COMP 461- Introduction to Cyber Security

“Supply Chain Attack”

Group - 4

Ali Rubar Kal - 042201040
Berkay Rigan - 042301120
Muhammed Hamza Taş - 042201055

Date: 04.01.2026
Instructor: Onur Erkan
Institution: MEF University

1. Introduction

- 1.1 Problem Definition
- 1.2 Motivation
- 1.3 Importance of Software Supply Chain Attacks
- 1.4 Brief Description of ChainGuard

2. Background and Related Work

- 2.1 Supply Chain Attack Types
- 2.2 Existing Solutions
- 2.3 Overview of Similar Systems

3. System Design and Methodology

- 3.1 Overall System Architecture
- 3.2 Static Analysis Approach (AST, Data-Flow)
- 3.3 Dynamic Analysis (Sandbox)
- 3.4 Threat Intelligence Integration
- 3.5 Scope and Limitations

4. Risk Scoring Model

- 4.1 Risk Categories
- 4.2 Weight Assignment
- 4.3 Scoring Strategy
- 4.4 Example Risk Calculation

5. Experimental Setup and Results

- 5.1 Test Environment
- 5.2 Analyzed Packages
- 5.3 Detection Results
- 5.4 Risk Score Outputs
- 5.5 Case Studies

6. Comparison with Existing Work

- 6.1 ChainGuard vs pip-audit
- 6.2 Strengths and Weaknesses
- 6.3 Practical Usability Comparison

1. Introduction

1.1 Problem Definition

Software supply chain attacks have emerged as one of the most critical security threats in modern software development. Instead of directly targeting end systems, attackers compromise trusted third-party dependencies or open-source libraries, allowing malicious code to propagate across thousands of downstream projects. The increasing reliance on third-party packages significantly expands the attack surface of software ecosystems.

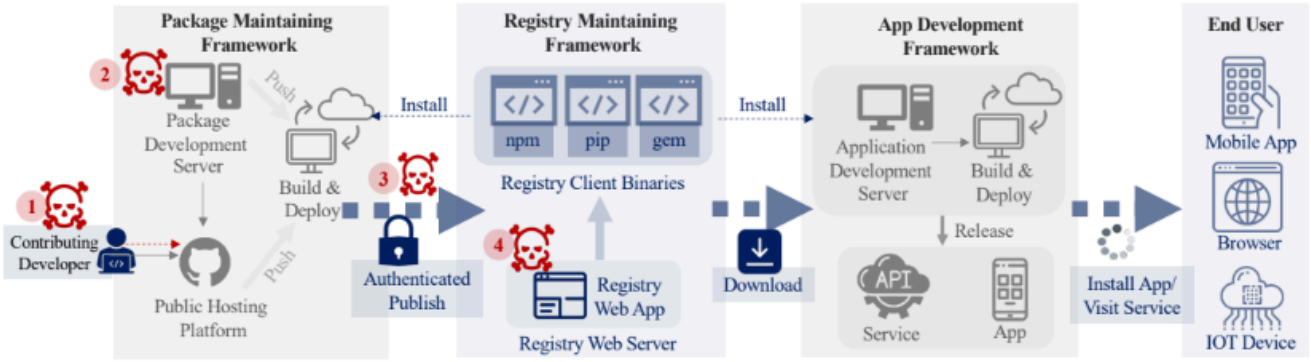


Figure 1. Overview of a software supply chain attack, showing how a compromised package can propagate from development and registry frameworks to end-user applications.

1.2 Motivation

Existing defense mechanisms primarily rely on reactive approaches such as community reporting or signature-based detection. These methods often fail to detect newly introduced or obfuscated malicious packages in a timely manner. Furthermore, most available tools focus either on large-scale ecosystem analysis or known vulnerabilities, leaving developers without practical tools to assess the real security risk of their own project dependencies.

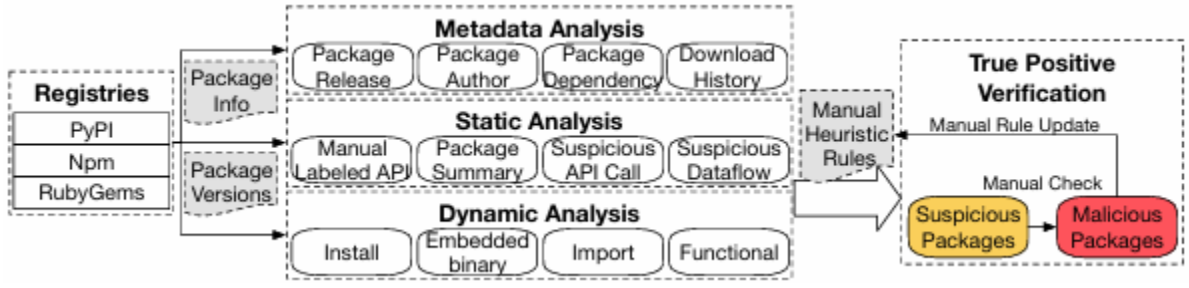


Figure 2. Multi-stage analysis pipeline used in existing software supply chain security systems.

1.3 Importance of Software Supply Chain Attacks

Recent studies show a rapid increase in malicious packages across major package registries. High-impact incidents demonstrate that a single compromised dependency can affect millions of users through transitive dependencies. These attacks threaten not only individual applications but also critical infrastructure, enterprises, and governmental systems.

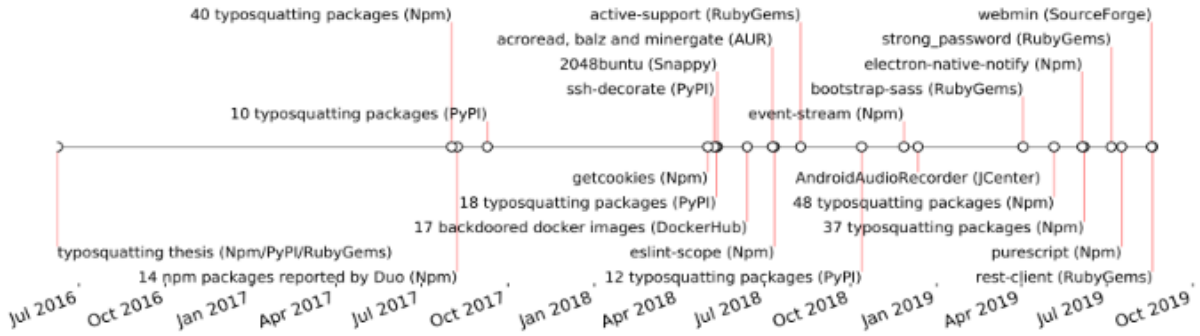


Figure 3. Timeline of reported typosquatting attacks across major package ecosystems.

1.4 Brief Description of ChainGuard

ChainGuard is a project-level security analysis tool designed to detect software supply chain risks within Python projects. It performs multi-layer analysis including threat intelligence checks, static code analysis using AST-based data-flow tracking, dynamic sandbox execution, and credential leakage detection. Each dependency is evaluated through a weighted risk scoring model, providing developers with an interpretable security risk score between 0 and 100.

2. Background and Related Work

2.1 Supply Chain Attack Types

- Typosquatting
- Dependency Confusion
- Backdoor Injection
- Malicious Post-Install Scripts

These attack vectors exploit trust relationships within software ecosystems and often remain undetected for long periods.

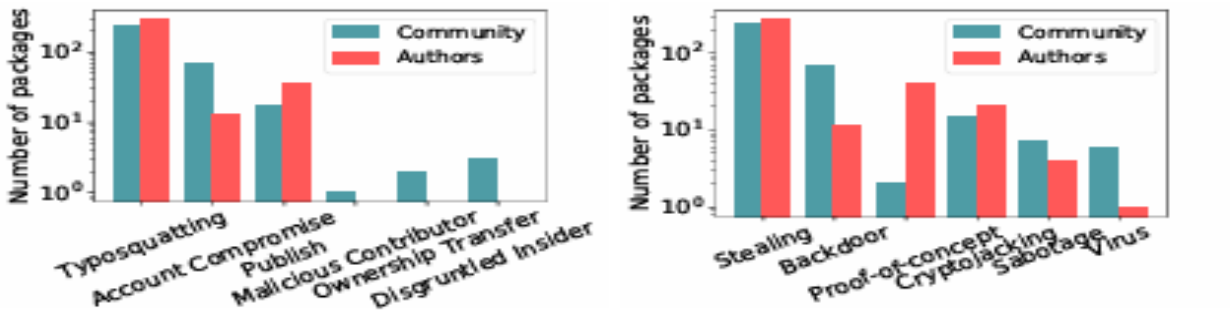
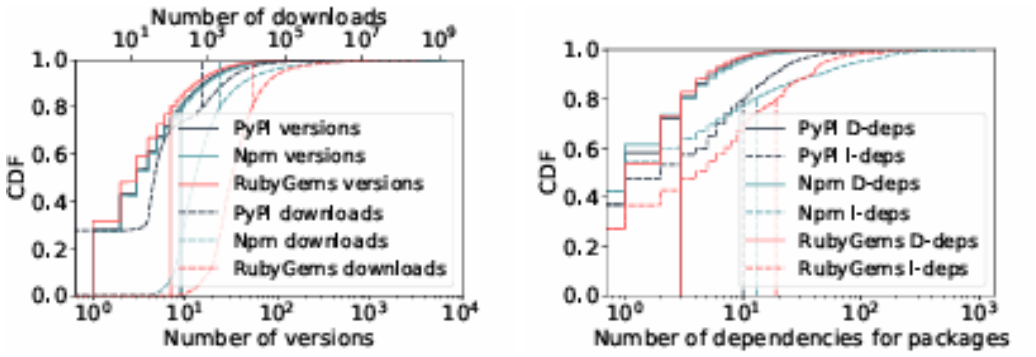


Figure 4. Distribution of software supply chain attack types reported by community members and package authors.

2.2 Existing Solutions

Existing solutions include vulnerability scanners, malware signature-based tools, and large-scale ecosystem monitoring systems. While effective in certain scenarios, most tools either focus on known vulnerabilities or require extensive infrastructure, limiting their applicability for individual developers.



(a)Distribution of the number of versions and downloads per package in each registry.

(b) Distribution of dependency count for top 10K downloaded packages in each registry.

2.3 Overview of Similar Systems

Existing software supply chain security tools can be broadly divided into ecosystem-scale analysis systems and project-level vulnerability scanners. Ecosystem-scale approaches, such as MALOSS, aim to detect malicious packages across entire registries, but they are primarily designed for researchers and registry maintainers rather than individual developers. In contrast, project-level tools such as pip-audit focus on identifying known vulnerabilities (CVEs) within a project's dependencies by querying curated security advisory databases such as OSV and PyPI.

While pip-audit provides reliable detection of documented and publicly disclosed vulnerabilities, its scope is limited to known issues and does not account for runtime behavior, code obfuscation, or emerging supply chain attack patterns. ChainGuard complements this approach by extending dependency analysis beyond CVE-based scanning through the integration of static code inspection, dynamic sandbox execution, and interpretable risk scoring. This enables ChainGuard to detect both known vulnerabilities and previously undocumented supply chain risks within individual software projects.

	PyPI	Npm	RubyGems
# of Packages	186,785	997,561	151,783
# of Package Versions	809,258	4,388,368	629,116
# of Package Maintainers†	67,552	284,009	51,505
# of Reported Malware	67	230	15
# of New Malware	7	41	291

Table 1. Comparison of package ecosystem statistics across PyPI, npm, and RubyGems.

3. System Design and Methodology

3.1 Overall System Architecture

ChainGuard follows a layered security architecture consisting of threat intelligence analysis, static analysis, dynamic sandbox execution, and risk aggregation.

3.2 Static Analysis Approach (AST, Data-Flow)

ChainGuard employs Abstract Syntax Tree (AST) parsing to analyze Python source code without execution. Data-flow (taint) analysis is applied to detect dangerous flows between untrusted inputs and sensitive sinks such as eval, exec, file writes, and subprocess execution.

3.3 Dynamic Analysis (Sandbox)

Suspicious packages are executed inside a hardened Docker sandbox, where runtime behaviors such as network connections, process spawning, and abnormal resource usage are monitored.

3.4 Threat Intelligence Integration

ChainGuard integrates multiple threat intelligence sources including PyPI metadata, OSV, GitHub Security Advisories, and community-maintained malicious package lists.

3.5 Scope and Limitations

ChainGuard focuses exclusively on Python dependencies and project-level analysis. Ecosystem-wide scanning, binary analysis, and machine learning-based detection are outside the scope of this work.

4. Risk Scoring Model

4.1 Risk Categories

- Threat Intelligence
- Static Data-Flow Risks
- Known Vulnerabilities
- Credential Leakage
- Obfuscation
- Dynamic Behavior

4.2 Weight Assignment

Each category contributes to the final score based on its relative security impact. Scores are normalized to produce a final risk value between 0 and 100.

4.3 Scoring Strategy

The scoring strategy prioritizes explainability and interpretability, allowing developers to understand why a package is considered risky.

4.4 Example Risk Calculation

ChainGuard computes the final risk score using a four-stage weighted analysis model, where each stage contributes a fixed maximum score, resulting in a total of 100 points. The stages include Threat Intelligence (25 points), Static Analysis (50 points), Setup Behavior Analysis (5 points), and Dynamic Analysis (20 points).

For example, if a package is flagged by threat intelligence sources due to known vulnerabilities (+20/25), exhibits dangerous data-flow patterns during static analysis (+30/50), triggers suspicious setup behavior (+5/5), and shows no abnormal runtime behavior during sandbox execution (+0/20), the final ChainGuard risk score is calculated as **55/100**, indicating a high-risk dependency.

5. Experimental Setup and Results

5.1 Test Environment

The experimental evaluation of ChainGuard was conducted on a local Windows-based development environment. The tool was executed on a standalone Python project containing a single source file with multiple third-party dependencies. ChainGuard automatically initialized its analysis pipeline, created an isolated temporary sandbox directory, and performed dependency discovery by statically parsing Python import statements.

Dynamic analysis was performed using a process-level sandbox due to the unavailability of Docker on the host system. Despite this limitation, runtime behaviors such as CPU usage, file access, and suspicious execution patterns were successfully monitored.

5.2 Analyzed Packages

The analyzed project contained 20 unique Python packages identified through static import analysis. Out of these, 11 packages were successfully downloaded and analyzed in depth, while the remaining packages were either part of the Python standard library or could not be resolved through PyPI.

The analyzed dependencies included a mix of widely used libraries (e.g., django, flask, numpy, requests) as well as cryptographic, parsing, and utility packages. This diverse dependency set enabled the evaluation of ChainGuard across multiple attack surfaces.

5.3 Detection Results

ChainGuard detected a total of 5 malicious packages and 1 suspicious package within the analyzed dependency set. Malicious classifications were primarily driven by threat intelligence findings such as OSV-reported vulnerabilities, yanked PyPI releases, and known security advisories.

Several packages triggered multiple security indicators simultaneously, including known CVEs, suspicious domains, credential leakage patterns, and abnormal runtime behavior. For example, high-severity vulnerabilities were detected in cryptographic and web framework libraries, while some packages exhibited excessive CPU usage during sandbox execution, indicating potentially malicious behavior.

To provide a baseline comparison, the same project was also analyzed using pip-audit. ChainGuard identified a total of 5 malicious and 1 suspicious packages out of 24 dependencies by combining threat intelligence, static code inspection, and dynamic sandbox execution. In contrast, pip-audit reported known vulnerabilities in only a single dependency (pycrypto), corresponding to two publicly disclosed security

advisories (PYSEC-2017-94 and PYSEC-2018-97). No additional packages were flagged by pip-audit, as it exclusively relies on CVE-based vulnerability databases and does not evaluate behavioral or structural indicators of supply chain attacks.

Tool	Total Packages	Malicious	Suspicious	Detection Basis
ChainGuard	24	5	1	CVE + Static + Dynamic + TI
pip-audit	24	1	0	CVE / OSV only

Table 2 summarizes the detection outcomes of ChainGuard and pip-audit on the same dependency set. While pip-audit identified known vulnerabilities in a single package based on CVE matching, ChainGuard flagged additional dependencies by leveraging behavioral, structural, and threat intelligence indicators beyond vulnerability databases.

5.4 Risk Score Outputs

Each analyzed package was assigned a normalized risk score between 0 and 100, computed using ChainGuard's four-stage weighted scoring model. The results demonstrated a wide distribution of risk levels across dependencies.

Low-risk packages such as six and crypto received scores below 5/100, indicating minimal security concerns. Medium-risk packages, including chardet, jwt, and lxml, accumulated scores between 25/100 and 30/100 due to obfuscation patterns and suspicious metadata. High-risk packages such as flask, django, and pycrypto exceeded 35/100, primarily due to known vulnerabilities, yanked releases, and critical security advisories.

These results indicate that ChainGuard effectively differentiates between benign, suspicious, and high-risk dependencies using interpretable risk scores.

5.5 Case Studies

Case Study 1 – High-Risk Dependency (pycrypto):
The pycrypto package triggered multiple high and critical security findings, including buffer overflow vulnerabilities and weak cryptographic key generation issues. Additionally, suspicious setup behavior and external domain references were detected. As a result, ChainGuard assigned a final risk score of 42/100, classifying the package as high risk.

Case Study 2 – Vulnerable Framework (django):
The django package was flagged by threat intelligence sources due to multiple OSV-reported vulnerabilities, including SQL injection and open redirect issues, as well as several yanked releases on PyPI. These combined findings resulted in a high-risk classification.

Case Study 3 – Low-Risk Utility Package (six):
In contrast, the six packages only exhibited minor code obfuscation indicators without any known vulnerabilities or suspicious runtime behavior. Consequently, it received a low ChainGuard risk score of 4/100, demonstrating the tool's ability to avoid overestimating benign dependencies.

6. Comparison with Existing Work

In this chapter, ChainGuard is compared with pip-audit, a widely adopted open-source vulnerability scanning tool for Python dependencies. Unlike ecosystem-scale research systems, pip-audit is commonly used in both academic studies and industrial practice as a baseline vulnerability assessment tool.
The comparison focuses on three aspects: analysis scope, detection capability, and practical usability.

6.1 ChainGuard vs pip-audit

pip-audit is a command-line–based security tool that scans Python dependencies against publicly available vulnerability databases such as OSV and PyPI Security Advisories. Its primary goal is to identify known vulnerabilities (CVEs) in installed packages without performing any form of code execution or behavioral analysis.

ChainGuard, in contrast, adopts a multi-layer supply chain risk analysis approach. While it also integrates OSV and PyPI advisory data similar to pip-audit, it extends beyond vulnerability scanning by incorporating static code analysis, setup behavior inspection, and dynamic sandbox execution. Additionally, ChainGuard provides an interpretable risk scoring model and a graphical user interface, supporting interactive investigation.

In terms of analysis depth, pip-audit focuses exclusively on known and previously disclosed vulnerabilities, whereas ChainGuard evaluates both known risks and previously unseen or non-CVE-based supply chain threats, such as obfuscated code, suspicious runtime behavior, and credential leakage.

From a technological perspective, pip-audit relies on database lookups and version matching, making it lightweight and deterministic. ChainGuard leverages AST-based data-flow analysis, sandboxed execution, and LLM-assisted interpretation to provide contextualized security insights.

In the experimental evaluation, ChainGuard and pip-audit were applied to the same dependency set consisting of 24 packages. While pip-audit successfully detected known vulnerabilities in pycrypto by matching package versions against OSV advisories, it did not report any issues for the remaining dependencies. ChainGuard, on the other hand, flagged additional packages based on non-CVE indicators such as excessive code obfuscation, suspicious domain references, yanked releases, and abnormal runtime behavior observed during sandbox execution. This comparison demonstrates that ChainGuard extends beyond traditional vulnerability scanning by addressing early-stage and undocumented software supply chain risks.

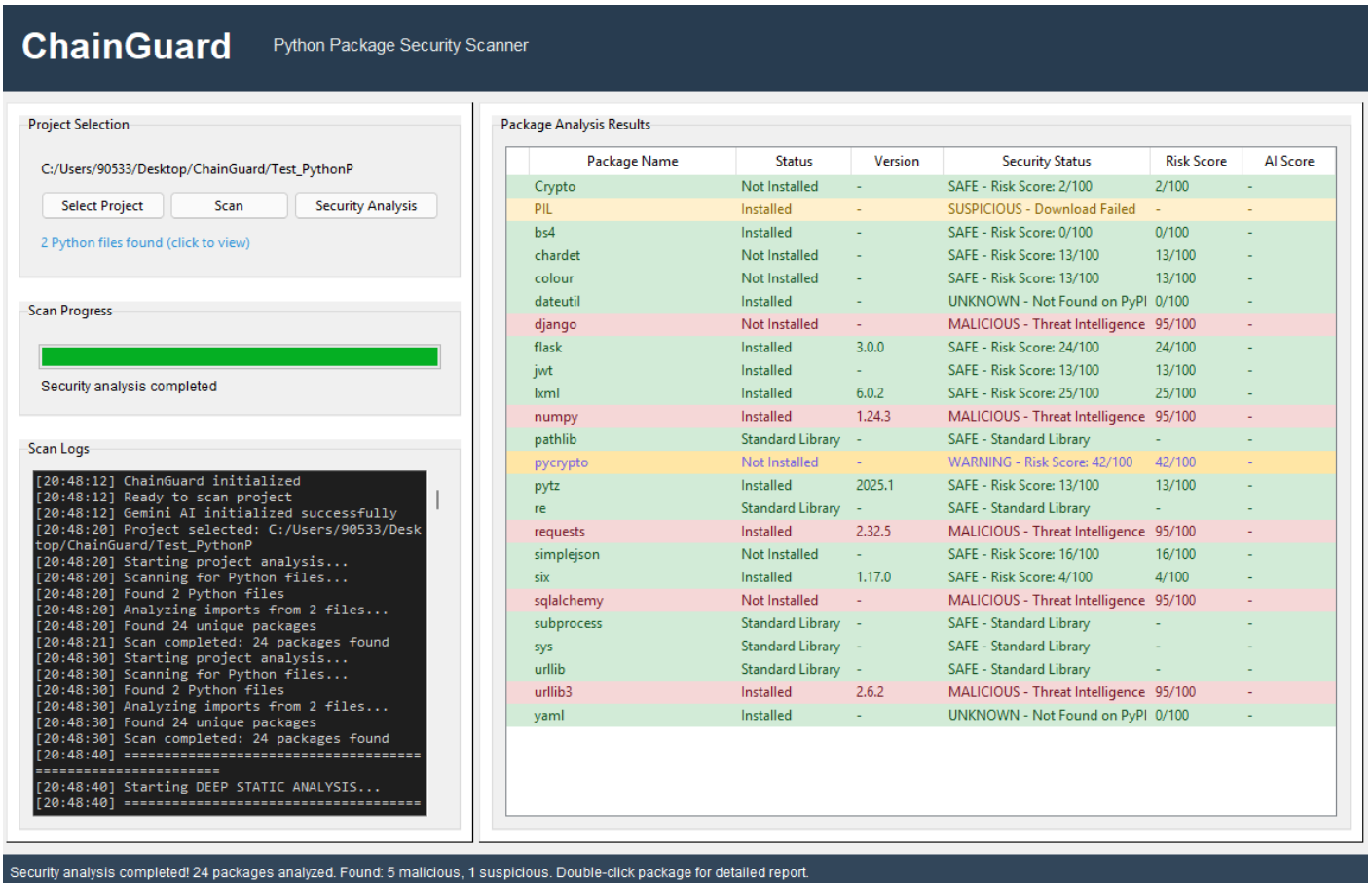


Figure 5. Graphical user interface of ChainGuard showing dependency risk scores and security classification results.

6.2 Strengths and Weaknesses

Strengths of ChainGuard

- Beyond CVE Detection: Unlike pip-audit, ChainGuard detects supply chain risks that are not yet documented in vulnerability databases, including runtime anomalies, obfuscation patterns, and suspicious installation behavior.
- Dynamic Analysis Capability: ChainGuard executes packages in a controlled sandbox environment and monitors CPU usage, file access, and network activity, enabling detection of behaviors such as crypto mining or covert data exfiltration.
- Explainable Risk Scoring: Instead of binary vulnerable/non-vulnerable results, ChainGuard produces a weighted risk score (0–100), allowing developers to understand the severity and origin of detected issues.
- User-Friendly Interface: The GUI-based design enables both developers and non-security experts to analyze dependency risks efficiently through visualized results and categorized findings.

Weaknesses of ChainGuard

- Performance Overhead: Dynamic analysis and sandbox execution introduce higher runtime and resource consumption compared to pip-audit's lightweight database-based scanning.
- Environmental Dependency: Full isolation requires Docker support. In environments without containerization, ChainGuard falls back to process-level monitoring, which may reduce analysis depth.
- Potential False Positives: Heuristic-based detection of obfuscation or suspicious behavior may occasionally flag benign legacy code.

6.3 Practical Usability Comparison

In practical usage scenarios, pip-audit and ChainGuard serve **complementary roles rather than competing purposes**.

- **Automation and CI/CD (pip-audit):**
  pip-audit is well suited for continuous integration pipelines, where rapid feedback and minimal overhead are critical. It functions effectively as an automated vulnerability gate, preventing known vulnerable dependencies from entering production systems.
- **In-depth Investigation and Risk Assessment (ChainGuard):**
  ChainGuard is optimized for scenarios requiring deeper analysis, such as manual security reviews, forensic investigation, or pre-deployment risk assessment. By combining static inspection, dynamic execution, and AI-assisted interpretation, ChainGuard supports informed decision-making in cases where dependency behavior cannot be assessed through CVE databases alone.

Conclusion of Comparison

In summary, pip-audit provides a reliable baseline for detecting known vulnerabilities, while ChainGuard extends dependency security analysis beyond CVEs by addressing behavioral and supply chain–specific risks.
Rather than replacing existing tools, ChainGuard complements vulnerability scanners by offering a broader and more context-aware view of dependency trustworthiness, particularly in scenarios involving emerging or previously undocumented threats.

7. Discussion and Future Work

7.1 Interpretation of Results

The experimental results demonstrate that ChainGuard is effective in identifying a wide range of software supply chain risks across real-world Python dependencies. The tool successfully detected high-risk packages associated with known vulnerabilities, yanked releases, and suspicious runtime behaviors, while assigning low risk scores to benign utility libraries.

The distribution of risk scores indicates that combining threat intelligence with static and dynamic analysis provides a balanced and interpretable assessment of dependency security. Packages flagged as high risk consistently exhibited multiple independent indicators, such as OSV-reported vulnerabilities and abnormal execution behavior, suggesting that the scoring model accurately reflects real security threats rather than isolated anomalies.

7.2 Limitations

Despite its effectiveness, ChainGuard has several limitations. First, the current implementation focuses exclusively on the Python ecosystem and does not analyze dependencies from other package managers such as npm or RubyGems. Second, dynamic analysis was performed using a process-level sandbox due to environmental constraints, limiting the depth of runtime behavior monitoring compared to full system-call–level sandboxing.

Additionally, ChainGuard relies on rule-based heuristics rather than machine learning models. While this design choice improves explainability, it may reduce detection capability against highly novel or obfuscated attack techniques.

7.3 False Positives and False Negatives

As with any static and dynamic analysis system, ChainGuard may produce false positives and false negatives. Certain benign packages were flagged due to high-entropy strings, external domain references, or legacy code patterns that resemble obfuscation or suspicious behavior. However, the use of multiple independent analysis stages helps mitigate excessive false positives by requiring corroborating evidence.

False negatives may occur in cases where malicious behavior is highly targeted, dormant, or activated only under specific runtime conditions not triggered during sandbox execution. These limitations highlight the inherent trade-off between analysis depth, performance, and coverage.

7.4 Planned Improvements

Future work will focus on extending ChainGuard to support additional ecosystems, including JavaScript (npm) and Java (Maven/Gradle) dependencies. Enhancing the dynamic analysis component with fully isolated container-based sandboxing and system-call monitoring is another planned improvement.

Additional future directions include incorporating machine learning–based anomaly detection to complement rule-based heuristics, improving false positive reduction mechanisms, and enabling continuous dependency monitoring as part of CI/CD pipelines. These enhancements aim to further strengthen ChainGuard's ability to detect evolving software supply chain threats.

8. Conclusion and Author Contributions

8.1 Summary of Contributions

This work introduced ChainGuard, a practical and developer-oriented tool for identifying software supply chain risks within Python projects. The proposed system combines threat intelligence, static code analysis, setup behavior inspection, and dynamic sandbox execution into a unified four-stage risk assessment framework.

The primary contribution of this study is the design and implementation of an explainable, weighted risk scoring model that enables early detection of malicious, vulnerable, and suspicious dependencies before deployment. Experimental results demonstrate that ChainGuard effectively distinguishes between low-risk and high-risk packages across real-world dependency sets.

8.2 Impact on Software Supply Chain Security

ChainGuard contributes to software supply chain security by promoting a shift-left security strategy, allowing developers to assess dependency risks at the earliest stages of development. By providing interpretable risk scores and concrete security findings, the tool supports proactive decision-making rather than reactive incident response.

This work highlights the importance of combining static and dynamic analysis with threat intelligence to improve visibility into dependency-related risks. Although limited to Python in its current form, ChainGuard establishes a strong foundation for future extensions to additional ecosystems and more advanced detection techniques.

8.3 Author Contributions

Muhammet Hamza Taş led the system design and implementation of the ChainGuard tool. He was primarily responsible for developing the static analysis pipeline, integrating threat intelligence sources, designing the risk scoring model, and implementing the dynamic analysis workflow. He also contributed to experiment execution and result interpretation.

Berkay Rigan focused on literature review, background research, and comparative analysis with existing solutions such as MALOSS and pip-audit. He contributed to defining the research problem, identifying gaps in current defenses, and writing the related work and comparison sections of the report.

Ali Rubar Kal contributed to methodology planning, experimental setup,experiment execution and evaluation. He was involved in defining the research problem, preparing the test environment, analyzing detection results, and assisting in report writing and presentation preparation.

All authors collaboratively reviewed the final manuscript, incorporated instructor feedback from the interim presentation, and approved the final version of the report.

9. References

[1] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2020.

[2] P. Phulsungsombati, K. Watanabe, and M. Akiyama, "MALOSS: Automated Malware Detection for Open Source Software Supply Chains," in *Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022.

[3] Python Packaging Authority (PyPA), "pip-audit: A tool for scanning Python environments for known vulnerabilities," GitHub, 2024. [Online]. Available: https://github.com/pypa/pip-audit.

[4] A. Birsan, "Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies," *Medium*, Feb. 2021. [Online]. Available: https://medium.com/@alex.birsan/dependency-confusion-4a5d60ab885.

[5] DataDog, "GuardDog: Identify malicious PyPI and npm packages," GitHub, 2024. [Online]. Available: https://github.com/DataDog/guarddog.

[6] Google, "OSV (Open Source Vulnerabilities)," 2024. [Online]. Available: https://osv.dev.

[7] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages," in *Network and Distributed Systems Security (NDSS) Symposium*, San Diego, CA, USA, Feb. 2021. [Online]. Available: https://dx.doi.org/10.14722/ndss.2021.23055.

[8] M. H. Taş, B. Rigan, and A. R. Kal, "ChainGuard: A project-level software supply chain security analysis tool for Python dependencies," GitHub repository, 2025. [Online]. Available: https://github.com/alirkal34-jpg/ChainGuard