

SBOM-Based Container Deployment Risk Assessment under Incomplete Runtime Observations

Byeongchan Park

Soongsil University

Sun-Jip Kim

`kims.j@hansei.ac.kr`

Hansei University

Research Article

Keywords: Software Bill of Materials, Container security, Vulnerability applicability, Incomplete runtime observations, Budget-constrained inspection

Posted Date: October 19th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-11178671/v1>

License: © ⓘ This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

SBOM-Based Container Deployment Risk Assessment under Incomplete Runtime Observations

Byeongchan Park¹, Sun-Jip Kim^{2*}

¹Department of Computer Science and Engineering, Soongsil University, Seoul, Korea

²Department of Convergence Security, Hansei University, Anyang, Korea

E-mail: kimsj@hansei.ac.kr

**Corresponding Author*

Abstract

Assessing container deployments from Software Bills of Materials (SBOMs) requires evidence about whether identified vulnerabilities apply to the target deployment. Incomplete runtime observations complicate this assessment because missing execution events do not establish non-applicability. This study proposes an SBOM-based assessment method that integrates component information, static analysis, runtime observations, and deployment configuration. The method classifies vulnerability applicability as confirmed, excluded within a declared scope, or unresolved, and maps these states to ALLOW, BLOCK, or HOLD decisions. Evidence is bound to the deployment context and revalidated when relevant conditions change. Within a limited inspection budget, an adaptive strategy prioritizes expected decision completion per unit cost, followed by expected resolution of unresolved obligations per unit cost. The evaluation used 100 deployment instances derived from 40 vulnerability cases across 20 Java projects, with independently established binary reference labels available for 90 instances. Experiments examined workload omission, shortened observation windows, event loss, and varying inspection budgets. Under a 120 s budget, the proposed method achieved a balanced correct decision rate of 78.4%, compared with 76.3% for resolution–cost inspection. The HOLD rate decreased from 22.0% to 20.0%, and mean budget-penalized completion time decreased from 55.9 s to 54.1 s. The proposed method’s false-allow and false-block rates were 1.2% and 2.0%, respectively. These results indicate that adaptive inspection ordering can improve deployment decision completion under the evaluated conditions while preserving unresolved states when available evidence is insufficient.

Keywords Software Bill of Materials; Container security; Vulnerability applicability; Incomplete runtime observations; Budget-constrained inspection

1 Introduction

Containers package applications and their dependencies into deployable images, supporting consistent execution and automated software delivery. These images also incorporate operating system packages, runtime environments, and direct and transitive dependencies that may contain known vulnerabilities[1]. Studies of Docker Hub images and containers for scientific data analysis have demonstrated the need for systematic vulnerability assessment and component maintenance[2, 3]. Ko and Kim proposed an automated pipeline for preventing the distribution of vulnerable container images[4]. Applying such approaches to operational services requires a decision process that translates vulnerability findings into deployment actions while accounting for the conditions under which each finding affects the target service.

A Software Bill of Materials (SBOM) provides structured information about software components and their relationships. The minimum elements defined by NTIA and the SPDX and CycloneDX specifications support the representation and exchange of this information[5–7]. Linking an SBOM to vulnerability databases enables the identification of potentially affected components and the tracking of changes across image versions. However, empirical studies have reported differences in SBOM adoption, information completeness, and consistency among generation tools[8, 9]. Deployment assessment therefore requires verification of the SBOM’s accuracy, its correspondence to the target image, and the evidence supporting component-to-vulnerability mappings.

The presence of a vulnerable component does not, by itself, establish exploitability in a particular deployment. Component inclusion, vulnerable code presence, code reachability, and the conditions required to trigger a vulnerability represent distinct assessment questions. Recent research on SBOM-based vulnerability management has examined the limitations of component-level findings and the benefits of incorporating function-call information[10]. Ponta et al. combined code-centric identification with static and dynamic analyses to assess vulnerable dependencies in application contexts[11, 12]. VulFinder further investigated test generation and execution for vulnerability reachability validation[13]. These approaches demonstrate the value of supplementing component inventories with application-specific evidence.

Nevertheless, combining static analysis and runtime observations leaves uncertainty that must be reflected in deployment decisions. Reflection and dynamic loading can limit the precision and coverage of Java static analysis[14], while practical analysis tools rely on assumptions that constrain the interpretation of their results[15]. Runtime observations are similarly limited to the workloads executed and the events successfully collected. Cimplifier illustrates how execution information can support container reduction[16], but an unobserved behavior cannot automatically be excluded from future executions. In particular, the absence of a vulnerable function from an execution trace is insufficient evidence of non-applicability.

This study addresses this uncertainty as a container deployment management problem. A pre-deployment workload may omit administrative operations,

exception-handling paths, or input-dependent functionality. Collector failures and event loss may prevent executed behavior from appearing in the available traces. Changes to an image or its configuration may also invalidate previously collected evidence. Interpreting these situations as evidence of safety can permit risky deployments, whereas blocking every unresolved finding can unnecessarily delay service updates. An operational assessment process must therefore distinguish supported conclusions from unresolved questions and identify the checks required to reach a deployment decision.

Vulnerability Exploitability eXchange (VEX) provides a foundation for communicating vulnerability impact and supporting justifications[17]. OpenVEX represents affected, not affected, fixed, and under investigation states[18]. These mechanisms already support explicit uncertainty and justified non-applicability. Building on them, this study examines how evidence obtained under incomplete observation should be validated, updated, and connected to deployment policy. The assessment must retain the scope and conditions of each conclusion and determine when additional analysis or renewed observation is necessary.

Severity and exploitation-related information provide further inputs to this process. The Common Vulnerability Scoring System (CVSS) characterizes vulnerability severity and incorporates threat and environmental metrics[19]. Exploit prediction research supports the prioritization of remediation effort under resource constraints[20]. However, these indicators do not independently establish applicability to a particular container. This study uses them to inform policy and inspection priorities while requiring deployment-specific evidence to support applicability judgments.

Resource constraints also limit the number and depth of checks that can be performed before deployment. Test prioritization research has examined how execution order and time constraints affect fault detection and cost effectiveness[21, 22]. For deployment management, the value of a check depends on its contribution to resolving the pending decision. Verifying a patch or a mandatory configuration condition may resolve a finding more efficiently than additional execution testing. When collection failures undermine existing traces, restoring observation and collecting new evidence may instead be necessary.

Accordingly, this study proposes an SBOM-based assessment method that combines vulnerability information, static analysis, runtime observations, and deployment configuration. Each vulnerability associated with a specific image and configuration is assigned one of three states: confirmed applicability, confirmed non-applicability within a specified scope, or unresolved applicability. The method distinguishes component presence and function execution from evidence establishing vulnerability applicability. Non-applicability requires a verifiable basis, such as a confirmed fix or the exclusion of a mandatory triggering condition within the declared scope. Missing execution events alone do not justify this state. Evidence is associated with image identifiers, configuration, analysis versions, and vulnerability information so that relevant changes trigger reassessment.

The resulting states are mapped to three deployment actions: ALLOW, BLOCK, and HOLD. Deployment is blocked when the applicability of a policy-relevant

vulnerability is confirmed. It is allowed when the assessment scope and input-quality requirements are satisfied and all policy-relevant findings have been resolved with valid evidence supporting non-applicability. Unresolved findings or invalid evidence result in a hold. Within the remaining inspection budget, the method selects additional checks by considering their expected cost and contribution to a conclusive deployment decision. This process connects evidence assessment, inspection selection, and policy enforcement in an iterative management procedure.

The evaluation design considers both decision correctness and operational cost. Public SBOM collections can support the examination of input diversity and quality[23], while reproducible Java vulnerability datasets can provide starting points for constructing validation cases[24]. Their existing labels require deployment-specific verification before use as ground truth. The evaluation separately considers incomplete workloads, shortened observation windows, event loss, and configuration changes. Comparisons use identical inputs and inspection budgets, measuring unsafe deployment approvals, unnecessary blocking, unresolved decisions, decision latency, and observation and analysis overhead. These measures allow security outcomes to be examined alongside the ability to progress eligible deployments.

The proposed contributions are as follows:

1. An evidence model for incomplete runtime observations. The model distinguishes confirmed applicability, scope-bounded non-applicability, and unresolved states, with explicit conditions for evidence validity and reassessment.
2. A budget-aware deployment decision procedure. The procedure connects additional inspection selection to ALLOW, BLOCK, and HOLD policies, directing limited resources toward evidence needed to resolve deployment decisions.
3. An evaluation framework covering security and operational cost. The framework examines decision errors, unresolved outcomes, deployment decision latency, and analysis overhead under distinct forms of incomplete observation.

Section 2 reviews related work. Section 3 defines the system model and assessment problem, and Section 4 presents the proposed method. Section 5 describes the implementation and experimental design. Section 6 presents and discusses the experimental results, including deployment decision performance, the effects of incomplete observations, and operational costs under different inspection budgets. Section 7 concludes the paper and outlines future work.

2 Related Work

2.1 Container Image and SBOM-Based Vulnerability Management

Container security research has examined the identification of vulnerable components and their management throughout image creation, storage, and deployment. NIST provides guidance on security risks associated with images,

registries, orchestrators, containers, and host operating systems[1]. Shu et al. investigated vulnerabilities in Docker Hub images[2], while Kaur et al. examined vulnerability assessment and image maintenance in containers for scientific data analysis[3]. These studies demonstrate the need for continuous inspection and maintenance of container components.

Ko and Kim proposed an automated pipeline to prevent the distribution of vulnerable container images[4]. Their work connects security inspection with the deployment process, providing a foundation for the management problem addressed here. This study focuses on findings whose applicability cannot be established from the available inspection results and specifies procedures for deployment holds, additional inspection, and reassessment.

SBOMs provide a structured basis for component identification and dependency tracking. The NTIA minimum elements and the SPDX and CycloneDX specifications support the representation and exchange of component information[5–7]. However, generating a standards-compliant document does not guarantee accurate component identification. Nocera et al. investigated SBOM adoption and use in open-source projects[8], while Wang et al. reported differences in standards adherence, consistency, and information accuracy among generation tools[9]. These findings indicate that input quality must be considered when SBOMs support deployment assessment.

Zhou et al. distinguished SBOM generation accuracy from the accuracy of subsequent vulnerability analysis[10]. They combined SBOM generation based on lock files and package management information with function-call analysis. Their work shows that component inventory accuracy and vulnerable code usage require separate assessment. The proposed method similarly uses SBOMs to identify candidate vulnerabilities and incorporates code-level and runtime evidence into deployment decisions.

2.2 Code Reachability Analysis and Execution-Based Validation

Code-centric vulnerability analysis examines the presence and potential execution of vulnerable code beyond package-name and version matching. Ponta et al. identified vulnerabilities using program constructs modified by security fixes and assessed reachability through static and dynamic analyses[11, 12]. Implemented in Eclipse Steady, this approach connects vulnerable code identification, application-specific usage assessment, and dependency update support. The combination of static analysis and runtime observation is therefore an established foundation for vulnerability assessment.

Analysis results must be interpreted within their underlying assumptions. Landman et al. investigated Java reflection usage and the limitations of static analysis[14], while Livshits et al. discussed the relationship between practical analysis assumptions and soundness[15]. Applied to deployment assessment, these findings indicate that the absence of a discovered call path cannot establish unreachability when relevant dynamic behavior is unsupported. Using a negative static-analysis result as evidence of non-applicability consequently requires an explicit analysis scope and stated assumptions.

VulFinder, proposed by Zhao et al., generates and executes tests based on static call paths to validate vulnerability reachability[13]. This approach can obtain evidence for paths not exercised by existing tests. Nevertheless, unsuccessful reproduction may result from inadequate generated inputs, environmental differences, or failure to reach the relevant code. The proposed method retains an unresolved state when such outcomes do not establish non-applicability.

Code reachability and exploitability also require separate interpretation. Even when a vulnerable function is reached, the resulting impact may depend on attacker control over inputs, configuration, and vulnerability-specific triggering conditions. This study treats reachability as one form of evidence and incorporates mandatory conditions and deployment policy to distinguish component inclusion, code execution, and vulnerability applicability.

2.3 Incomplete Runtime Observations and Vulnerability Status Representation

Runtime information supports the analysis of container resource usage and behavior. Cimplifier, proposed by Rastogi et al., identifies resources used during execution to remove unnecessary container components[16]. Ponta et al. also explain that the effectiveness of dynamic analysis depends on test coverage[12]. Applying execution evidence to deployment decisions therefore requires an understanding of the behaviors represented by the observations.

This study distinguishes three conditions that limit the use of runtime evidence. The first is workload omission, where relevant functionality or inputs are not exercised. The second is collection loss, where executed behavior is absent from the traces because of collector interruption or event loss. The third is evidence invalidation following changes to the image or configuration. These conditions require different responses: additional execution, restoration of collection followed by renewed observation, or reassessment of the changed deployment conditions.

VEX provides mechanisms for representing vulnerability impact[17]. OpenVEX defines the states `affected`, `not_affected`, `fixed`, and `under_investigation`, and requires a justification for a `not_affected` statement[18]. Product identifiers, timestamps, and updated statements support automated consumption, including deployment policies. Explicit uncertainty and justified non-applicability are thus established concepts on which this study builds.

The proposed method connects these representations to observation and inspection procedures. It associates evidence with the image, configuration, collection conditions, and reassessment requirements under which it was produced. Externally supplied VEX statements are considered alongside their scope and supporting evidence before informing deployment decisions. This connects vulnerability status information to operational actions and follow-up inspection.

2.4 Risk-Based Prioritization and Inspection Budget Management

Vulnerability management requires decisions about where to allocate limited resources. CVSS characterizes vulnerability severity, with version 4.0 providing

threat and environmental metrics alongside base metrics[19]. Jacobs et al. investigated exploit prediction as a basis for improving remediation prioritization[20]. These indicators support assessments of relative importance but remain distinct from direct evidence of applicability to a particular image and configuration.

The relationship between inspection order and cost has also been studied through test prioritization. Elbaum et al. examined how execution order affects fault detection[21], while Do et al. investigated the cost effectiveness of prioritization under time constraints[22]. These studies establish the importance of considering selection criteria and available resources together.

In this study, the selection unit comprises a vulnerability and an inspection action that can improve its assessment. Patch verification, configuration inspection, static-analysis refinement, and execution testing can differ in cost and expected contribution for the same vulnerability. A single action may also provide evidence relevant to several findings. Inspection priorities therefore consider severity, unresolved conditions, expected cost, and the potential contribution to a deployment decision.

Blocking and allowing deployment require different evidence scopes. Confirming the applicability of one vulnerability covered by a blocking policy may establish a blocking decision. Allowing deployment requires resolution of all policy-relevant findings and satisfaction of input-quality requirements. Consequently, an inspection order that maximizes vulnerability discovery or execution coverage may not minimize the time needed to resolve deployment. The proposed method incorporates this distinction into inspection selection and the ALLOW, BLOCK, and HOLD policies.

2.5 Comparison and Research Positioning

Table 1 summarizes the reviewed approaches and the deployment management problems addressed in this study. The comparison focuses on the evidence each approach contributes to deployment assessment.

This study focuses on the conditions under which evidence obtained from incomplete observations can support deployment decisions. The proposed procedure evaluates the scope and validity of static and dynamic findings and VEX statements, selects inspections to resolve uncertainty, and updates decisions when evidence or deployment conditions change. Its intended contribution is the integration of evidence management, inspection selection, and deployment policy into an iterative management procedure.

Evaluation resources also serve distinct purposes. Wild SBOMs provides publicly collected SBOMs for examining document diversity and component information[23], while Vul4J provides reproducible Java vulnerabilities with associated tests and fixes[24]. Neither directly establishes ground truth for an arbitrary container deployment. Evaluation cases therefore require verification of their execution environments and vulnerability conditions, and unsuccessful reproduction alone is not treated as a non-applicability label.

The proposed procedure is evaluated against an uncertainty-preserving baseline and fixed-order, random, severity-based, and cost-aware inspection strategies using identical inputs and budgets. The evaluation considers unsafe deployment approvals, the progression of eligible deployments, unresolved decisions, decision latency, and analysis cost. These measures support separate assessment of the contributions of evidence validity management and inspection selection to deployment management.

Table 1. Existing approaches and their relationship to deployment management

Research area	Representative work	Main approach and output	Management problem addressed in this study
Container image security	Shu et al.[2], Kaur et al.[3], Ko and Kim[4]	Image vulnerability assessment and security inspection pipelines	Mapping findings to ALLOW, BLOCK, and HOLD
SBOM quality and use	Nocera et al.[8], Wang et al.[9], Zhou et al.[10]	Analysis of SBOM accuracy and consistency, and incorporation of call information	Checking input quality and correspondence to the target image
Code-centric vulnerability analysis	Ponta et al.[11, 12], Zhao et al.[13]	Vulnerable code identification, reachability analysis, and execution-based validation	Distinguishing analysis scope and evidence strength
Execution-based container analysis	Rastogi et al.[16]	Identification of used resources and removal of unnecessary components	Determining observation limitations and the need for further inspection
Vulnerability impact representation	VEX[17], OpenVEX[18]	Exchange of impact states, justifications, and updated statements	Validating applicability conditions and triggering reassessment
Risk and inspection prioritization	CVSS[19], Jacobs et al.[20], Elbaum et al.[21], Do et al.[22]	Severity assessment, exploit prediction, and selection under resource constraints	Selecting actions according to cost and contribution to deployment decisions

3 System Model and Problem Definition

3.1 Assessment Scope and Deployment Context

The overall assessment workflow is illustrated in Fig. 1.

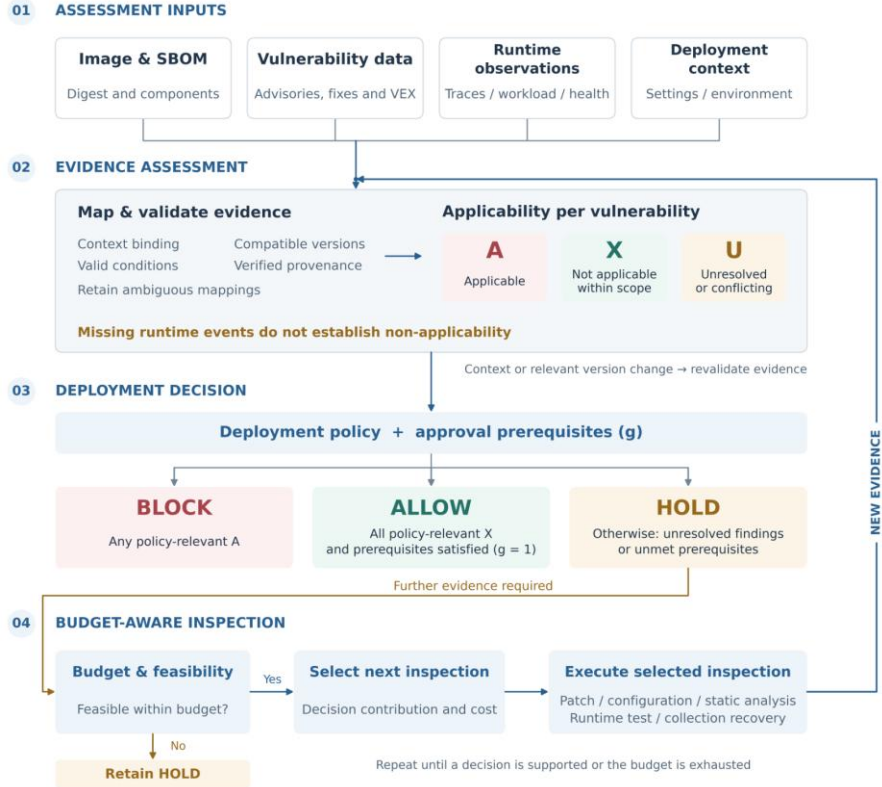


Fig. 1 Overview of SBOM-based container deployment assessment with evidence validation, applicability classification, and budget-aware reassessment

The framework combines SBOM and deployment information with runtime evidence to assess vulnerability applicability, determine deployment actions, and select additional inspections under a limited budget.

A deployment context is represented as

$$d = (h_I, h_C, \eta) \quad (1)$$

where h_I identifies the container image by its digest, h_C identifies the effective deployment configuration, and η describes the relevant execution environment and threat assumptions. These include runtime versions, externally supplied executable artifacts, exposed interfaces, and attacker capabilities considered by the assessment.

Changes affecting these properties create a revised context that requires reassessment.

Let $\mathcal{V}_t(d)$ denote the candidate vulnerabilities identified at assessment step t from the SBOM, component inspection, and vulnerability information. The candidate set may change as additional evidence becomes available. Ambiguous component identities and vulnerability mappings remain unresolved until verified; they are not discarded because matching or analysis fails.

A deployment policy π determines which candidates require resolution before deployment. Its criteria may include severity, component scope, service exposure, and organizational requirements. The corresponding set is denoted by $\mathcal{V}_t^\pi(d)$. Candidates with uncertain policy eligibility remain within the assessment until their eligibility is resolved.

The model assumes trusted analysis services and evidence whose provenance can be checked. Detected collection failures and unverifiable inputs are represented as assessment limitations. Undetected compromise of the assessment infrastructure is outside the threat model.

3.2 Evidence and Observation Incompleteness

Each evidence item records its source, claim, target vulnerability, deployment context, generation conditions, and analysis versions. Evidence may originate from component inspection, patch verification, static analysis, runtime execution, configuration inspection, or collector-health checks.

For an evidence item e , validity is defined as

$$\text{Valid}_t(e, d) = \text{Bind}(e, d) \wedge \text{Compatible}_t(e) \wedge \text{ConditionsHold}_t(e, d) \wedge \text{ProvenanceOK}(e). \quad (2)$$

Here, Bind checks correspondence to the target artifacts and configuration. Compatible checks whether the analysis and vulnerability-information versions remain applicable. ConditionsHold verifies the assumptions required by the evidence, and ProvenanceOK checks its origin and integrity. These predicates are specified for each evidence type.

Validity is claim-specific. Event loss may invalidate a coverage claim without invalidating an independently verifiable positive execution event. Conversely, a healthy collector does not establish that the workload exercised every relevant behavior. Evidence freshness therefore depends on its generation conditions and subsequent changes, in addition to its timestamp.

The model distinguishes three limitations:

1. Workload omission: relevant functionality or input conditions were not exercised.
2. Collection loss: executed behavior was not recorded because of interruption, sampling, or event loss.
3. Context change: the image, configuration, or environment changed after evidence generation.

These limitations require different responses. Workload omission may require targeted execution, collection loss may require renewed observation, and context changes require validation of the affected evidence. An observation containing no relevant event does not identify which explanation applies and cannot independently establish non-applicability.

3.3 Vulnerability Applicability States

For each candidate vulnerability v , the assessment maintains two evidence predicates. $P_t(v, d)$ indicates that valid evidence establishes applicability under the declared deployment context and threat assumptions. $N_t(v, d)$ indicates that valid evidence establishes non-applicability within the same scope.

Positive evidence must satisfy a vulnerability-specific validation rule. Examples include a validated vulnerability-triggering execution or verified conditions that are jointly sufficient to establish applicability. Component presence, a version match, or execution of a vulnerable function alone does not satisfy this requirement.

Negative evidence must establish a defensible exclusion. Examples include verification that the relevant fix is present, confirmation that vulnerable code is absent within the inspected scope, or exclusion of a mandatory triggering condition for the declared context. An unsuccessful test or an undiscovered call path is insufficient unless additional evidence establishes the required exclusion.

The applicability state is defined as

$$s_t(v, d) = \begin{cases} A, & P_t(v, d) = 1 \wedge N_t(v, d) = 0, \\ X, & P_t(v, d) = 0 \wedge N_t(v, d) = 1, \\ U, & \text{otherwise,} \end{cases} \quad (3)$$

where A denotes confirmed applicability, X denotes confirmed non-applicability within the declared scope, and U denotes unresolved applicability.

The unresolved state includes insufficient evidence, unsupported analysis conditions, and conflicting positive and negative evidence. Conflicts are retained explicitly for further inspection. They are not resolved by selecting the more recent statement without examining its supporting conditions.

Unsuccessful reproduction, inspection failure, and timeout do not independently establish non-applicability or invalidate previously accepted evidence. The applicability state is determined from all currently valid evidence. It remains U when neither applicability nor non-applicability is established, and becomes U when valid positive and negative claims conflict within the same assessment scope. Existing A or X conclusions are retained while their supporting evidence remains valid and no valid conflicting evidence is present.

VEX statements may contribute evidence to these predicates, but their status labels do not automatically determine the internal state. Their target, justification, and applicability conditions must be checked. Verified fixes can support X , while investigation states generally retain U . This permits the use of existing VEX representations[17, 18] within a deployment-specific assessment process.

State transitions are not necessarily permanent. If a relevant condition changes, previously accepted evidence is revalidated, and an A or X state may return to U .

3.4 Deployment Decision Policy

Let $g_t(d) \in \{0,1\}$ indicate whether the prerequisites for issuing a deployment approval are satisfied. These prerequisites include correspondence between the assessed artifacts and deployment target, completion of the declared inventory checks, and the absence of unresolved input-quality or coverage gaps within the supported assessment scope.

The deployment decision is

$$D_t(d) = \begin{cases} \text{BLOCK}, & \exists v \in \mathcal{V}_t^\pi(d): s_t(v, d) = A, \\ \text{ALLOW}, & g_t(d) = 1 \wedge \forall v \in \mathcal{V}_t^\pi(d): s_t(v, d) = X, \\ \text{HOLD}, & \text{otherwise.} \end{cases} \quad (4)$$

A confirmed policy-relevant vulnerability is sufficient to block deployment. Approval requires both valid non-applicability evidence for all policy-relevant candidates and satisfaction of the assessment prerequisites. When the candidate set is empty, approval still requires $g_t(d) = 1$; an empty scanner report alone is insufficient.

HOLD indicates that the available evidence does not support a terminal decision. It may trigger further inspection or preserve the request for later reassessment. Budget exhaustion, inspection timeout, and missing observations do not convert HOLD into ALLOW.

An ALLOW decision is limited to the known vulnerabilities, policy, and deployment context covered by the assessment. It does not assert the absence of every possible security weakness. Each decision is therefore associated with its context identifiers, policy version, supporting evidence, and unresolved limitations.

3.5 Budget-Constrained Inspection Problem

When a request remains on HOLD, additional inspection can improve the evidence available for decision-making. Let $\mathcal{A}_t(d)$ denote the feasible actions at step t . These may include component identification, patch verification, configuration inspection, static-analysis refinement, targeted execution, and collector validation followed by renewed observation.

An action can affect several vulnerability states or an assessment prerequisite. For example, identifying an ambiguous dependency may resolve multiple mappings, while validating a shared configuration condition may support several findings. Action value therefore depends on the current evidence and policy.

An adaptive inspection strategy selects an action according to

$$a_t = \sigma(d, H_t, b_t), \quad (5)$$

where H_t contains the evidence and action outcomes observed so far, and b_t is the remaining budget. The strategy may also stop when a terminal decision is reached or no feasible action remains.

The initial model assumes sequential inspection. Costs are measured as elapsed assessment time, including relevant collection, analysis, and orchestration overhead:

$$C_\sigma = \sum_{t=1}^{T_\sigma} c(a_t, o_t) \leq B, \quad (6)$$

where o_t is the observed outcome, $c(a_t, o_t)$ is the incurred cost, and B is the total budget. Action time limits enforce the remaining budget. A timeout consumes its incurred cost and leaves any unsupported conclusion unresolved. Cost estimates used for selection are kept separate from measured costs.

The objective must account for both deployment progress and incorrect decisions. Let $Y(d) \in \{\text{ALLOW}, \text{BLOCK}\}$ denote an independently established reference decision under the same assessment scope and policy. Define the balanced correct-decision rate as

$$J(\sigma) = \frac{1}{2} [\Pr(D_\sigma = \text{ALLOW} \mid Y = \text{ALLOW}) + \Pr(D_\sigma = \text{BLOCK} \mid Y = \text{BLOCK})]. \quad (7)$$

A HOLD outcome contributes to neither term. Balancing the two reference classes prevents the objective from being dominated by their relative frequencies.

The inspection problem is formulated as

$$\begin{aligned} \max_{\sigma} \quad & J(\sigma) \\ \text{subject to} \quad & C_\sigma \leq B, \\ & \Pr(D_\sigma = \text{ALLOW} \mid Y = \text{BLOCK}) \leq \epsilon_A, \\ & \Pr(D_\sigma = \text{BLOCK} \mid Y = \text{ALLOW}) \leq \epsilon_B, \end{aligned} \quad (8)$$

where ϵ_A and ϵ_B are predefined tolerances for incorrect approval and incorrect blocking.

This formulation specifies the evaluation objective; the reference decision is unavailable to the online strategy. Action selection uses only the current history, policy, and remaining budget. The error constraints are evaluation targets whose satisfaction requires empirical assessment. Decision latency, HOLD frequency, and resource overhead are reported alongside $J(\sigma)$ to characterize the operational trade-offs.

4 Proposed Assessment and Inspection Method

4.1 Evidence Assembly and Validation

The proposed assessment and inspection procedure is illustrated in Fig. 2.

Validated evidence determines deployment decisions, while estimated inspection outcomes guide the selection of additional checks under the remaining budget.

Each deployment request is initialized with the image digest, effective configuration, environment description, and deployment policy defined in Section 3. The method reconciles SBOM entries with the inspected artifacts and associates

identified components with candidate vulnerabilities. Ambiguous identities, incomplete mappings, and unsupported artifacts are recorded as unresolved assessment obligations.

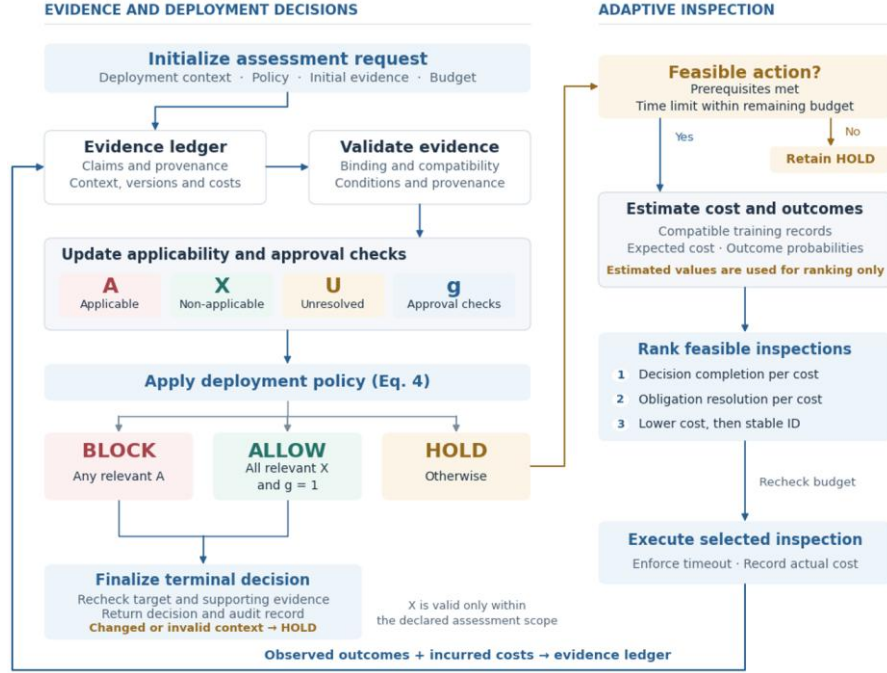


Fig. 2 Evidence-based deployment assessment and adaptive inspection under a limited budget

An evidence ledger stores each observation together with its target, provenance, generation conditions, and analysis versions. Records remain available for auditing even when they become invalid. Changes to component mappings or policy eligibility are recorded with supporting reasons so that findings cannot disappear through unsuccessful matching or budget exhaustion.

At assessment step t , the usable evidence set is

$$\mathcal{E}_t^{\text{valid}}(d) = \{e \in \mathcal{E}_t : \text{Valid}_t(e, d) = 1\}. \quad (9)$$

The method derives the positive and negative predicates in Eq. (3) from this set. Each inference retains links to its supporting records and the vulnerability-specific validation rule. A rule must specify the required observations, environmental assumptions, and conditions under which its conclusion remains applicable.

Conflicting evidence produces an unresolved state and an explicit conflict record. Resolution requires examining the competing claims, their target artifacts, and their validity conditions. A later timestamp alone does not override an earlier claim.

Approval prerequisites are evaluated alongside vulnerability states. They include artifact correspondence, completion of the declared inventory checks, and resolution of input-quality gaps. Consequently, resolving all currently identified vulnerabilities does not permit approval while an unresolved inventory or mapping issue could conceal additional policy-relevant findings.

4.2 Inspection Action Construction

Additional inspections are generated from unresolved vulnerability states and unmet approval prerequisites. Each action specifies its target artifacts, required inputs, execution conditions, admissible conclusions, and maximum execution time.

Actions sharing an inspection target may be grouped when one execution can supply evidence for several findings. For example, inspecting a dependency artifact may resolve multiple vulnerability mappings. Such grouping preserves the justification associated with each affected finding.

Table 2 defines the inspection categories and their evidential roles.

Table 2. Inspection actions and admissible assessment updates

Inspection category	Purpose	Admissible assessment update
Component and mapping verification	Resolve artifact identity, version, and vulnerability associations	Correct candidate mappings and discharge verified inventory obligations
Patch and configuration verification	Inspect fixes and mandatory triggering conditions	Support X when a fix or scope-valid exclusion is established
Static-analysis refinement	Examine unresolved call relationships and analysis assumptions	Add reachability evidence; support X only through a validated exclusion
Targeted execution	Exercise relevant functionality and validate triggering behavior	Support A when applicability is demonstrated; unsuccessful reproduction retains U
Collector validation and renewed observation	Diagnose collection limitations and obtain usable traces	Restore observation prerequisites and supply evidence for subsequent validation

An action becomes eligible only when its required inputs and execution conditions are satisfied. Collector-dependent inspections, for example, require the relevant collection prerequisites to be established. Missing prerequisites generate separate repair actions.

Let b_t denote the remaining budget and $\tau_t(a)$ the allocated time limit, including the action’s required result processing. The feasible action set is

$$\mathcal{A}_t^{\text{feas}}(d) = \{a \in \mathcal{A}_t(d) : \text{Ready}_t(a) = 1 \wedge 0 < \tau_t(a) \leq b_t\}. \quad (10)$$

An identical unsuccessful inspection is not repeated unless its inputs or prerequisites change. When no feasible action remains, the request retains HOLD.

4.3 Cost and Outcome Estimation

Inspection selection uses estimates of action cost and possible evidence outcomes. These estimates determine execution order; they do not establish vulnerability applicability.

The estimated cost $\hat{c}_t(a)$ is obtained from the median elapsed cost of compatible historical executions. Compatibility considers action type, execution environment, workload size, and time-limit class. When no compatible history exists, the allocated time limit provides the initial cost estimate. Actual elapsed costs are recorded separately and deducted from the remaining budget.

For each action, $\Omega(a)$ contains admissible outcome templates. A template describes possible changes to validated predicates or assessment prerequisites. Outcomes include confirmed applicability, verified exclusion, partial resolution, conflict, inconclusive execution, and timeout, where permitted by the action's validation rules. For actions affecting several findings, templates can represent joint changes.

Outcome frequencies are estimated from training-partition execution records using

$$\hat{p}_t(o \mid a, H_t) = \frac{n_{k,o} + 1}{\sum_{o' \in \Omega(a)} n_{k,o'} + |\Omega(a)|}, \quad (11)$$

where k identifies a compatible action and evidence class, and $n_{k,o}$ counts recorded outcomes of type o . The class uses observable properties of the action, affected targets, current evidence, and prerequisites. Unit pseudocounts provide an initial distribution when compatible observations are sparse.

Structurally inadmissible transitions are excluded from $\Omega(a)$. In particular, an unsuccessful runtime test cannot be represented as confirmed non-applicability. Joint outcome templates preserve dependencies among findings affected by the same action rather than assuming that their resolutions are independent.

Benchmark reference decisions are excluded from the selection features. Data partitioning and parameter selection are fixed before evaluation. The resulting estimates describe inspection outcomes and must not be interpreted as deployment exploitability probabilities.

4.4 Decision-Oriented Inspection Selection

The selector first considers whether an inspection can produce a terminal deployment decision. Let $\Phi_t(a, o)$ denote the hypothetical assessment state obtained by applying outcome template o to the current evidence state. The estimated probability of decision completion is

$$\hat{p}_t^{\text{dec}}(a) = \sum_{o \in \Omega(a)} \hat{p}_t(o \mid a, H_t) \mathbb{I}[D(\Phi_t(a, o)) \in \{\text{ALLOW}, \text{BLOCK}\}]. \quad (12)$$

The decision function in this calculation is the policy in Eq. (4). Therefore, the forecast accounts for the different evidence requirements of approval and blocking. A hypothetical applicability confirmation may complete a blocking decision, whereas an exclusion supports approval only when the remaining findings and prerequisites are also resolved.

When several inspections are required, an individual action may provide useful progress without completing the decision. Let Q_t contain the current unresolved obligations, including vulnerability assessments and approval prerequisites. Expected progress is

$$\hat{r}_t(a) = \sum_{o \in \Omega(a)} \hat{p}_t(o \mid a, H_t) \frac{|Q_t \setminus Q(\Phi_t(a, o))|}{\max(1, |Q_t|)}. \quad (13)$$

An obligation is removed only when a validated conclusion or justified scope determination resolves it. Newly discovered obligations remain in the subsequent assessment and prevent premature approval.

The selected action is

$$a_t^* = \text{lex arg max}_{a \in \mathcal{A}_t^{\text{feas}}(d)} \left(\frac{\hat{p}_t^{\text{dec}}(a)}{\hat{c}_t(a)}, \frac{\hat{r}_t(a)}{\hat{c}_t(a)}, -\hat{c}_t(a) \right). \quad (14)$$

Lexicographic ordering first prioritizes expected decision completion per unit cost, then expected obligation resolution per unit cost, and finally lower estimated cost. Remaining ties are resolved by a stable action identifier. This avoids introducing an arbitrary weighted sum of decision completion and partial progress.

The selector is a greedy heuristic for the objective in Eq. (8). Its decision quality, cost, and error constraints require empirical assessment. Prediction errors may reduce efficiency or increase HOLD outcomes, but predictions do not directly create accepted evidence or authorize deployment.

4.5 Execution, Reassessment, and Decision Finalization

The assessment executes one selected action at a time and appends its outcome to the evidence ledger. It then revalidates affected records, updates candidate mappings and applicability states, and reevaluates approval prerequisites. Inspection priorities are recomputed after every action because new evidence may change the value or feasibility of subsequent checks.

The complete procedure is presented in Algorithm 1.

Algorithm 1. Budget-constrained deployment assessment

Input:
Deployment context d , policy π , initial evidence E_0 , budget B

Output:

Decision D and an auditable assessment record

```
1: deadline  $\leftarrow$  current_time() + B
2: L  $\leftarrow$  initialize_ledger(d,  $\pi$ , E0, deadline)
3: while current_time() < deadline do
4:   if deployment_context_changed(d) then
5:     return HOLD with context-change reason and ledger L
6:   end if

7:   Evalid  $\leftarrow$  validate_evidence(L, d)
8:   update_candidates_states_and_prerequisites(Evalid, d,  $\pi$ )
9:   D  $\leftarrow$  evaluate_deployment_policy(d,  $\pi$ )

10:  if D  $\in$  {ALLOW, BLOCK} then
11:    return finalize_decision(D, d,  $\pi$ , L, deadline)
12:  end if

13:  b  $\leftarrow$  deadline - current_time()
14:  Afeas  $\leftarrow$  generate_feasible_actions(L, d, b)
15:  if Afeas is empty then
16:    break
17:  end if

18:  estimate_costs_and_outcomes(Afeas, L)
19:  a*  $\leftarrow$  select_action_using_Eq_14(Afeas)
20:  o  $\leftarrow$  execute_with_time_limit(a*, deadline)
21:  append_outcome_and_incurred_cost(L, a*, o)
22: end while

23: return HOLD with unresolved obligations and ledger L
```

Initialization, selection, validation, and result processing are included in the assessment budget. Action feasibility is checked again before execution to account for time consumed during planning. An expired deadline prevents the initiation of further inspections. Incomplete or unvalidated results remain unresolved.

Before issuing a terminal decision, finalization checks that the target image, configuration, policy, and supporting evidence still correspond to the assessed request. If a relevant change or an unresolved inconsistency is detected, finalization

returns HOLD. The resulting decision record identifies the applicable context, policy version, supporting evidence, incurred cost, and reason for termination.

For a held request, the record also identifies unresolved obligations and feasible follow-up checks. Subsequent assessment can reuse evidence only after its validity conditions have been checked against the new request. A previous approval is therefore not transferred automatically to a modified image or configuration.

Two properties follow from the separation between scheduling and evidence evaluation. First, missing observations and unsuccessful inspections cannot independently produce a non-applicability state. Second, budget exhaustion and the absence of further actions cannot produce an approval. These are procedural properties of the proposed rules; the correctness of individual applicability judgments remains dependent on the validity of the underlying evidence and validation rules.

5 Implementation and Experimental Design

5.1 Implementation and Experimental Environment

The assessment system was implemented with four modules: an SBOM reconciliation module, an evidence collection module, a vulnerability applicability assessment engine, and an inspection controller. The modules shared component information, inspection outcomes, and deployment states through a common evidence ledger.

The SBOM reconciliation module associated declared components with the corresponding artifacts in the container image. Findings with ambiguous component identifiers, versions, or dependency relationships remained unresolved. The evidence collection module converted artifact inspection, static analysis, configuration verification, and runtime observations into a common representation for storage.

The assessment engine verified that evidence was validly bound to the target image and configuration before updating vulnerability states and deployment decisions. The inspection controller identified feasible actions within the remaining time budget and selected the next action according to Eq. (14). Inspection failures and timeouts were recorded in the execution history, and their actual execution costs were deducted from the budget.

The experimental environment and principal settings are summarized in Table 3. All comparison methods were evaluated under identical resource limits and execution conditions. Assessment time was measured from the submission of the image, configuration, and SBOM to the return of the final decision. Image construction and SBOM generation were accounted for separately as preparation costs.

Table 3. Experimental environment and evaluation settings

Category	Item		Configuration
Hardware	CPU		AMD Ryzen 9 7950X, 16 cores and 32 threads
Hardware	Memory		64 GB
Execution environment	Operating system and kernel		Ubuntu Server 24.04 LTS, x86-64, Linux 6.8 series
Execution environment	Container and version	runtime	Docker Engine 27.5.1
Execution environment	JDK and build tools		Eclipse Temurin OpenJDK 17.0.14+7; Apache Maven 3.9.9
Analysis tools	SBOM generation and vulnerability identification		Syft 1.52.0, CycloneDX JSON format; Gype 0.119.0
Analysis tools	Static analysis and runtime collection		Soot 4.6.0; Java instrumentation agent based on Byte Buddy 1.17.5
Resource limits	CPU and memory per container		CPU quota equivalent to four logical CPUs; 8 GiB memory
Evaluation settings	Deployment policy and vulnerability information cutoff		All known Java dependency vulnerabilities included; a single vulnerability database snapshot fixed at the start of the experiments
Evaluation settings	Time budgets and reference observation window		$B \in \{30, 60, 120, 300\}$ s; default comparison budget: 120 s; reference observation window: $T_0 = 10$ s
Repetition settings	Repetitions and random seeds		Ten repetitions per stochastic condition; seeds 0–9 shared across all comparison methods

5.2 Dataset and Experimental Protocol

The experiments used cases from Vul4J [24], a public Java vulnerability dataset, for which vulnerable source code, developer-provided patches, and proof-of-vulnerability tests were available. Cases were included when they could be built, their vulnerabilities could be reproduced, and their deployment contexts could be explicitly defined. The final evaluation set comprised 20 projects and 40 vulnerability cases, yielding 100 deployment instances when deployment variants were included.

For the selected cases, 40 deployments containing vulnerable code and 40 patched deployments were constructed. An additional 20 deployments were created by configuring the applications to exclude a mandatory vulnerability condition where such control was possible. Related variants from the same project were assigned to the same partition to prevent information leakage between the training and test sets.

Reference labels were verified independently of the assessment system. A BLOCK label was assigned when at least one policy-relevant vulnerability was confirmed to be applicable to the deployment. An ALLOW label was assigned only when all policy-relevant candidates were verified as patched or excluded within the declared scope and the consistency of the component inventory was established. Failure to reproduce a vulnerability alone was not considered sufficient for an ALLOW label.

Of the 100 deployment instances, 90 had independently verified binary reference labels. The remaining 10 instances were evaluated separately for robustness and excluded from performance metrics requiring binary reference labels. Training, validation, and test partitions were formed at a ratio of 60:20:20 by project group, with all deployment variants from the same project assigned to a single partition. The results reported in Tables 4–6 were obtained from labeled deployment instances in the held-out test partition. Each stochastic condition was evaluated over 10 repetitions using random seeds 0–9, with identical seeds shared across comparison methods. Repeated runs did not constitute additional independent deployment samples.

The overall evaluation procedure is illustrated in Fig. 3.

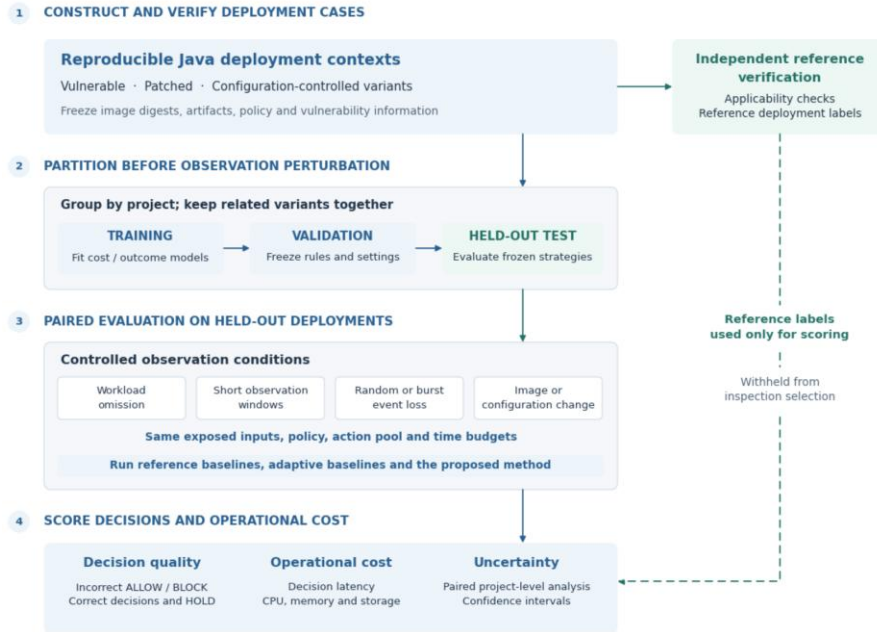


Fig. 3 Evaluation procedure for deployment risk assessment under incomplete runtime observations. Observation conditions were varied after project-level partitioning, and independently verified reference labels were excluded from inspection selection.

The data were partitioned into training, validation, and test sets at a ratio of 60:20:20 by project group. Execution histories collected from the training set were used to estimate inspection costs and outcomes, while the validation set was used to finalize the settings. The test set was evaluated using the frozen strategies, and its reference labels were used only for final scoring.

Incomplete observation conditions comprised workload omission, shortened observation windows, and event loss. Workload omission was implemented by excluding functional workload groups, whereas event loss was introduced by removing randomly selected events or contiguous event sequences from the generated observations. Under deployment-context changes, evidence validity was reassessed after modifying the image or configuration.

The comparison runs used identical inputs and perturbation seeds. Execution order was randomized, and the actual observation duration and event-loss proportion were recorded for each run.

5.3 Comparison Methods and Evaluation Metrics

The comparison included an SBOM-based gate, evidence-only assessment, fixed-order inspection, random-order inspection, severity–cost inspection, resolution–cost inspection, and the proposed method.

The SBOM-based gate determined deployment decisions primarily from policy-relevant component–vulnerability matches. Evidence-only assessment applied the proposed evidence validation and decision rules without additional inspection. The remaining inspection-based methods shared the same candidate actions and decision rules but differed in how they selected additional inspections. Methods requiring cost estimates used the same estimator.

Let $n_{y,d}$ denote the number of runs with reference label $y \in \{\text{ALLOW}, \text{BLOCK}\}$ and assessment decision $d \in \{\text{ALLOW}, \text{BLOCK}, \text{HOLD}\}$. The total number of runs in each reference class was defined as $N_y = \sum_d n_{y,d}$. The false-allow and false-block rates were calculated as

$$\text{FAR} = \frac{n_{\text{BLOCK},\text{ALLOW}}}{N_{\text{BLOCK}}}, \quad \text{FBR} = \frac{n_{\text{ALLOW},\text{BLOCK}}}{N_{\text{ALLOW}}}. \quad (15)$$

The false-allow rate measures the proportion of deployments requiring rejection that were incorrectly approved, whereas the false-block rate measures the proportion of eligible deployments that were incorrectly rejected. The overall HOLD rate was defined as

$$\text{HR} = \frac{n_{\text{ALLOW},\text{HOLD}} + n_{\text{BLOCK},\text{HOLD}}}{N_{\text{ALLOW}} + N_{\text{BLOCK}}}. \quad (16)$$

The balanced correct decision rate J , defined in Eq. (7), was evaluated alongside the error rates. This distinguished methods that merely reduced errors by returning HOLD for every input from methods that reached correct terminal decisions.

Decision speed was evaluated using actual latency and budget-penalized completion time:

$$\tau_i^* = \begin{cases} \tau_i, & D_i = Y_i, \\ B_i, & D_i \neq Y_i. \end{cases} \quad (17)$$

For each run i , τ_i denotes the measured elapsed time from assessment submission to the return of a decision. The budget-penalized completion time τ_i^* equals τ_i when the method returns the correct terminal decision within the budget. An incorrect decision or a HOLD outcome is assigned the full budget B_i , regardless of its actual return time. The mean of τ_i^* therefore summarizes both decision correctness and timeliness. Comparisons between methods use the same budget because changing the budget also changes the penalty assigned to unsuccessful outcomes.

Comparison methods were evaluated using matched deployment instances, observation conditions, and random seeds. Tables 4–6 report measured performance metrics and observed differences between methods. Repeated runs characterize variation under the specified stochastic conditions and do not increase the number of independent projects. The reported differences should therefore be interpreted as empirical results for the evaluated test set. Point estimates alone do not establish statistical significance or demonstrate that the same performance advantage will hold across other projects and deployment environments.

6 Experimental Results and Discussion

6.1 Overall Deployment Decision Performance

The comparison results for the entire test set are presented in Table 4. The proportion of correct decisions, false-allow and false-block rates, the HOLD rate, and decision time were compared under the same time budget.

The proposed method’s balanced correct decision rate is 78.4%, which is 2.1 percentage points higher than the 76.3% of resolution–cost inspection. Its HOLD rate is 2.0 percentage points lower, at 20.0% compared with 22.0%, and its mean budget-penalized completion time is 1.8 s shorter, at 54.1 s compared with 55.9 s. These differences provide a basis for evaluating a strategy that jointly considers the reduction of unresolved items and the likelihood of reaching a terminal deployment decision.

The proposed method’s false-allow and false-block rates are 1.2% and 2.0%, respectively. Compared with resolution–cost inspection, the false-allow rate is 0.4 percentage points lower, while the false-block rate is 0.2 percentage points higher. Differences relative to the SBOM-based gate also reflect the use of evidence-based decision rules. The contribution of inspection selection should therefore be assessed primarily against methods that share the same decision rules.

Table 4. Comparison of deployment decision performance

Method	Balanced correct decision rate $\backslash(J)$ (%)	False- allow rate (%)	False- block rate (%)	HOLD rate (%)	Mean τ^* (s)
SBOM-based gate	68.4	1.2	62.0	0.0	43.4
Evidence-only assessment	49.1	0.6	1.2	50.0	67.0
Fixed-order inspection	68.2	1.4	2.2	30.0	65.4
Random-order inspection	62.8	1.8	2.6	35.0	69.8
Severity–cost inspection	70.1	1.0	2.8	28.0	62.5
Resolution– cost inspection	76.3	1.6	1.8	22.0	55.9
Proposed method	78.4	1.2	2.0	20.0	54.1

Compared with evidence-only assessment, the proposed method increased the balanced correct decision rate by 29.3 percentage points and reduced the HOLD rate from 50.0% to 20.0%. These experimental results show that additional inspections converted some initially unresolved cases into terminal decisions. The comparison with evidence-only assessment reflects the combined effect of additional inspection and adaptive selection, whereas comparisons with methods sharing the same inspection actions and decision rules more directly assess the contribution of inspection ordering.

6.2 Effects of Incomplete Runtime Observations

Performance changes under incomplete observations are presented in Table 5. The time budget was fixed at 120 s, and the effects of workload omission, shortened observation windows, and event loss were compared separately. The remaining conditions and reference labels were held constant while each factor was varied. Changes were calculated by subtracting the value under lower incompleteness from the value under higher incompleteness.

When workload omission increased from 0% to 75%, the proposed method’s balanced correct decision rate decreased from 78.4% to 66.0%, while that of resolution–cost inspection decreased from 76.3% to 59.7%. The respective reductions were 12.4 and 16.6 percentage points, resulting in a difference of 6.3 percentage points under the higher-omission condition. The proposed method’s HOLD rate increased from 20.0% to 32.0%. These results indicate that the proposed method retained a higher correct decision rate under increased workload omission, although more deployments remained unresolved.

Table 5. Performance changes with increasing observation incompleteness

Factor	Comparison: lower → higher incompleteness	Proposed method ΔJ (pp)	Resolution –cost inspection ΔJ (pp)	Proposed method ΔFAR (pp)	Proposed method ΔHR (pp)
Workload omission	0% → 75%	–12.4	–16.6	+0.4	+12.0
Shortened observation window	10 s → 2.5 s	–4.2	–3.2	+0.2	+4.0
Random event loss	0% → 50%	–8.3	–10.4	+0.4	+8.0
Burst event loss	0% → 50%	–14.6	–16.8	+0.8	+14.0

When the observation window is shortened from 10 s to 2.5 s, the balanced correct decision rates were set to 74.2% for the proposed method and 73.1% for resolution–cost inspection. The reduction for the proposed method is 1.0 percentage point greater, narrowing the difference between the methods from 2.1 to 1.1 percentage points. Shorter observation windows can reduce initial evidence while increasing the budget available for additional inspections. Actual inspection counts and decision-transition histories therefore need to be analyzed together.

At an event-loss rate of 50%, the proposed method’s balanced correct decision rate was 70.1% under random loss and 63.8% under burst loss. The difference was 6.3 percentage points, with corresponding HOLD rates of 28.0% and 34.0%. Under the tested conditions, burst loss was associated with a lower correct decision rate and a higher HOLD rate than random loss.

In the experimental results, most of the reduction in the balanced correct decision rate corresponded to an increase in the HOLD rate. Across the four conditions in Table 5, the proposed method’s false-allow rate increased by 0.2–0.8 percentage points, its false-block rate increased by 0.2–0.4 percentage points, and its HOLD rate increased by 4.0–14.0 percentage points.

These results show that increasing observation incompleteness primarily increased deferred decisions, although incorrect decisions also increased. The reported differences are point estimates and should be interpreted together with project-level paired 95% confidence intervals.

6.3 Effects of Inspection Budget and Operational Costs

The experimental results for different time budgets are presented in Table 6. The experiments used time budgets of 30, 60, 120, and 300 s to compare the correct decision rate, HOLD rate, number of additional inspections, and operational costs. Identical inputs and resource limits were applied to all methods to evaluate how the time available for additional inspections affected deployment decisions.

Table 6. Effects of inspection budget on deployment decision performance and operational costs

Time budget t (s)	Proposed method J(%)	Resolution–cost baseline J(%)	Proposed method HOLD (%)	Mean additional inspections	Mean τ^* (s)	Memory range (MB)
30	58.8	50.6	39.8	0.9	25.3	2,000–4,000
60	71.4	65.2	27.1	1.7	37.2	2,000–4,000
120	78.4	76.3	20.0	2.8	54.1	2,000–4,000
300	83.0	82.1	15.2	4.1	90.8	2,000–4,000

As shown in Table 6, the correct decision rate increased from 58.8% to 83.0% for the proposed method and from 50.6% to 82.1% for the resolution–cost baseline as the time budget increased from 30 to 300 s. The proposed method’s HOLD rate decreased from 39.8% to 15.2%, while the mean number of additional inspections increased from 0.9 to 4.1. The differences in correct decision rates between the two methods were 8.2 and 6.2 percentage points at 30 and 60 s, respectively, and 2.1 and 0.9 percentage points at 120 and 300 s. The observed difference was largest under the shortest budget and narrowed as the budget increased, indicating that inspection ordering had a greater effect under shorter budgets in this evaluation.

The mean budget-penalized completion time was 25.3 s under the 30 s budget and 90.8 s under the 300 s budget. These values were calculated from the observed decision outcomes and measured elapsed times using Eq. (17). The reference classes were equally represented, and the measured mean times to correct decisions were 22, 28, 36, and 48 s for the respective budgets. Because a larger time budget also increases the penalty assigned to incorrect decisions and HOLD outcomes, values across different budgets should not be directly compared to rank processing speed. Under the same 120 s budget, the mean budget-penalized completion times were 54.1 s for the proposed method and 55.9 s for the resolution–cost baseline.

Measured memory usage ranged from 2,000 to 4,000 MB across the evaluated budget conditions. Under the default 120 s budget, runtime instrumentation reduced throughput by 5.0% and increased mean response latency by 7.0%. Mean CPU utilization was 45.0%, peak memory usage was 3,200 MB, and evidence storage per deployment was 20 MB. Changes in throughput and response latency were measured relative to the same workload with instrumentation disabled. CPU utilization was normalized so that the combined capacity of the four logical CPUs allocated to the evaluation container corresponded to 100%. These measurements characterize the operational costs associated with the observed decision performance.

7 Conclusion

This study proposed a risk assessment method that connects SBOM-based vulnerability information to container deployment decisions under incomplete runtime observations. The method distinguishes component-level vulnerability matches from vulnerability applicability in a specific deployment context and integrates evidence from static analysis and runtime observations. In particular, the absence of observed vulnerable behavior is not treated as sufficient evidence to exclude applicability, and uncertainty is explicitly retained when the available evidence is insufficient.

The proposed method manages evidence validity by associating evidence with container images, runtime configurations, and collection conditions, and derives ALLOW, BLOCK, or HOLD decisions accordingly. BLOCK is selected when the applicability of a vulnerability requiring blocking under the deployment policy is established. ALLOW is permitted only when applicability has been excluded for all assessed vulnerabilities and the approval prerequisites have been satisfied. Otherwise, HOLD is maintained so that insufficient observations or budget exhaustion cannot serve as grounds for approval. This structure supports traceability of both the evidence underlying a deployment decision and the remaining uncertainty.

The experimental evaluation examined deployment decisions under workload omission, shortened observation windows, event loss, and varying inspection budgets. At the 120 s budget, the proposed method achieved a balanced correct decision rate of 78.4%, a false-allow rate of 1.2%, a false-block rate of 2.0%, and a HOLD rate of 20.0%. Compared with resolution–cost inspection, the balanced correct decision rate was 2.1 percentage points higher, the HOLD rate was 2.0 percentage points lower, and the mean budget-penalized completion time was 1.8 s lower. The false-allow rate decreased by 0.4 percentage points, while the false-block rate increased by 0.2 percentage points. These results show the observed trade-offs of prioritizing inspections according to their contribution to a terminal deployment decision within the evaluated conditions.

The scope of this study is limited to known dependency vulnerabilities in Java applications running in Linux containers; unknown vulnerabilities and kernel-level vulnerabilities are outside its scope. Decision validity depends on the accuracy and scope of the evidence, while inspection selection efficiency depends on the accuracy of cost and outcome estimates. Future work should examine the generalizability of the observed performance benefits and operational costs across additional deployment environments and extend the approach to other language ecosystems and long-running workloads.

Statements and Declarations

Funding

The authors declare that no funds, grants, or other support were received during the preparation of this manuscript.

Competing interests

The authors declare that they have no competing interests.

Data availability

The data supporting the findings of this study are available from the corresponding author upon reasonable request.

Author contributions

Byeongchan Park designed the study, implemented the proposed method, conducted the experiments, and drafted the manuscript. Sun-Jip Kim supervised the research and reviewed and revised the manuscript.

References

- [1] Souppaya, M., Morello, J., Scarfone, K.: Application Container Security Guide. NIST Special Publication 800-190 (2017). <https://doi.org/10.6028/NIST.SP.800-190>
- [2] Shu, R., Gu, X., Enck, W.: A Study of Security Vulnerabilities on Docker Hub. In: Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy (CODASPY 2017), pp. 269–280 (2017). <https://doi.org/10.1145/3029806.3029832>
- [3] Kaur, B., Dugré, M., Hanna, A., Glatard, T.: An analysis of security vulnerabilities in container images for scientific data analysis. *GigaScience* 10(6), giab025 (2021). <https://doi.org/10.1093/gigascience/giab025>
- [4] Ko, S.-J., Kim, S.-J.: A Study on Pipeline Design Methods for Providing Secure Container Image Registry. *Journal of Internet of Things and Convergence* 9(3), 21–26 (2023). (In Korean). <https://doi.org/10.20465/KIOTS.2023.9.3.021>
- [5] National Telecommunications and Information Administration: The Minimum Elements for a Software Bill of Materials (SBOM) (2021). <https://www.ntia.gov/report/2021/minimum-elements-software-bill-materials-sbom>
- [6] SPDX Project: SPDX Specifications. <https://spdx.dev/use/specifications/>. Accessed 27 September 2026.
- [7] OWASP Foundation: CycloneDX Specification Overview. <https://cyclonedx.org/specification/overview/>. Accessed 27 September 2026.
- [8] Nocera, S., Romano, S., Di Penta, M., Francese, R., Scanniello, G.: On the adoption of software bill of materials in open-source software projects. *Journal of Systems and Software* 230, 112540 (2025). <https://doi.org/10.1016/j.jss.2025.112540>
- [9] Wang, C., Wu, J., Lyu, H., Ling, X., Luo, T., Wu, Y., Zhao, C.: A Large Scale Empirical Analysis on the Adherence Gap between Standards and Tools in SBOM. *ACM Transactions on Software Engineering and Methodology* 35(10), 1–37 (2026). <https://doi.org/10.1145/3788692>
- [10] Zhou, L., Dacier, M., Konstantinou, C.: A Reality Check on SBOM-based Vulnerability Management: An Empirical Study and A Path Forward. In: Proceedings of the Sixteenth ACM Conference on Data and Application Security and Privacy (CODASPY 2026), pp. 255–268 (2026). <https://doi.org/10.1145/3800506.3803490>
- [11] Ponta, S.E., Plate, H., Sabetta, A.: Beyond Metadata: Code-Centric and Usage-Based Analysis of Known Vulnerabilities in Open-Source Software. In: 2018 IEEE International Conference

- on Software Maintenance and Evolution (ICSME), pp. 449–460 (2018). <https://doi.org/10.1109/ICSME.2018.00054>
- [12] Ponta, S.E., Plate, H., Sabetta, A.: Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering* 25, 3175–3215 (2020). <https://doi.org/10.1007/s10664-020-09830-x>
 - [13] Zhao, Y., Zhou, Y., Xie, J., Huang, Z., Gong, L.: VulFinder: A Multi-Agent-Driven Test Generation Framework for Guiding Vulnerability Reachability Analysis. *IET Software* 2026(1), 6602283 (2026). <https://doi.org/10.1049/sfw2/6602283>
 - [14] Landman, D., Serebrenik, A., Vinju, J.J.: Challenges for Static Analysis of Java Reflection: Literature Review and Empirical Study. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 507–518 (2017). <https://doi.org/10.1109/ICSE.2017.53>
 - [15] Livshits, B., et al.: In Defense of Soundness: A Manifesto. *Communications of the ACM* 58(2), 44–46 (2015). <https://doi.org/10.1145/2644805>
 - [16] Rastogi, V., Davidson, D., De Carli, L., Jha, S., McDaniel, P.: Cimplifier: Automatically Debloating Containers. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017), pp. 476–486 (2017). <https://doi.org/10.1145/3106237.3106271>
 - [17] Cybersecurity and Infrastructure Security Agency: Minimum Requirements for Vulnerability Exploitability eXchange (VEX) (2023). <https://www.cisa.gov/sites/default/files/2023-04/minimum-requirements-for-vex-508c.pdf>
 - [18] OpenVEX: OpenVEX Specification, Version 0.2.0. <https://github.com/openvex/spec/blob/main/OPENVEX-SPEC.md>. Accessed 27 September 2026.
 - [19] FIRST: Common Vulnerability Scoring System version 4.0: Specification Document. <https://www.first.org/cvss/v4.0/specification-document>. Accessed 27 September 2026.
 - [20] Jacobs, J., Romanosky, S., Edwards, B., Adjerid, I., Roytman, M.: Exploit Prediction Scoring System (EPSS). *Digital Threats: Research and Practice* 2(3), Article 20, 1–17 (2021). <https://doi.org/10.1145/3436242>
 - [21] Elbaum, S., Malishevsky, A.G., Rothermel, G.: Test Case Prioritization: A Family of Empirical Studies. *IEEE Transactions on Software Engineering* 28(2), 159–182 (2002). <https://doi.org/10.1109/32.988497>
 - [22] Do, H., Mirarab, S., Tahvildari, L., Rothermel, G.: The Effects of Time Constraints on Test Case Prioritization: A Series of Controlled Experiments. *IEEE Transactions on Software Engineering* 36(5), 593–617 (2010). <https://doi.org/10.1109/TSE.2010.58>
 - [23] Soeiro, L., Robert, T., Zacchiroli, S.: Wild SBOMs: A Large-scale Dataset of Software Bills of Materials from Public Code. In: 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR), pp. 164–168 (2025). <https://doi.org/10.1109/MSR66628.2025.00036>
 - [24] Bui, Q.-C., Scandariato, R., Díaz Ferreyra, N.E.: Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques. In: Proceedings of the 19th International Conference on Mining Software Repositories (MSR 2022), pp. 464–468 (2022). <https://doi.org/10.1145/3524842.3528482>