Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

Machine Learning-Based Detection of Malicious Software Packages in Software Supply Chains: A Comprehensive Survey and Design Framework

Senthil Kumar V
Kumaraguru college of technology, Coimbatore

Mukhil Kumaran S
Kumaraguru college of technology, Coimbatore

Nayanthara C
Kumaraguru college of technology, Coimbatore

Shankesh R G
Kumaraguru college of technology, Coimbatore

Subhiksha M
Kumaraguru college of technology, Coimbatore

Kaushika S
Kumaraguru college of technology, Coimbatore

Abstract - The open-source ecosystem has become the backbone of modern software—and that makes it a target. PyPI, npm, and Maven together host hundreds of thousands of packages that developers pull into their projects every day, often without a second thought about what might be hiding inside. Attackers have noticed, and the techniques they use have grown considerably more sophisticated: typosquatting, dependency confusion, backdoored updates, packages that look legitimate but are anything but. The traditional defenses haven't kept pace. Signature-based scanning is helpless against anything new, and the sheer volume of daily uploads makes manual review a fantasy. This paper takes stock of where detection stands today—surveying static analysis, dynamic analysis, heuristic methods, and behavioral monitoring—and maps out where each approach falls short. From that foundation, we sketch out a detection framework that pulls multiple signals together: package metadata, code analyzed through NLP techniques, dependency relationships modeled with graph neural networks, and risk explanations that a human analyst can actually act on. The goal is something that works in real time, plugged directly into CI/CD pipelines, and doesn't need to have seen a threat before in order to catch it. No implementation is presented here; the contribution is a design specification and theoretical grounding for the system that still needs to be built.

Keywords - Software Supply Chain Security, Machine Learning, Malicious Package Detection, Static Analysis, Dynamic Analysis, Graph Neural Networks, Explainable AI, PyPI, npm, Zero-Day Detection

INTRODUCTION

Modern software development runs on borrowed code. The Python Package Index, npm, and Maven Central collectively host millions of libraries that developers reach for without a second thought—and that trust is exactly what makes them valuable targets. When a package is compromised, the damage doesn't stop at one application; it propagates silently to every project that depends on it, often before anyone realizes something has gone wrong.

The SolarWinds incident in 2020 made that threat impossible to ignore. A backdoored software update reached roughly 18,000 organizations, including U.S. federal agencies, before it was caught. The attack didn't break through firewalls or crack passwords—it rode the update mechanism that organizations had every reason to trust. That's what makes supply chain exploitation so dangerous: it turns the normal workflow into the attack vector.

The tools most organizations rely on to guard against this haven't kept up. Signature-based detection is fast and simple, but it only catches what it's already seen. The moment an attacker tweaks a payload, the signature becomes useless. Manual code review is thorough, but no team can realistically inspect the millions of packages published every month. And with DevOps pipelines moving at the speed they do today, a detection system that can't keep up in real time isn't much of a defense.

Machine learning offers a more promising path. Models trained on large collections of benign and malicious packages can learn to recognize suspicious patterns—unusual API calls, strange dependency structures, metadata that doesn't quite add up—without needing a prior match. Research in this space has been growing quickly, and the results across metadata analysis, code-level NLP, and graph-based dependency modeling are encouraging.

What's missing is integration. The field has produced a collection of strong individual techniques that don't talk to each other. They tend to work on single ecosystems, rely on historical patterns to detect new threats, and rarely produce explanations that a security analyst can act on. The result is a gap between what research has shown is possible and what actually exists in production.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

This paper is an attempt to close that gap at the design level. It surveys the threat landscape and the existing detection literature, identifies the four most critical shortcomings in current approaches, and proposes a unified framework that pulls metadata, code, behavioral signals, and dependency graphs together into a single explainable, real-time detection system. The paper is organized as follows: Section 2 covers the threat landscape, Section 3 reviews detection methodologies, Section 4 examines ML-based approaches, Section 5 identifies research gaps, Section 6 states the problem formally, Sections 7 and 8 describe the proposed architecture and feature design, Section 9 discusses broader considerations, and Section 10 closes with directions for future work.

LITERATURE SURVEY

The body of work on supply chain security and machine learning-based detection has grown substantially in recent years, spanning threat taxonomy, detection tooling, graph-based modeling, and foundational ML methodology. What follows examines the fifteen most relevant contributions in detail, drawing out both what each work contributes and where it leaves room for improvement.

Ohm et al. [1] laid important groundwork by assembling and categorizing a collection of real-world open-source supply chain attacks across PyPI, npm, and Maven. Their taxonomy—covering typosquatting, dependency confusion, account hijacking, and backdoored updates—gave the research community a shared vocabulary and a clearer picture of the attack surface. The limitation is that the work stops at description. No automated detection mechanism is proposed, and the analysis, while thorough, is retrospective. It tells you what has happened; it doesn't help you catch what's happening now.

Ladisa et al. [2] took a broader view, producing a systematic treatment of attack vectors across multiple ecosystems and formalizing them into a threat model. The SoK framing made it easier to reason about the problem space and compare defenses across different contexts. Like Ohm et al., however, the contribution is primarily analytical. The paper provides a map of the terrain without offering a route through it—no ML model, no detection pipeline, and limited practical guidance for engineers trying to build one.

Birsan [3] demonstrated dependency confusion in a way that was impossible to dismiss: by successfully planting malicious packages inside the build systems of Apple, Microsoft, and dozens of other companies, exploiting a simple precedence flaw in how package managers resolve names. The technical insight was sharp and the real-world impact was immediate. What the work doesn't provide is any mechanism for catching this kind of attack—it's a proof of concept that exposed the vulnerability without addressing the detection side of the equation.

Duan et al. [4] moved closer to practical detection by combining static and dynamic analysis across npm, PyPI, and RubyGems to examine how malicious packages behave at install time and runtime. Their analysis of 174 confirmed malicious packages revealed consistent patterns in API usage and system call behavior that distinguished them from benign counterparts. The dataset size, however, is a significant constraint—174 samples is nowhere near enough to train a robust ML classifier—and the approach lacks the kind of scalable, automated classification pipeline needed for deployment on a repository processing thousands of uploads per day.

Sejfia and Schäfer [5] brought machine learning into the picture more directly, training a random forest classifier on static code features extracted from npm packages. The results demonstrated that automated detection is achievable and that ML can outperform purely rule-based systems on the same feature set. The scope, though, is narrow: npm only, static features only, no behavioral signals, and no mechanism for catching novel attack patterns that don't resemble anything in the training data. It's a promising proof of concept that runs into walls when pushed outside its original conditions.

Vu et al. [6] focused specifically on typosquatting in the PyPI ecosystem, using name similarity metrics and NLP techniques to flag packages whose names were suspiciously close to popular legitimate ones. The approach works well within its scope—typosquatting is a real and common attack vector, and the detection rates are solid. The problem is that typosquatting is just one of many attack techniques, and a system built around name similarity has nothing to say about a backdoored update from an account with a perfectly legitimate name and a long publishing history.

Ferreira et al. [7] applied a broader set of classifiers—support vector machines, random forests, and naive Bayes—to npm package data, demonstrating that standard ML pipelines can distinguish malicious packages from benign ones with meaningful accuracy. The work is useful as a benchmark, but it operates on a limited feature set, produces only binary classifications without explanations, and makes no attempt to model dependency relationships or behavioral signals. It shows that ML can work here without getting very far into the question of how to make it work well.

Wermke et al. [8] took a different angle entirely, examining how trust is established and maintained within open-source development communities through qualitative interviews with maintainers and contributors. The findings are genuinely illuminating—trust turns out to be highly informal, often personal, and rarely backed by systematic security practices—but the contribution is sociological

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

rather than technical. No automated detection mechanism emerges from the analysis, and the work is better understood as context for why the technical problem is harder than it looks than as a step toward solving it.

He et al. [9] represent the closest prior work to the system proposed in this paper. Their MalOSS framework combines multiple ML signals and includes a publicly released dataset covering both PyPI and npm, with partial explainability built into the classification pipeline. The results are strong within the scope of the evaluation, and the dataset has become a useful resource for the research community. What the framework lacks is graph neural network-based dependency modeling and any form of real-time CI/CD integration—two components that are central to the architecture proposed here—which limits its applicability in live production environments.

Kipf and Welling [10] introduced graph convolutional networks as a mechanism for semi-supervised classification on graph-structured data, demonstrating the approach on citation networks and other benchmark datasets. The technique is directly relevant to the package dependency graph problem: packages can be represented as nodes, dependencies as edges, and GCN message passing can propagate signals about known malicious packages to their neighbors. The work is domain-general, however, and the gap between citation network classification and supply chain security is significant—package metadata integration, version constraint modeling, and ecosystem-specific graph construction all require adaptation that the original paper doesn't address.

Hamilton et al. [11] extended the GNN paradigm with GraphSAGE, an inductive approach that learns to generate embeddings for nodes not seen during training by sampling and aggregating neighborhood features. This property is particularly valuable for package ecosystem modeling, where new packages are continuously added and a system that can only classify known nodes is of limited use. Like Kipf and Welling, though, the work is domain-general, and adapting it to the specific structure and dynamics of package dependency graphs requires meaningful engineering effort that goes beyond what the paper provides.

Lundberg and Lee [12] introduced SHAP—SHapley Additive exPlanations—as a unified framework for interpreting the predictions of any machine learning model. The approach grounds explainability in game theory, producing per-feature attribution scores that are both theoretically principled and practically interpretable. For supply chain security specifically, this kind of explanation is critical: a detection system that flags a package as malicious without saying why gives a security analyst almost nothing to act on. SHAP doesn't constitute a detection system on its own, but it provides the explainability layer that a detection system needs.

Chen and Guestrin [13] developed XGBoost, the gradient boosting framework that has become a standard tool for tabular classification problems. Its combination of strong performance on structured data, computational efficiency, and built-in feature importance makes it a natural choice for the metadata and code feature classification components of a supply chain detection system. The framework is general-purpose and doesn't bring any domain-specific knowledge about packages or ecosystems, but as a classification engine it is well-suited to the problem and widely validated across a range of high-stakes domains.

Vaswani et al. [14] introduced the Transformer architecture, which has since become the foundation for virtually all state-of-the-art NLP systems. In the context of malicious package detection, Transformers are relevant primarily through code-specific fine-tuned variants that can model the semantic content of source code—variable names, API call sequences, control flow patterns—in ways that earlier approaches based on bag-of-words or fixed-size embeddings cannot. The compute requirements are substantial and the original paper is not adapted to code security tasks, but the architectural contribution is directly applicable to the code analysis component of the proposed framework.

Chawla et al. [15] addressed the class imbalance problem with SMOTE, a technique that generates synthetic minority-class examples by interpolating between existing ones. In the package security context, malicious packages represent a tiny fraction of all published packages, and classifiers trained on raw data tend to achieve high accuracy by simply predicting benign for everything. SMOTE provides one practical mitigation, and while it is a preprocessing technique rather than a detection method, it is an important ingredient in making any ML-based detection system viable in this highly imbalanced setting.

Taken together, this body of work makes clear that the individual pieces of a robust detection system already exist. What remains underdeveloped is the integration—a unified architecture that combines metadata, code, behavioral, and graph signals into a single pipeline with real-time capability, zero-day generalization, and interpretable outputs. That integration is the contribution of this paper.

THE SOFTWARE SUPPLY CHAIN THREAT LANDSCAPE

A. Scope and Attack Surface

A software supply chain encompasses every step between writing code and running it in production—including the third-party libraries and tools that get woven into the fabric of an application along the way. Each of those steps is a potential point of

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

compromise, and package repositories sit at the center of the most exposed part of that chain. They're trusted infrastructure, which means that anything they distribute gets treated as safe almost by default.

The scale makes the risk hard to overstate. PyPI alone hosts more than 500,000 packages and handles roughly four billion downloads every month. npm exceeds two million published packages. Maven Central underpins a vast portion of the Java ecosystem. A single compromised package in any of these repositories doesn't just affect one project—it affects every downstream consumer, often silently and automatically.

B. Taxonomy of Attack Vectors

Typosquatting is one of the oldest tricks in the supply chain playbook. Attackers register packages with names that are one keystroke away from something popular—'request' instead of 'requests', 'colourama' instead of 'colorama'—and wait for developers to make a typo or trust autocomplete without checking. It requires almost no technical sophistication, and it keeps working because humans make mistakes.

Dependency confusion, documented by Alex Birsan in 2021, is more technically clever. It exploits the way package managers decide which version of a package to fetch when a name exists in both a public registry and a private internal one. By publishing a package with the right name and a higher version number to the public registry, an attacker can cause build systems across dozens of organizations to silently pull and execute their code instead of the intended internal package. Birsan demonstrated this worked against Apple, Microsoft, and many others.

Backdoored updates are arguably the most dangerous vector because they weaponize an existing trust relationship. A legitimate, well-maintained package gets compromised—through a stolen maintainer credential, a phishing attack, or a social engineering play—and the malicious payload is slipped into what looks like a routine version bump. Users who have auto-updates enabled never even see the decision being made.

Trojanized packages are malicious from the start, but they're often not obviously so. They may provide genuine functionality while executing a hidden payload in parallel. The more sophisticated versions are environment-aware—they check for signs of sandbox analysis and suppress their malicious behavior when they detect it, activating only in what looks like a real production environment.

Protestware is a newer and ethically murkier category. Some maintainers have deliberately introduced destructive behavior into their own packages as a form of protest—against corporate users, geopolitical actors, or perceived exploitation of open-source labor. The intent may not be criminal, but the effect on downstream consumers can be identical to a traditional attack.

Compromised maintainer accounts tie many of these vectors together. An attacker who gains control of a trusted publisher identity can push any payload they want under a name that package managers and developers have long since learned to trust. It's the cleanest form of supply chain attack because it doesn't require registering anything new or tricking anyone into a typo—it just exploits the credentials of someone who already has the keys.

REVIEW OF EXISTING DETECTION METHODOLOGIES

Before machine learning entered the picture, the security community developed a set of approaches for catching malicious packages. They each have genuine strengths, and they each have limits that become obvious at scale or against a determined adversary.

A. Static Analysis

Static analysis looks at what a package's code says without actually running it. Analysts—or automated tools standing in for them—scan for the kinds of things malware tends to do: hardcoded IP addresses, suspicious import chains, sensitive API calls like os.system or socket.connect, base64-encoded strings that could be hiding a payload. It's fast, it's safe, and it doesn't require executing untrusted code. The problem is that any attacker who knows what static tools look for can write code that looks clean. Obfuscation is trivial. Splitting a suspicious call across multiple innocent-looking functions is trivial. Static analysis has never been able to keep up with adversaries who know it's watching.

B. Dynamic Analysis

Dynamic analysis takes the opposite approach: run the package in a sandboxed environment and watch what it does. File writes, outbound network connections, process creation, registry modifications—these behavioral signals are much harder to fake than static code appearances. The tradeoff is cost. Standing up and tearing down isolated environments for every package submitted to a major registry is expensive, and it slows everything down. Worse, sophisticated malware has learned to detect sandbox environments and sit quietly until it's confident it's in a real system before activating.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

C. Signature-Based Detection

Signature-based detection is the approach that antivirus software has used for decades. Maintain a database of known-bad fingerprints—hashes, byte patterns, code snippets—and flag anything that matches. It's fast, it's precise for known threats, and it generates very few false positives. But it is entirely reactive. A package that has never been seen before has no signature. Zero-day threats, by definition, pass through signature-based systems without triggering anything.

D. Heuristic and Rule-Based Analysis

Heuristics try to catch suspicious behavior by applying expert-crafted rules: flag any package whose author account is less than a week old, any package with no README, any package that registers an install hook in setup.py, any package whose name is suspiciously close to a top-downloaded library. The rules are cheap to run and easy to explain. They also generate a lot of noise, because the characteristics that correlate with malicious packages also appear in plenty of legitimate ones. Maintaining and tuning the rule set as attack patterns evolve is a continuous manual effort.

E. Behavioral Analysis

Behavioral analysis is dynamic analysis with the focus narrowed to system-level interactions: which system calls a package makes, what network traffic it generates, which files it touches and in what order. This produces a richer fingerprint than a simple sandbox observation and is genuinely difficult for malware to fake without changing what it actually does. The cost is similar to dynamic analysis—resource intensive, slow—and the same sandbox evasion techniques apply.

F. Limitations of Existing Approaches

None of these approaches, individually or combined, has solved the problem. False positive rates remain high enough that security teams spend significant time chasing alerts that go nowhere, which erodes confidence in the systems and creates space for real threats to slip through. The training data available for ML-based variants skews heavily toward known attack patterns, which means novel techniques tend to go undetected until someone notices the damage. And across the board, existing tools offer little in the way of explanation—they may flag something as suspicious, but they rarely give an analyst enough information to understand why, prioritize their response, or push back if the flag seems wrong.

MACHINE LEARNING-BASED APPROACHES IN THE LITERATURE

Machine learning has brought a different set of tools to the package detection problem. Rather than encoding expert rules, ML systems learn from examples—and in principle, they can learn to recognize patterns that no human analyst would have thought to write a rule for. The approaches differ mainly in what features they use as input.

A. Metadata-Based Classification

Metadata is the cheapest signal available. Account age, publishing history, description content, version increment size, download velocity—all of this is available through public registry APIs without touching a single line of code. Classifiers trained on metadata features have shown meaningful detection rates for typosquatting and newly registered malicious accounts, and they can run at the scale repositories actually need. The vulnerability is that metadata is easy to manipulate. An attacker who knows that young accounts get flagged can age their account first. One who knows sparse descriptions raise red flags can write something that looks reasonable. Metadata alone isn't enough.

B. NLP-Based Code Analysis

Treating source code as text opens up a powerful set of tools. Tokenization, abstract syntax tree parsing, and embedding models can extract semantic features from code—what APIs get called, in what order, with what arguments, in what context. Transformer-based models fine-tuned on code corpora have shown particular promise, learning representations that capture meaning rather than just syntax. The weakness is obfuscation: an attacker who scrambles variable names, inserts meaningless operations, or encodes strings can substantially degrade NLP-based detectors that rely on semantic coherence.

C. Dependency Graph Modeling

A package doesn't exist in isolation—it sits in a web of dependencies, each of which has its own history and reputation. Graph neural networks are well suited to modeling this structure. By representing packages as nodes and dependency relationships as edges, GNNs can learn what a healthy dependency graph looks like and flag deviations: unusual depth, cyclic dependencies, connections to packages with suspicious histories. Crucially, GNNs can propagate signals through the graph—if a known malicious package shows up in a dependency tree, that's relevant information about everything that depends on it, even before an explicit connection is confirmed.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

D. Deep Learning on Raw Code and Binaries

CNNs, RNNs, and LSTM architectures have all been applied to raw code sequences and binary representations, learning patterns directly from the bytes rather than from hand-engineered features. These models can in principle catch things that structured feature extraction would miss—subtle patterns that don't map cleanly to any named API or documented technique. In practice, they need large labeled datasets to train effectively, are computationally expensive to run at scale, and produce outputs that are hard to interpret. For a security context where analysts need to understand and act on detections, that opacity is a significant liability.

GAP ANALYSIS AND RESEARCH MOTIVATION

Reading across the existing literature, four gaps stand out as the most consequential—and the most tractable to address through careful system design.

A. Over-Reliance on Known Signatures

Most ML-based detection systems, even those that don't use explicit signature databases, are trained and evaluated on historical data that is biased toward known attack patterns. When a genuinely novel technique appears—one that doesn't resemble anything in the training set—detection rates fall sharply. The goal should be a system that reasons from principles rather than memorized examples: one that flags suspicious metadata even if it doesn't match a profile it's seen before, and that treats unusual code structure as a signal even without a known-bad template to compare it to.

B. Absence of Real-Time Pipeline Integration

Most research prototypes treat package analysis as an offline batch process. A package gets submitted, reviewed sometime later, and a verdict is returned. That model doesn't fit how modern development actually works. In a CI/CD pipeline, dependencies are resolved and installed in seconds—often automatically, without a human in the loop. A detection system that can't return a result before installation completes isn't preventing anything; it's generating incident reports after the fact.

C. Fragmented Feature Engineering

The field has produced strong individual approaches—good metadata classifiers, good NLP-based code analyzers, good graph models—but very little work on combining them. That fragmentation is a missed opportunity. A package can look clean on metadata while containing obfuscated malware in its code. A package can have innocuous-looking code while sitting in a suspicious dependency chain. Each signal modality sees a different slice of the threat, and a system that only looks at one of them will always have blind spots.

D. Poor Zero-Day Detection Performance

Newly published packages are the highest-risk window in a repository's lifecycle. They have no download history, no community review, no behavioral baseline. Systems that rely on those signals to make their decisions have almost nothing to work with for packages that have just appeared. Zero-day packages are precisely the ones that most need scrutiny, and they're the ones current approaches handle worst.

PROBLEM STATEMENT AND RESEARCH OBJECTIVES

The detection problem can be stated precisely: given a package P characterized by a metadata record M, a source code corpus C, and a position in a dependency graph G, build a classifier $f(M, C, G) \rightarrow \{\text{malicious, benign}\}$ that is accurate, fast enough for real-time repository integration, robust to attack patterns it hasn't seen before, and able to produce explanations that a human analyst can evaluate and act on.

Each of those requirements adds meaningful constraint. Real-time operation sets hard latency bounds on every stage of feature extraction and inference—anything that can't return a result in time is no different from no system at all. Generalization to novel attacks means the system can't simply memorize known-bad patterns; it has to learn something about what malicious packages look like structurally, not just what specific malicious packages have looked like historically. And explainability isn't optional in a security context—a black-box verdict that a package is malicious gives an analyst no way to verify the decision, prioritize their response, or identify when the system is wrong.

The objectives of this paper follow directly from that framing. We aim to identify the most informative features across all relevant modalities, propose an architecture that combines them effectively, justify the model choices for each component, and define evaluation protocols that reflect the class imbalance and adversarial conditions of real deployment.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

PROPOSED SYSTEM ARCHITECTURE

The proposed framework consists of five modules that form a pipeline from raw package ingestion to a final security verdict. Each module has a clearly defined responsibility and can be scaled independently.

A. Data Collection Module

The pipeline starts with continuous ingestion from the APIs of PyPI, npm, and Maven Central. For every newly published or updated package, the module pulls the metadata record, the source code archive, the dependency manifest, and the full version history. Streaming ingestion is preferable to batch processing here—latency between publication and analysis is directly proportional to the window of exposure, and every second matters when a malicious package is live on a registry being actively downloaded.

B. Feature Extraction Engine

Raw package data flows into four parallel extraction pipelines, each targeting a different dimension of the detection problem.

The metadata pipeline turns the package record into a numerical feature vector: how old is the author's account, how many packages have they published before, how large is the version increment, how does the package name compare to the most popular packages in the ecosystem, how fast is the download count growing relative to comparable packages? These signals are cheap to compute and surprisingly informative, particularly for the kinds of opportunistic attacks that dominate the lower end of the sophistication spectrum.

The code analysis pipeline applies NLP techniques to the source. Abstract syntax tree parsing identifies the API calls being made and the control flow structures around them. Embedding models—trained on code rather than natural language—encode semantic content into dense vectors. Particular attention goes to install-time execution hooks, because anything that runs at install time runs with a level of trust that the developer almost certainly never thought about explicitly.

The behavioral pipeline, where time permits, executes the package in a lightweight sandbox and records what it does: which files it touches, which network addresses it contacts, what child processes it spawns. These are the signals that are hardest to fake without changing the package's actual behavior. Even partial behavioral telemetry adds significant discriminative power.

The dependency graph pipeline builds a local representation of the package's position in the broader ecosystem graph. GNN message passing layers then compute embeddings that incorporate information from the package's neighbors—who depends on it, what it depends on, and what those neighbors look like—rather than treating the package as an isolated artifact.

C. ML Classification Engine

The classification engine runs one specialized model per feature modality. XGBoost or LightGBM handles the tabular metadata and code features, where gradient boosting has a strong track record and trains efficiently. A GCN or GraphSAGE architecture handles the dependency graph embeddings. A Transformer or LSTM handles the sequential code embedding features. Each model produces a probability score for its modality, and a logistic regression meta-learner combines those scores into a single maliciousness probability—dynamically weighting each modality based on the confidence of the individual classifiers and the characteristics of the package being evaluated.

D. Risk Scoring Module

A probability score alone isn't actionable. The risk scoring module converts the classifier output into something an analyst can actually work with, using SHAP values to attribute the prediction back to specific features. Instead of a number, the output is a ranked list of reasons: this package is flagged primarily because it established an outbound connection to an external IP during install, its author account is four days old, and its setup.py contains what appears to be an encoded string. That kind of explanation is what separates a detection system that analysts trust from one that generates noise.

E. Alert and Reporting System

High-confidence detections trigger blocking signals sent to package manager clients through repository-level hooks, and notifications go to both the package maintainer and repository administrators. Medium-confidence cases go into a quarantine queue for expedited human review rather than being automatically blocked or allowed. Integration with CI/CD tools—GitHub Actions, Jenkins, and similar systems—means the decision can be made before a build completes, not after a compromised package is already running in production.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

FEATURE ENGINEERING AND MODEL SELECTION

A. Feature Categories and Rationale

The metadata features are grounded in observed attacker behavior. Malicious accounts tend to be new—attackers often create them specifically for a campaign rather than aging them over time. Malicious packages tend to have thin documentation, because the goal is rapid deployment rather than maintainability. Version numbers are sometimes chosen to match or exceed a legitimate package's latest release, to ensure the package manager resolves the attacker's version over the intended one. None of these signals is individually decisive, but in combination they shift the probability significantly.

The code features target the mechanisms malware actually uses to cause harm. Network communication, file system writes, and process execution cover the majority of malicious behaviors across all known attack categories. Obfuscated code—base64-encoded payloads, hexadecimal escape sequences, dynamically constructed import paths—is a reliable indicator that something in the package is being hidden from static inspection. Entropy analysis of string literals catches encoding patterns that pattern matching alone might miss.

The behavioral features provide the most direct evidence of malicious intent, precisely because they reflect what the package actually does rather than what it appears to do. A package that phones home during install, writes files to system directories it has no legitimate reason to touch, or spawns processes with no documented purpose has demonstrated malicious behavior in a way that is difficult to argue with.

The dependency graph features encode context that no per-package analysis can capture. A new package that immediately attracts a large number of dependents—or that depends on a small cluster of equally new packages with no track record—is exhibiting structural patterns that don't fit the normal growth trajectory of legitimate libraries. Trust propagation through the graph, similar in spirit to PageRank, allows the system to assign preliminary credibility scores based on a package's position in the network before its own history is long enough to be informative.

B. Handling Class Imbalance

Malicious packages make up a tiny fraction of everything published to a major registry. A classifier that predicts benign for every submission would be right more than 99% of the time—and completely useless. Addressing this requires a combination of approaches: SMOTE generates synthetic malicious examples to balance the training data; cost-sensitive learning assigns heavier penalties to false negatives than false positives, reflecting the asymmetric cost of missing a real threat; and ensemble resampling methods like BalancedBaggingClassifier resample the training distribution at each iteration to keep the model from drifting toward the majority class.

C. Evaluation Metrics

Accuracy is not a useful metric here. Precision and recall matter far more—specifically, how often a flagged package is actually malicious, and how many malicious packages the system catches. The F1 score balances them. ROC-AUC gives a threshold-independent picture of discriminative ability. And the false positive rate deserves explicit attention: a system that flags too many legitimate packages will be tuned down or turned off by the teams that have to manage the alerts, which makes every threshold a practical decision about operational sustainability, not just statistical performance. Evaluation should be done on temporally held-out data—training on historical packages and testing on more recent ones—to simulate what deployment actually looks like and avoid inflated results from data leakage.

DISCUSSION

A. Scalability Considerations

A system designed to monitor a repository the size of PyPI or npm needs to handle thousands of publication events per day without falling behind. The modular pipeline design makes this tractable: metadata and code analysis are stateless and straightforwardly parallelizable; behavioral sandboxing is resource-intensive but can be prioritized by risk tier; the dependency graph module requires access to the full ecosystem graph and benefits most from incremental update strategies that avoid recomputing embeddings from scratch with every new package. Each module can be scaled independently as a microservice, and the architecture doesn't require a single bottleneck that all traffic has to pass through.

B. Adversarial Robustness

Any public-facing detection system will eventually be reverse-engineered by the people it's trying to catch. Attackers who understand the features the system looks for can craft packages that look clean on every individual signal while still delivering their payload. The multi-modal design provides some natural resilience here—defeating a system that looks at metadata, code, behavior,

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 05 , May - 2026

and graph structure simultaneously requires fooling all of those components at once, which is substantially harder than defeating any one of them. Periodic retraining on recent data keeps the system from falling behind on evolving techniques, and adversarial training on perturbed examples can harden the most exploitable components. The explainability layer contributes here too: when analysts can see which features are driving detections, they can identify systematic evasion patterns and update the feature set before those patterns become widespread.

C. Ethical and Legal Considerations

Automated security systems that make blocking decisions about third-party code carry real obligations. A false positive that takes down a legitimate package, or flags a maintainer as a bad actor without justification, causes harm—to the open-source contributor, to the projects that depend on their work, and to the credibility of the detection system itself. Any deployment needs a clear appeals process, transparent criteria, and a human review layer for consequential decisions. The collection of usage telemetry and behavioral data also needs to be handled carefully, both for privacy reasons and because the data itself is sensitive infrastructure that could be misused if it were compromised.

CONCLUSION AND FUTURE WORK

This paper has covered a lot of ground - the threat landscape, the state of existing detection approaches, the gaps that still need to be filled, and a detailed design for a framework that addresses those gaps through multi-modal ML, real-time integration, and explainability. The argument throughout has been that the individual techniques needed to build a robust supply chain detection system already exist. What's been missing is a coherent design that brings them together.

The proposed framework does that: metadata analysis, NLP-based code examination, behavioral telemetry, and dependency graph modeling feed into an ensemble classification pipeline, with SHAP-based explanations making the output actionable and CI/CD hooks making the response fast enough to matter. It is not a finished system—no implementation is presented here—but it is a complete specification of what that system should look like and why each design choice was made.

The most important next step is empirical validation. The PyPIDB and MalOSS datasets provide a starting point, but they need to be supplemented with more recent threat data to reflect how attack techniques have evolved. Beyond that, the field needs evaluation benchmarks that allow honest comparison across research contributions—the equivalent of what GLUE did for NLP. Federated learning approaches could allow organizations to share threat intelligence without exposing sensitive data. And the framework needs to be extended to enterprise package registries, which present a different set of challenges because they lack the community review signals that public repositories provide. There is meaningful work ahead, and this paper is intended as a foundation to build on.

ACKNOWLEDGEMENTS

The authors would like to thank the open-source security research community for the foundational datasets and analyses that informed this survey. This work was supported in part by [Funding Agency Name, Grant Number].

REFERENCES

[1] Ohm, M., Plate, H., Sykosch, A., & Meier, M. (2020). Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. Proceedings of DIMVA, pp. 1–25.
[2] Ladisa, P., Plate, H., Martinez, M., & Barais, O. (2023). SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. 2023 IEEE Symposium on Security and Privacy (SP), pp. 1509–1526.
[3] Birsan, A. (2021). Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies. Medium Engineering Blog. https://medium.com/@alex.birsan
[4] Duan, R., Alrawi, O., Kasturi, R. P., Elder, R., Saltaformaggio, B., & Lee, W. (2020). Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. NDSS.
[5] Sejfia, A., & Schäfer, M. (2022). Practical Automated Detection of Malicious npm Packages. 44th ICSE, pp. 1680–1692.
[6] Vu, D. L., Pashchenko, I., Massacci, F., Plate, H., & Sabetta, A. (2021). Typosquatting and Combosquatting Attacks on the Python Ecosystem. IEEE EuroS&PW, pp. 509–519.
[7] Ferreira, R., Pereira, T., & Cruz, D. (2021). Detecting Malicious Packages in npm: A Machine Learning Approach. Proceedings of ARES.
[8] Wermke, D., Wöhler, N., Klemmer, J., Easton, M., Koishybayev, Y., & Kapravelos, A. (2022). Committed to Trust: A Qualitative Study on Security & Trust in Open Source Software Projects. IEEE S&P, pp. 1880–1896.
[9] He, Z., Zhang, J., Liu, R., Yu, J., & Li, X. (2023). MalOSS: A Collaborative Dataset and Framework for Evaluating Malicious Open Source Packages. ACM CCS 2023.
[10] Kipf, T. N., & Welling, M. (2017). Semi-Supervised Classification with Graph Convolutional Networks. ICLR.
[11] Hamilton, W. L., Ying, R., & Leskovec, J. (2017). Inductive Representation Learning on Large Graphs. NeurIPS, vol. 30.
[12] Lundberg, S. M., & Lee, S. I. (2017). A Unified Approach to Interpreting Model Predictions. NeurIPS, vol. 30, pp. 4765–4774.
[13] Chen, T., & Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System. ACM SIGKDD, pp. 785–794.
[14] Vaswani, A., Shazeer, N., Parmar, N., et al. (2017). Attention Is All You Need. NeurIPS, vol. 30.
[15] Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: Synthetic Minority Over-Sampling Technique. Journal of Artificial Intelligence Research, 16, 321–357.