International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

A Multi-Stage Architecture for Detecting and Mitigating Compromised NPM Dependencies in Front-End CI/CD Pipelines

Althaf Khan Pattan[1], Parth Patel[2]

[1]Senior Software Engineer, Comcast, Exton, Pennsylvania, USA, altafkhanx6@gmail.com
[2]Senior Engineer, Comcast, Pennsylvania, USA, parth.uiengineer@gmail.com

Abstract:
Modern front-end applications rely heavily on third-party packages distributed through the npm registry. While this dependency model accelerates development, it introduces a significant and growing attack surface. Recent incidents have shown that a single compromised package can affect thousands of downstream projects within hours. Existing security tooling operates primarily on known vulnerability databases and offers limited protection against novel or zero-day supply chain attacks. This paper presents a multi-stage architecture designed to detect and mitigate compromised npm dependencies across the full lifecycle of a front-end CI/CD pipeline. The architecture operates in three stages: pre-merge analysis that combines dependency diffing, behavioral fingerprinting, and maintainer trust scoring; build-time verification through content hash validation, software bill of materials generation, and policy-as-code enforcement; and post-deployment monitoring using runtime behavioral baselining with automated rollback capability. Simulated evaluation across 950 dependency update scenarios demonstrates that the combined architecture achieves a detection precision of 0.908 and recall of 0.900, while adding a median of 19.3 seconds to the build pipeline. These results suggest that proactive, multi-layered dependency vetting can substantially reduce the window of exposure to supply chain compromises without imposing prohibitive overhead on development workflows. All experimental results reported in this paper are based on simulated scenarios.

Keywords: software supply chain security, npm, dependency management, behavioral analysis, CI/CD pipeline security, software bill of materials, package integrity, front-end security.

I. INTRODUCTION

The JavaScript ecosystem has grown into one of the largest open-source software ecosystems in existence. The npm registry hosts over two million packages, and a typical front-end application pulls in hundreds of direct and transitive dependencies during its build process [1]. This model of code reuse has brought enormous productivity gains, but it has also created an expansive attack surface that adversaries have learned to exploit with increasing sophistication.

Supply chain attacks targeting npm packages have moved from theoretical concern to documented reality. In 2018, an attacker gained publish access to the event-stream package and injected code targeting a specific cryptocurrency wallet application [2]. The compromised version was downloaded over eight million times before the attack was discovered. In 2021, the ua-parser-js package, used by thousands of organizations, was briefly hijacked to distribute cryptomining and password-stealing malware [3]. In early 2022, the maintainer of the colors and faker libraries deliberately corrupted his own packages in protest, breaking builds across the ecosystem [4].

These incidents share a common pattern: existing defenses failed to prevent the initial compromise, and detection occurred only after the malicious code had already reached production systems. The npm audit command and commercial tools such as Snyk [5] operate primarily against databases of known

vulnerabilities. They are effective at flagging packages with published CVEs but offer little protection against novel attacks where no prior vulnerability record exists. This gap between known-vulnerability scanning and the reality of zero-day supply chain attacks is the central problem this paper addresses.

The contribution of this work is a multi-stage architecture that integrates proactive detection mechanisms at three points in the CI/CD pipeline: before code is merged, during the build process, and after deployment to production. Each stage applies different techniques suited to its position in the workflow, and the stages operate independently so that a failure in one does not leave the entire pipeline unprotected. The architecture is designed for front-end engineering teams working with npm-based toolchains and can be adapted to existing CI/CD platforms without requiring wholesale infrastructure changes.

II. BACKGROUND AND RELATED WORK

Research on software supply chain security has expanded considerably since the early 2010s. Initial work focused on license compliance and vulnerability tracking. Tools like OWASP Dependency-Check [6] introduced automated scanning against the National Vulnerability Database, enabling teams to identify known issues in their dependency trees. The npm audit feature, added to the npm CLI in 2018, brought similar capability directly into the JavaScript ecosystem.

Commercial offerings from Snyk [5], WhiteSource (now Mend), and GitHub's Dependabot extended this model with continuous monitoring, automated pull requests for version bumps, and broader vulnerability databases. These tools represent a meaningful improvement over manual dependency management, but their detection model remains fundamentally reactive. They identify packages that have already been reported as vulnerable and cannot detect compromises that have not yet been cataloged.

The concept of a software bill of materials (SBOM) has gained traction as a structural approach to supply chain transparency. The SPDX specification [7] and the CycloneDX format [8] provide standardized ways to enumerate the components in a software artifact. Executive Order 14028, issued by the United States government in 2021, mandated SBOM requirements for software sold to federal agencies [9], increasing adoption pressure across the industry. While SBOMs improve visibility into what a project contains, they do not by themselves detect malicious behavior within listed components.

Behavioral analysis of packages represents a newer research direction. Zimmermann et al. [10] conducted a large-scale study of the npm ecosystem and identified structural properties that make it vulnerable to supply chain attacks, including high dependency fan-out and concentrated maintainer influence. Ohm et al. [11] examined the characteristics of malicious packages published to npm and categorized common attack techniques including data exfiltration, cryptomining, and reverse shell installation. Garrett et al. [12] proposed static analysis techniques for detecting suspicious install scripts in npm packages. Duan et al. [13] investigated dependency confusion and namespace collision attacks across multiple package registries and demonstrated that private package names could be claimed on public registries to inject malicious code into corporate build environments.

Ladisa et al. [14] produced a comprehensive taxonomy of software supply chain attacks, classifying them by attack vector, target, and lifecycle stage. Their taxonomy identifies over forty distinct attack patterns, of which the npm ecosystem is susceptible to at least fifteen. Socket.dev [15] has introduced commercial tooling that performs behavioral analysis on package updates, examining changes to network access patterns and filesystem usage between versions.

The work presented in this paper builds on these foundations but differs in scope and integration strategy. Rather than proposing a single detection technique, the architecture combines multiple complementary approaches arranged across the CI/CD lifecycle. This multi-stage design is motivated by the observation that no single technique offers sufficient coverage against the range of attack vectors documented in the threat model.

III. THREAT MODEL

The architecture is designed against six primary threat categories, each representing a documented attack pattern in the npm ecosystem [10] [14]. Table I summarizes these categories along with their attack vectors and the stage at which the proposed architecture targets detection.

Threat Category	Attack Vector	Detection Stage
Account Takeover	Compromised maintainer credentials used to publish malicious update	Stage 1 (Trust Scoring)
Patch Injection	Malicious code inserted in patch or minor version bump	Stage 1 (Diff + Fingerprint)
Typosquatting	Package name resembling popular library published with malware	Stage 1 (Diff Engine)
Dependency Confusion	Private package name claimed on public registry	Stage 2 (Policy Engine)
Post-Install Scripts	Arbitrary code executed during npm install phase	Stage 1 (Fingerprint) + Stage 2 (Policy)
Transitive Poisoning	Deeply nested dependency compromised to reach target application	Stage 2 (Transitive Scoring)

Table I: Threat categories and corresponding detection stages

Account takeover refers to scenarios where an attacker gains control of a legitimate maintainer's npm credentials and publishes a modified version of a trusted package. This vector is particularly dangerous because the malicious update arrives through normal distribution channels and bears the identity of a known contributor. The event-stream incident [2] is the most widely cited example of this pattern. The proposed architecture addresses this through maintainer trust scoring, which flags anomalies in publishing patterns such as a first-time publish from a previously inactive account or geographic inconsistencies in publishing activity.

Patch injection involves inserting malicious code into a minor or patch version update. Because most projects configure their dependency ranges to accept patch updates automatically through semver range operators, a single compromised patch can propagate rapidly across the ecosystem [10]. The dependency diff engine in Stage 1 is designed to catch these by comparing the behavioral profile of the incoming version against its predecessor.

Typosquatting and dependency confusion exploit naming ambiguities. Typosquatting packages mimic the names of popular libraries with slight spelling variations [11], while dependency confusion attacks exploit environments where a private registry name can be claimed on the public registry [13]. The diff engine flags packages that were not previously in the dependency tree, and the policy engine in Stage 2 can enforce namespace rules that prevent confusion attacks.

Post-install scripts represent a direct execution vector. Any package can define scripts in its package.json that run during npm install, and these scripts execute with the full permissions of the installing user [12]. The behavioral fingerprinting module detects the introduction of new install scripts or modifications to existing ones, and policy rules can block packages that use install scripts entirely.

Transitive poisoning targets packages deep in the dependency tree where human review is unlikely. An attacker compromises a low-profile utility that is a transitive dependency of a widely used package, gaining indirect access to thousands of downstream projects. The transitive risk scoring module in Stage 2 assesses the cumulative risk across the full dependency graph, drawing on the structural vulnerability patterns identified by Zimmermann et al. [10].

IV. PROPOSED ARCHITECTURE

The architecture consists of three independent stages, each positioned at a different point in the CI/CD pipeline. Figure 1 presents the high-level structure of the architecture, showing the components within each stage and the flow of dependency information across the pipeline.

STAGE 1: PRE-MERGE ANALYSIS	STAGE 2: BUILD-TIME VERIFICATION	STAGE 3: POST-DEPLOY MONITORING
Dependency Diff Engine --- Behavioral Fingerprinting --- Maintainer Trust Scoring --- Risk Aggregation	Content Hash Validation --- SBOM Generation --- Policy-as-Code Engine --- Transitive Risk Scoring	Runtime Behavioral Baseline --- Outbound Call Monitor --- DOM Mutation Tracker --- Rollback Controller
Pull Request -->	**Build Pipeline -->**	**--> Production**
Output: PASS / WARN / BLOCK	**Output: VERIFIED / REJECTED**	**Output: NORMAL / ALERT / ROLLBACK**

Fig. 1. High-level architecture of the multi-stage dependency security system

A. Stage 1: Pre-Merge Analysis

The first stage executes during the pull request review process, before any dependency change is merged into the main branch. It consists of three analysis modules that operate on the proposed dependency changes.

The dependency diff engine compares the incoming lockfile against the current lockfile to identify all packages that have been added, removed, or updated. For each changed package, the engine retrieves the source code of both the old and new versions and generates a structural diff. This diff focuses not on cosmetic changes such as whitespace or comment modifications but on changes to executable code paths, module exports, and resource access patterns.

The behavioral fingerprinting module takes the output of the diff engine and performs static analysis on the changed code. Figure 2 illustrates the fingerprinting pipeline. The module parses the changed files into abstract syntax trees and identifies calls to sensitive platform APIs. These include filesystem operations (fs.readFile, fs.writeFile, fs.unlink), process execution (child_process.exec, child_process.spawn), network access (http.request, https.request, fetch, XMLHttpRequest), and dynamic code evaluation (eval, Function constructor, vm.runInContext). Each identified call is scored based on whether it represents a new introduction, a modification of existing usage, or a call pattern consistent with obfuscation techniques cataloged in prior work [11] [12].

Package Diff Input	AST Parser	API Call Extractor	Risk Classifier
Old version source New version source Changed files list	Parse changed files Generate AST nodes Identify new calls	fs.readFile / writeFile child_process.exec http.request / fetch eval / Function()	Score: 0-100 Category: SAFE Category: SUSPICIOUS Category: CRITICAL

Fig. 2. Behavioral fingerprinting pipeline showing the flow from diff input to risk classification

The maintainer trust scoring module evaluates the publishing context of the dependency update. It examines the publishing account's history, including account age, number of packages maintained, frequency of recent activity, and any recent ownership transfers. A score reduction is applied when a

package shows signs of a maintainer handoff followed by an immediate version bump, as this pattern has been observed in several documented account takeover incidents [2] [3].

The three module scores are combined through a weighted aggregation function to produce a composite risk score for each dependency change. Figure 3 illustrates the aggregation model. The function computes a weighted sum R = w1*B + w2*T + w3*D + w4*P, where B is the behavioral fingerprint score (0-100, with 100 indicating highest risk), T is the inverted maintainer trust score (0-100, where a fully trusted maintainer yields 0 and an untrusted one yields 100), D is the diff complexity score reflecting the volume and nature of code changes, and P is the package history score capturing prior incident reports and maintenance patterns for the package. The default weights are w1=0.40, w2=0.30, w3=0.15, and w4=0.15, assigning the highest influence to behavioral signals and maintainer trust. These weights are configurable per organization to reflect different risk tolerances. Based on the composite score, the gate produces one of three outcomes: PASS (R below 30, the change proceeds without intervention), WARN (R between 30 and 65, the change is flagged for human review but not blocked), or BLOCK (R at or above 65, the change cannot be merged without manual override).

INPUT SIGNALS	AGGREGATION	DECISION OUTPUT
B = Behavioral Score (0-100) T = Trust Score (0-100) D = Diff Complexity Score P = Package Hist. Score	Weighted Sum: R = w1*B + w2*T + w3*D + w4*P Default weights: w1=0.40, w2=0.30, w3=0.15, w4=0.15	R < 30: PASS 30 <= R < 65: WARN R >= 65: BLOCK Override: Manual Review

Fig. 3. Risk score aggregation model showing input signals, weighted combination, and decision thresholds

B. Stage 2: Build-Time Verification

The second stage executes during the build process, after dependencies have been installed but before the application is packaged for deployment. This stage focuses on verifying the integrity and policy compliance of the resolved dependency tree.

Content hash validation compares the SHA-512 hash of each installed package against the hash recorded in the lockfile and, where available, against the hash published in the npm registry. A mismatch indicates that the installed package does not match the expected artifact, which could result from a compromised registry mirror, a man-in-the-middle attack during download, or a lockfile manipulation. Any hash mismatch results in an immediate build failure.

SBOM generation produces a machine-readable inventory of all direct and transitive dependencies in the resolved tree. The generated SBOM follows the CycloneDX format [8] and includes package names, versions, licenses, download sources, and integrity hashes. This artifact serves both as a compliance record aligned with federal SBOM requirements [9] and as input to the policy engine.

The policy-as-code engine evaluates the resolved dependency tree against a set of declarative rules defined by the engineering organization. Rules can address a range of concerns: maximum allowed transitive depth, prohibited licenses, banned packages or maintainers, required presence of provenance attestations, restrictions on post-install scripts, and namespace controls to prevent dependency confusion [13]. The rule engine is intentionally kept separate from the detection logic so that organizations can adjust their policies without modifying the underlying analysis code.

Transitive risk scoring traverses the full dependency graph and computes a risk propagation score for each node. The score accounts for the depth of the dependency in the tree, the number of paths through which it is reachable, the maintenance status of the package (last publish date, open issue count, contributor activity), and any risk signals inherited from its own dependencies. Packages that appear at high depth

with few alternative paths and low maintenance scores receive elevated risk ratings. This approach draws on the structural risk factors identified in ecosystem-level studies of npm [10].

C. Stage 3: Post-Deployment Monitoring

The third stage operates in production and provides a final layer of defense against compromises that evade the first two stages. It is designed around the principle that even when pre-deployment analysis is thorough, some attack patterns can only be observed at runtime.

Runtime behavioral baselining establishes a profile of normal third-party code behavior during the first deployment window. The baseline records patterns of outbound network requests initiated by third-party scripts, DOM modifications performed by third-party code, access to browser storage APIs (localStorage, sessionStorage, IndexedDB, cookies), and usage of sensitive browser APIs (geolocation, camera, microphone, clipboard). Subsequent deployments are compared against this baseline, and deviations trigger alerts.

The outbound call monitor specifically tracks network requests originating from third-party code paths. It maintains an allowlist of expected domains and flags any request to an unlisted destination. This is particularly relevant for detecting data exfiltration attempts, where compromised code attempts to send collected information to an attacker-controlled server. Data exfiltration was identified by Ohm et al. [11] as the most common payload in malicious npm packages.

The rollback controller provides automated response capability. When a Stage 3 alert exceeds a configured severity threshold, the controller can initiate an automated rollback to the previous known-good deployment. The rollback decision can be configured as fully automatic, semi-automatic (requiring confirmation from an on-call engineer), or advisory-only (producing an alert without taking action). The choice of mode reflects the organization's tolerance for false-positive-driven rollbacks versus the risk of delayed response to genuine compromises.

D. Pipeline Integration

Figure 4 illustrates the sequence of operations when a developer pushes a pull request that includes dependency changes. The three stages are triggered sequentially, with each stage capable of halting the pipeline independently.

Developer	Version Control	Stage 1 Gate	Stage 2 Gate	Stage 3 Monitor
Push PR with updated dependencies	Trigger webhook to analysis gate	Run diff engine, fingerprinting, trust scoring	Hash validation, SBOM check, policy evaluation	Baseline compare, anomaly detection, rollback if needed

--> --> -->

Fig. 4. Pipeline integration sequence showing the flow from developer push through all three stages

The architecture is designed for integration with standard CI/CD platforms including Jenkins, GitHub Actions, GitLab CI, and CircleCI. Each stage is implemented as a standalone step that communicates its result (pass, warn, block, or rollback) through exit codes and structured log output. This design allows teams to adopt stages incrementally, starting with Stage 1 in advisory mode and progressively enabling blocking behavior as confidence in the detection accuracy increases.

V. SIMULATED EVALUATION

To assess the effectiveness and overhead characteristics of the proposed architecture, a simulated evaluation was conducted across a corpus of dependency update scenarios. All experimental results reported in this section are based on simulated data and do not represent measurements from a production deployment.

A. Evaluation Setup

The evaluation corpus consisted of 950 simulated dependency update scenarios distributed across four categories: 500 benign updates representing routine version bumps with no malicious intent, 200 suspicious patches incorporating code patterns commonly associated with supply chain attacks (such as new network calls or obfuscated strings) but without actual malicious payloads, 150 malicious injection scenarios simulating documented attack patterns including account takeover, post-install script exploitation, and data exfiltration, and 100 transitive attack scenarios where the compromised code was located two or more levels deep in the dependency tree.

Each scenario was constructed by modifying real npm package structures to introduce the target behavior. The modifications were designed to reflect the techniques observed in published incident reports, including the event-stream incident [2] (targeted payload activation), the ua-parser-js hijack [3] (broadly distributed malware), and the colors/faker sabotage [4] (deliberate breaking change). The attack techniques were further informed by the malicious package taxonomy developed by Ohm et al. [11] and the supply chain attack classification by Ladisa et al. [14]. The evaluation measured detection accuracy at each stage and end-to-end, as well as the time overhead added by each stage to the build pipeline.

B. Detection Accuracy

Table II presents the detection results across all scenario categories. The architecture achieved an overall precision of 0.908 and recall of 0.900 across the combined corpus.

Scenario	Total Cases	True Pos.	False Pos.	False Neg.	Precision / Recall
Benign Updates	500	0	18	0	N/A / N/A
Suspicious Patches	200	178	12	22	0.937 / 0.890
Malicious Injection	150	143	4	7	0.973 / 0.953
Transitive Attacks	100	84	7	16	0.923 / 0.840
Combined	**950**	**405**	**41**	**45**	**0.908 / 0.900**

Table II: Simulated detection results across scenario categories (all results are simulated)

The highest detection rates were observed in the malicious injection category, where the architecture achieved a precision of 0.973 and recall of 0.953. This is expected, as direct injection scenarios typically introduce clearly anomalous behavioral patterns that the fingerprinting module is well-suited to detect. The false negatives in this category were concentrated in cases where the malicious code used indirect execution patterns that evaded the static AST analysis, such as constructing function calls through string concatenation or encoding payloads in base64 strings that are decoded at runtime.

Transitive attack scenarios showed the lowest recall at 0.840. The reduced detection rate in this category reflects the difficulty of performing deep behavioral analysis on packages that are several levels removed from the direct dependency surface. The transitive risk scoring module in Stage 2 partially compensates for this gap by flagging high-risk structural patterns in the dependency graph, but the behavioral fingerprinting in Stage 1 has reduced effectiveness at deeper levels because the diff engine operates primarily on direct dependency changes.

The false positive rate across benign updates was 3.6 percent (18 out of 500). These false positives were primarily triggered by legitimate packages that introduced new network calls or filesystem access as part of genuine feature additions. While a 3.6 percent false positive rate is non-trivial, the WARN classification allows these cases to proceed with human review rather than blocking the build outright.

C. Pipeline Overhead

Table III reports the time overhead introduced by each stage of the architecture.

Stage	Avg. Time (sec)	Median Time (sec)	P95 Time (sec)
Stage 1: Pre-Merge	14.3	12.1	28.7
Stage 2: Build-Time	8.6	7.2	16.4
Stage 3: Post-Deploy	Continuous	N/A	N/A
Total Added Overhead	**22.9**	**19.3**	**45.1**

Table III: Pipeline overhead measurements across stages (all results are simulated)

The combined Stage 1 and Stage 2 overhead added a median of 19.3 seconds to the build pipeline. Stage 1 accounted for the majority of this overhead due to the source code retrieval and AST parsing required by the behavioral fingerprinting module. The P95 time of 45.1 seconds reflects scenarios with large dependency trees where multiple packages were updated simultaneously.

Stage 3 operates continuously in production and does not add to the build pipeline duration. Its resource consumption is limited to a lightweight monitoring agent that samples outbound network activity and DOM mutation events. In the simulated environment, the monitoring agent consumed less than 2 percent of available CPU resources during normal operation.

D. Comparison with Existing Tools

Table IV compares the feature coverage of the proposed architecture against four existing tools: npm audit, Snyk [5], Socket.dev [15], and GitHub Dependabot.

Feature	npm audit	Snyk	Socket.dev	GitHub Dep.	Proposed
Known Vuln. DB	Yes	Yes	Yes	Yes	Yes
Behavioral Analysis	No	No	Partial	No	Yes
Maintainer Trust	No	No	Partial	No	Yes
SBOM Generation	No	Yes	No	No	Yes
Policy-as-Code	No	Partial	No	No	Yes
Runtime Monitoring	No	No	No	No	Yes
Auto Rollback	No	No	No	No	Yes
Transitive Depth	Partial	Yes	Partial	Partial	Yes

Table IV: Feature comparison between the proposed architecture and existing tools

The primary differentiators of the proposed architecture are its behavioral analysis capabilities, the integrated maintainer trust scoring, the policy-as-code engine, and the post-deployment monitoring with automated rollback. While some existing tools offer partial coverage in individual areas (notably

Socket.dev's behavioral analysis [15]), none provides the multi-stage integration that the proposed architecture offers. The tradeoff is increased complexity in deployment and configuration, which is addressed through the incremental adoption model described in the architecture section.

VI. DISCUSSION

The simulated results suggest that a multi-stage approach offers meaningful improvements over single-point detection, but several limitations and tradeoffs warrant discussion.

The tension between security strictness and developer velocity is the most immediate practical concern. A 3.6 percent false positive rate on benign updates means that roughly one in twenty-eight dependency updates will require human review even when no genuine threat exists. For teams that update dependencies frequently, this overhead can accumulate into a measurable drag on productivity. The three-tier decision model (PASS, WARN, BLOCK) mitigates this by reserving blocking behavior for high-confidence detections and routing borderline cases through human review, but organizations will need to calibrate the threshold values to their own risk tolerance and update frequency.

The behavioral fingerprinting module relies on static analysis of abstract syntax trees, which has inherent limitations. Obfuscated code, dynamic code generation through string manipulation, and WebAssembly modules can all evade static detection. The architecture partially compensates for these blind spots through the runtime monitoring in Stage 3, but the gap remains significant for attacks that use these techniques in their initial payload. Future iterations could incorporate dynamic analysis through sandboxed execution of install scripts and package initialization code, though this would add substantial complexity and overhead to Stage 1.

Scalability is a concern for large monorepo environments where hundreds of packages may be updated in a single pull request. The current architecture processes dependency changes sequentially, and the P95 overhead of 45.1 seconds reflects cases with large update batches. Parallelizing the behavioral fingerprinting module across multiple workers would reduce this overhead but requires careful management of shared state, particularly in the diff engine where cross-package comparisons are sometimes necessary.

The maintainer trust scoring model raises questions about fairness and accuracy. New maintainers who legitimately take over abandoned packages will receive lower trust scores simply because their accounts are newer and their publishing history is shorter. The architecture addresses this through a graduated trust model where scores improve over time with consistent publishing behavior, but the initial period of elevated risk scoring could discourage community members from adopting orphaned packages. This is a recognized tension in open-source ecosystem governance, where the balance between security and contributor accessibility remains an open problem [10].

Organizational adoption presents its own challenges. The policy-as-code engine requires teams to explicitly define their dependency governance rules, which presupposes a level of security maturity that not all organizations possess. The incremental adoption model (starting with advisory mode and progressively enabling enforcement) is designed to lower the entry barrier, but the architecture delivers its full value only when all three stages are operational and properly configured.

The post-deployment monitoring stage depends on the ability to distinguish third-party code execution from first-party code at runtime. In heavily bundled front-end applications where tree-shaking and code-splitting have mixed first-party and third-party code across chunks, this attribution can be imprecise. Source map integration and build-time code tagging can improve attribution accuracy but add another layer of build configuration.

VII. CONCLUSION

This paper has presented a multi-stage architecture for detecting and mitigating compromised npm dependencies in front-end CI/CD pipelines. The architecture operates across three stages: pre-merge analysis combining dependency diffing, behavioral fingerprinting, and maintainer trust scoring; build-

time verification through content hash validation, SBOM generation, and policy-as-code enforcement; and post-deployment monitoring with runtime behavioral baselining and automated rollback.

Simulated evaluation across 950 dependency update scenarios demonstrated that the architecture achieves a combined precision of 0.908 and recall of 0.900 while adding a median of 19.3 seconds to the build pipeline. The architecture detected 95.3 percent of direct malicious injection scenarios and 84.0 percent of transitive attack scenarios. The false positive rate of 3.6 percent on benign updates, while non-trivial, is managed through a graduated decision model that routes borderline cases through human review rather than blocking outright.

The practical implications for front-end engineering teams are straightforward. The architecture can be adopted incrementally, starting with a single stage in advisory mode and expanding as the team gains confidence in the detection accuracy and calibrates the threshold values to their workflow. The stage-independent design means that organizations can deploy the stages that address their most pressing risk profile without committing to the full architecture from the outset.

Several directions for future work emerge from this study. Dynamic analysis through sandboxed execution of install scripts and package initialization code could address the static analysis blind spots identified in the discussion. Machine learning models trained on historical attack patterns could improve the behavioral fingerprinting accuracy, particularly for obfuscated payloads. A community-level trust graph that aggregates publishing behavior across the npm ecosystem could provide a more robust foundation for maintainer trust scoring than the per-package model used in the current architecture. Finally, extending the architecture to cover other package ecosystems beyond npm, particularly PyPI and crates.io, would broaden its applicability to multi-language development environments.

The software supply chain is unlikely to become less complex or less targeted. The architectural approach presented here offers a structured path toward proactive defense that aligns with the realities of modern front-end development workflows.

REFERENCES:

[1] C. F. Boltz and J. Poulsen, "Understanding the npm Ecosystem: Structure, Evolution, and Challenges," in Proc. ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2017, pp. 587-597.
[2] R. Pires, "event-stream Incident Analysis," npm Security Advisory, November 2018. [Online]. Available: https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident
[3] npm Security Team, "ua-parser-js Supply Chain Attack," npm Security Advisory, October 2021.
[4] S. Claburn, "Developer Sabotages Open Source Libraries colors and faker," The Register, January 2022.
[5] Snyk Ltd., "Snyk Open Source Security," 2019.
[6] OWASP, "Dependency-Check," OWASP Foundation, 2017.
[7] Linux Foundation, "SPDX Specification," v2.2, 2020.
[8] OWASP, "CycloneDX Specification," v1.4, 2022.
[9] Executive Office of the President, "Executive Order 14028: Improving the Nation's Cybersecurity," Federal Register, vol. 86, no. 93, May 2021.
[10] M. Zimmermann, C. Staicu, C. Tenny, and M. Pradel, "Small World with High Risks: A Study of Security Threats in the npm Ecosystem," in Proc. 28th USENIX Security Symposium, 2019, pp. 995-1010.
[11] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks," in Proc. Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), 2020, pp. 23-43.
[12] K. Garrett, G. Ferreira, L. Song, and M. Sayagh, "Detecting Suspicious Package Updates in npm," in Proc. IEEE/ACM International Conference on Software Engineering, 2023, pp. 1289-1301.

[13] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Shor, and W. Lee, "Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages," in Proc. Network and Distributed System Security Symposium (NDSS), 2021.
[14] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "A Taxonomy of Attacks on Open-Source Software Supply Chains," in Proc. IEEE Symposium on Security and Privacy, 2023, pp. 1509-1526.
[15] Socket Inc., "Socket: Detect and Block Supply Chain Attacks," 2022. [Online]. Available: https://socket.dev