security and Privacy (SP) (pp. 1509–1526). IEEE. https://doi.org/10.1109/SP46215.2023.10179304

Lipp, S., Banescu, S., & Pretschner, A. (2022). An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 544–555). https://doi.org/10.1145/3533767.3534380

ISSN: © 2025 INATGI (Institute of Advanced Technology and Green Innovation). Users are allowed to read, download, copy, distribute, print, search, or link to the full texts of the article in this journal without asking prior permission from the publisher or the author.
See: https://inatgi.in/index.php/rft/index for more information.

Lu, W., Lu, Y., Li, J., Sigov, A., Ratkin, L., & Ivanov, L. A. (2024). Quantum machine learning: Classifications, challenges, and solutions. Journal of Industrial Information Integration, 42, 100736. https://doi.org/10.1016/j.jii.2024.100736

Lu, Y. (2017a). Industry 4.0: A survey on technologies, applications and open research issues. Journal of Industrial Information Integration, 6, 1–10. https://doi.org/10.1016/j.jii.2017.04.005

Lu, Y. (2017b). Cyber physical system (CPS)-based Industry 4.0: A survey. Journal of Industrial Integration and Management, 2(3), 1750014. https://doi.org/10.1142/S2424862217500142

Lu, Y. (2018). Blockchain and the related issues: A review of current research topics. Journal of Management Analytics, 5(4), 231–255. https://doi.org/10.1080/23270012.2018.1516523

Lu, Y. (2019a). Artificial intelligence: A survey on evolution, models, applications and future trends. Journal of Management Analytics, 6(1), 1–29. https://doi.org/10.1080/23270012.2019.1570365

Lu, Y. (2019b). The blockchain: State-of-the-art and research challenges. Journal of Industrial Information Integration, 15, 80–90. https://doi.org/10.1016/j.jii.2019.04.002

Lu, Y. (2022). Implementing blockchain in information systems: A review. Enterprise Information Systems, 16(12), 1876–1907. https://doi.org/10.1080/17517575.2021.2008513

Lu, Y. (2025). The current status and developing trends of Industry 4.0: A review. Information Systems Frontiers, 27(1), 215–234. https://doi.org/10.1007/s10796-021-10221-w

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

Lu, Y., & Zheng, X. (2020). 6G: A survey on technologies, scenarios, challenges, and the related issues. Journal of Industrial Information Integration, 19, 100158. https://doi.org/10.1016/j.jii.2020.100158

Martínez, J., & Durán, J. M. (2021). Software supply chain attacks, a threat to global cybersecurity: SolarWinds' case study. International Journal of Safety and Security Engineering, 11(5), 537–545. https://doi.org/10.18280/ijsse.110505

Mazidi, N., Ciliberto, A., & Zaidman, A. (2024). Mining REST APIs for potential mass assignment vulnerabilities. In Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE) (pp. 1–10). ACM. https://doi.org/10.1145/3661167.3661204

Ohm, M., Plate, H., Sykosch, A., & Meier, M. (2020). Backstabber's knife collection: A review of open source software supply chain attacks. In Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA) (pp. 23–43). Springer. https://doi.org/10.1007/978-3-030-52683-2_2

Okafor, C., Schorlemmer, T. R., Torres-Arias, S., & Davis, J. C. (2022). SoK: Analysis of software supply chain security by establishing secure design properties. In Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (pp. 15–24). https://doi.org/10.1145/3560835.3564556

OWASP Foundation. (2023). OWASP API security top 10 – 2023 edition. https://owasp.org/API-Security/editions/2023/en/

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

Putra, A. M., & Kabetta, H. (2022). Implementation of DevSecOps by integrating static and dynamic security testing in CI/CD pipelines. In 2022 IEEE International Conference of Computer Science and Information Technology (ICOSNIKOM) (pp. 1–6). IEEE. https://doi.org/10.1109/ICOSNIKOM56551.2022.10034884

Rahman, A., Parnin, C., & Williams, L. (2023). The seven sins: Security smells in infrastructure as code scripts. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) (pp. 164–175). IEEE. https://doi.org/10.1109/ICSE.2019.00033

Steenhoek, B., Rahman, M. M., Roy, M. K., Alam, M. S., Barr, E. T., & Le, W. (2024). A comprehensive study of the capabilities of large language models for vulnerability detection. arXiv preprint arXiv:2403.17218. https://doi.org/10.48550/arXiv.2403.17218

Tao, F., Wang, Y. Y., & Zhu, S. H. (2023). Impact of blockchain technology on the optimal pricing and quality decisions of platform supply chains. International Journal of Production Research, 61(11), 3670–3684. https://doi.org/10.1080/00207543.2022.2053188

Wu, A., & Feng, Z. (2025). Rethinking broken object level authorization attacks under zero trust principle. arXiv preprint arXiv:2507.02309. https://doi.org/10.48550/arXiv.2507.02309

Wu, C. K., Tsang, K. F., Liu, Y., et al. (2019). Supply chain of things: A connected solution to enhance supply chain productivity. IEEE Communications Magazine, 57(8), 78–83. https://doi.org/10.1109/MCOM.2019.1800825

Wu, H. P., Liu, Z., Dong, H. Y., Lu, Y., & Xu, L. D. (2025). Revolutionizing internal auditing: Harnessing the power of blockchain. Enterprise Information Systems, 19(1–2), 2448003. https://doi.org/10.1080/17517575.2024.2448003

Xia, B., Bi, T., Xing, Z., Lu, Q., & Zhu, L. (2024). An empirical study on software bill of materials: Where we stand and the road ahead. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) (pp. 2630–2642). IEEE. https://doi.org/10.1109/ICSE48619.2023.00219

ISSN: © 2025 INATGI (Institute of Advanced Technology and Green Innovation). Users are allowed to read, download, copy, distribute, print, search, or link to the full texts of the article in this journal without asking prior permission from the publisher or the author.
See: https://inatgi.in/index.php/rft/index for more information.

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

Xu, R., Zhu, J., Yang, L., Lu, Y., & Xu, L. D. (2024). Decentralized finance (DeFi): A paradigm shift in the FinTech. Enterprise Information Systems, 18(9), 2397630. https://doi.org/10.1080/17517575.2024.2397630

Yang, L., Hou, Q., Zhu, X., Lu, Y., & Xu, L. D. (2025). Potential of large language models in blockchain-based supply chain finance. Enterprise Information Systems, 19(11), 2541199. https://doi.org/10.1080/17517575.2025.2541199

Zhang, C., & Lu, Y. (2021). Study on artificial intelligence: The state of the art and future prospects. Journal of Industrial Information Integration, 23, 100224. https://doi.org/10.1016/j.jii.2021.100224

Zhang, H., & Lu, Y. (2025). Web 3.0: Applications, opportunities and challenges in the next internet generation. Systems Research and Behavioral Science, 42(4), 996–1015. https://doi.org/10.1002/sres.3036

Zheng, X. R., & Lu, Y. (2022). Blockchain technology: Recent research and future trend. Enterprise Information Systems, 16(12), 1939895. https://doi.org/10.1080/17517575.2021.1939895

Zhou, X., Cao, S., Sun, X., & Lo, D. (2024). Large language model for vulnerability detection and repair: Literature review and the road ahead. ACM Transactions on Software Engineering and Methodology, 33(8), 1–36. https://doi.org/10.1145/3708533

Zhou, Y., Sharma, A., & Halfond, W. G. J. (2018). Automatic detection of mass assignment vulnerabilities in RESTful APIs. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA) (pp. 1–11). https://doi.org/10.1145/3213846.3213872

ISSN: © 2025 INATGI (Institute of Advanced Technology and Green Innovation). Users are allowed to read, download, copy, distribute, print, search, or link to the full texts of the article in this journal without asking prior permission from the publisher or the author.
See: https://inatgi.in/index.php/rft/index for more information.