Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Masterarbeit

Informatik

VeXing NPM

Callgraph-Analyse für die Reduzierung von False-Positives bei der Schwachstellenidentifikation in JavaScript

Tim Bastin

Erstprüfer	Prof. Dr. Luigi Lo Iacono
Zweitprüfer	Dr. Marc Ohm
Eingereicht am	15.12.2025

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten Dritter entnommen sind, habe ich als solche kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Insb. wurden zur Erstellung dieser Arbeit auch folgende KI Systeme eingesetzt:

- Claude Sonet in Version 4.5 und Mistrals Le Chat Model wurden zur initialen Ausformulierung basierend auf von mir vorgegebenen Stichpunkten in der gesamten Arbeit eingesetzt.

Mir ist bewusst, dass von KI Systemen generierte Inhalte das sorgfältige wissenschaftliche Arbeiten nicht ersetzen, weshalb sämtliche derartige Inhalte durch mich kritisch überprüft und finalisiert wurden.

Die Arbeit war mit gleichem Inhalt bzw. in wesentlichen Teilen noch nicht Gegenstand einer anderen Prüfung.

Bonn, 15.12.2025

Tim Bastin

Zusammenfassung

Softwareprojekte im Node Package Manager (NPM)-Ökosystem weisen ein Verhältnis von etwa 1:19 zwischen direkten und transitiven Abhängigkeiten auf, was die Komplexität der Schwachstellenbewertung erheblich erhöht. Diese Arbeit belegt, dass gängige metadatenbasierte Schwachstellenscanner wie `npm audit` eine False-Positive-Rate von etwa 70 Prozent aufweisen, da sie ausschließlich die Präsenz vulnerabler Abhängigkeiten prüfen, ohne deren tatsächliche Erreichbarkeit zu berücksichtigen. Zur Reduktion dieser Fehldetektionen wurde eine automatisierte Pipeline entwickelt, die Security-Patch-Commit-Mining, Callgraph-Analyse und systematische Vererbungsanalyse kombiniert. Die Pipeline generiert Vulnerability Exploitability Exchange (VEX)-Statements, die Schwachstellen auf Basis von Erreichbarkeitsanalysen als `affected` oder `not_affected` klassifizieren.

Die Pipeline wurde auf 172.177 NPM-Paketversionen angewendet, die aufgrund ihrer hohen Relevanz und Verbreitung ausgewählt wurden, da sie mehr als 10.000 Downloads in der vorherigen Woche (Stand: 15.08.2025) aufwiesen. Die Ergebnisse zeigen eine exponentielle Abnahme der Propagierungswahrscheinlichkeit mit zunehmender Abhängigkeitstiefe, wobei 89,5 Prozent der vererbten Schwachstellen nur eine Paketgrenze überschreiten. Zudem wurde eine Korrelation zwischen API-Größe und Vererbungswahrscheinlichkeit festgestellt.

Eine manuelle Verifikation von 68 Stichprobenelementen bestätigte eine Präzision von 100 Prozent bei einem Recall von zunächst 45,7 Prozent. Durch Verbesserungen am Prototyp konnten der Recall auf 65,7 Prozent und der F1-Score auf 79,3 Prozent gesteigert werden. Die verbleibenden False-Negatives resultieren vorwiegend aus technischen Limitierungen wie Problemen beim TypeScript-zu-JavaScript-Mapping und CVE-Metadaten-Inkonsistenzen, aber auch aus konzeptionellen Grenzen statischer Code-Analyse.

Die Arbeit schlägt vor, die Metriken Abhängigkeitstiefe und API-Größe in Risikobewertungen zu integrieren. Langfristig sollten VEX-Dokumente von Upstream-Maintainern bereitgestellt werden, um nachgelagerte Analysen zu reduzieren und ein effizienteres Schwachstellenmanagement zu ermöglichen.

Der gesamte in dieser Arbeit entwickelte Quellcode sowie die generierten Ergebnisse und Datensätze sind unter `https://gitlab.com/timbastin/masterarbeit` öffentlich zugänglich.

Inhaltsverzeichnis

Abkürzungsverzeichnis

API Application programming interface 10, 29, 35, 37, 39, 43, 45, 47, 48, 54, 56, 65

AST Abstract Syntax Tree 18, 28

CISA Cybersecurity and Infrastructure Security Agency 15

CPE Common Platform Enumeration 13, 14

CSAF Common Security Advisory Framework 23

CVE Common Vulnerabilities and Exposures 12, 15, 17, 23–27, 31, 38, 39

CVSS Common Vulnerability Scoring System 12, 13, 15, 16, 23, 24, 39, 52, 75, 77

EPSS Exploit Prediction Scoring System 15, 16

FIRST Forum of Incident Response and Security Teams 13, 15, 16, 75

JSON JavaScript Object Notation 14, 34, 41, 65

KEV Known Exploited Vulnerabilities 15

NIST National Institute of Standards and Technology 12

NPM Node Package Manager 3, 7, 8, 10–14, 26, 28, 30, 31, 33–36, 38, 43, 47, 49–52, 54, 57, 62, 65, 72, 73, 75, 78

NVD National Vulnerability Database 8, 12–17

OSV Open Source Vulnerabilities 10, 13–16, 37, 65

PCI DSS Payment Card Industry Data Security Standard 14, 15

PURL Package URL 13, 14, 34, 37, 44, 65

SBOM Software Bill of Materials 76

VEX Vulnerability Exploitability Exchange 3, 12, 23, 24, 32, 43–45, 47, 49, 65, 77, 78, 85, 86

1. Einleitung

1.1. Problemstellung

Die Anzahl gemeldeter Schwachstellen in Softwareprojekten steigt seit Jahren kontinuierlich an. Im Jahr 2024 wurden bereits 40.077 neue Schwachstellen registriert, während es 2023 noch 28.961 waren [9]. Immer häufiger befinden sich unter diesen Schwachstellen auch Sicherheitslücken in weitverbreiteten Software-Bibliotheken, die eine Vielzahl von abhängigen Projekten betreffen. Die Konsequenzen von solchen Schwachstellen und erfolgreichen Angriffen auf die Abhängigkeitsstruktur einer Software sind dabei erheblich, wie prominente Vorfälle eindrücklich zeigen: die Softwareschwachstelle `Log4Shell`, der Angriff auf die Debian-Bibliothek `XZ-Utils` oder der Angriff auf das NPM-Paket `event-stream` [36]. Die Software-Supply-Chain rückt zunehmend in das Visier von Angreifern, da sie durch Multiplikationseffekte ein lohnendes Ziel darstellt [55].

Gerade das JavaScript-Ökosystem, verwaltet über den NPM-Paketmanager, wirkt dabei wie ein Brennglas auf diese Problematik. NPM-Pakete sind wiederverwendbare JavaScript-Softwarebibliotheken, die Funktionalität kapseln und von anderen Projekten als Abhängigkeiten eingebunden werden können. Die NPM-Registry umfasst aktuell ca. 3,5 Millionen Pakete (Stand: 27.04.2025 [47]) und ein Vielfaches an unterschiedlichen Paketversionen, womit sie das größte Softwarerepository weltweit darstellt [24, 55]. Gleichzeitig sind JavaScript-Anwendungen häufig als Webanwendungen öffentlich erreichbar und damit potenziellen Angriffen aus dem gesamten Internet ausgesetzt. Insbesondere in jüngster Vergangenheit häufen sich Angriffe auf dieses Ökosystem: Am 9. September 2025 führte der *qix-Phishing-Angriff* zu mehr als 2,6 Milliarden infizierten Paketversionen [21], am 15. September 2025 infizierte der *Shai-Hulud-Wurm* mehr als 180 Pakete [27], und am 24. November 2025 betraf *Shai-Hulud-Wurm 2.0* mehr als 700 NPM-Pakete [22].

Durchschnittlich 90 % des Codes typischer JavaScript-Anwendungen werden von Drittanbietern in Form von NPM-Paketen bereitgestellt und durch die EntwicklerInnen hinzugeladen. Eine Studie aus dem Jahr 2019 zeigt auf, dass etwa 40 % dieser NPM-Pakete selbst eine Schwachstelle oder aber eine, von einer Schwachstelle betroffene, Abhängigkeit besitzen [55]. Diese Schwachstellen müssen aufwendig von Sicherheitsteams und Softwareentwicklern, auch aufgrund von Compliance-Vorgaben, analysiert, bewertet und gegebenenfalls mitigiert werden [13, 23].

Verschärft wird dieses Problem durch die komplexen Abhängigkeitsbeziehungen im JavaScript-Ökosystem. Pakete deklarieren in ihrer `package.json` direkte

Abhängigkeiten zu anderen Bibliotheken, die sie explizit nutzen. Diese direkten Abhängigkeiten bringen jedoch wiederum eigene Abhängigkeiten mit sich, die als transitive oder indirekte Abhängigkeiten bezeichnet werden. Die Installation einer einzelnen Bibliothek führt dazu, dass durchschnittlich 79 transitive Abhängigkeiten von etwa 39 verschiedenen Maintainern, den für die Entwicklung und Wartung verantwortlichen Personen oder Teams, installiert werden [24, 55].

Diese Abhängigkeitsstruktur hat unmittelbare Konsequenzen für die Schwachstellenbewertung. Schwachstellen in direkten Abhängigkeiten, also in bewusst installierten Paketen, deren APIs im eigenen Quellcode verwendet werden, sind typischerweise leichter zu bewerten. Entwickler sind mit der Funktionalität vertraut und können die Relevanz einer Schwachstelle im konkreten Nutzungskontext einschätzen. Schwachstellen in transitiven Abhängigkeiten hingegen sind deutlich schwieriger zu bewerten, da diese Pakete implizit installiert werden, ihre Funktionalität nicht direkt im eigenen Code aufgerufen wird, und Schwachstellenbeschreibungen ohne Kenntnis des konkreten Verwendungskontexts wenig hilfreich sind. Die Frage, ob eine Schwachstelle in einer transitiven Abhängigkeit tatsächlich das eigene Projekt betrifft, erfordert eine Analyse der gesamten Aufrufkette über möglicherweise mehrere Paketgrenzen hinweg.

Für die Identifikation von Abhängigkeiten mit bekannten Schwachstellen setzen Entwicklerinnen und Entwickler heute hauptsächlich auf automatisierte Schwachstellenscanner. Die verbreitetsten dieser Scanner, wie `npm audit` oder `Snyk`, analysieren die Metadaten der installierten Bibliotheken und nutzen Schwachstellendatenbanken, um nach bekannten Sicherheitslücken in installierten Bibliotheken zu suchen. Konkret findet hier eine Überprüfung auf Basis der Version und des Namens des Pakets statt. In den seltensten Fällen ist allerdings eine Bibliothek in ihrer Gänze vulnerabel.

Die tatsächliche Exploitierbarkeit einer Schwachstelle hängt entscheidend davon ab, ob die verwundbare Funktion innerhalb der Anwendung genutzt wird. Nur wenn der Codepfad zur betroffenen Funktion tatsächlich erreicht wird, besteht ein praktisches Sicherheitsrisiko. Funktionen, die zwar in einer Bibliothek enthalten und als verwundbar identifiziert sind, aber im konkreten Projekt nie aufgerufen werden, sind theoretisch verwundbar, stellen jedoch kein reales Risiko dar. Diese Funktionen erfordern in jedem Fall eine weitere Schwachstelle, die die Ausnutzbarkeit zunächst ermöglicht. Empirische Studien stützen diese Einschätzung: Younis et al. [44] zeigen, dass nur ein kleiner Bruchteil der in der National Vulnerability Database (NVD) gelisteten Schwachstellen tatsächlich ausgenutzt wird. Plate et al. [33] betonen zudem, dass die Exploitierbarkeit stark vom konkreten Nutzungskontext der Anwendung abhängt. Die Berücksichtigung der tatsächlichen Nutzung ermöglicht daher eine präzisere Bewertung der Exploitierbarkeit und reduziert Falsch-Positive-Meldungen, wie sie auch die Autoren der Arbeit *Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for NPM JavaScript Packages* [12] für metadatenbasierte Schwachstellenscanner berichten: 73 % der identifizierten Probleme führen nicht zu einer real ausnutzbaren Schwachstelle, da die betroffenen

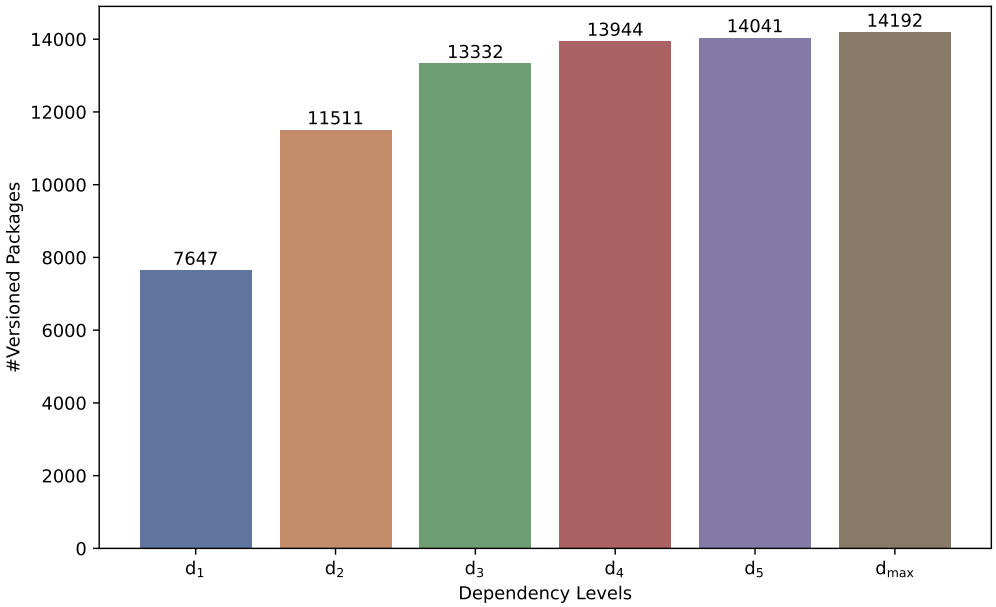

Abbildung 1.1.: Schwachstellen in Bezug auf die Tiefe einer Callgraph-Analyse [31]. Graph zeigt, wie viele Schwachstellen gefunden werden, wenn nach Tiefe d die Analyse abgebrochen wird. Die Darstellung wurde nach [31] erstellt.

Funktionen im Projekt nicht verwendet werden.

Für das Java-Ökosystem zeigen die Autoren der Arbeit *On the Effect of Transitivity and Granularity on Vulnerability Propagation in the Maven Ecosystem* [31], dass von 1,1 Millionen untersuchten Java-Projekten, welche eine Schwachstelle in ihrem Abhängigkeitsbaum besitzen, lediglich 31 % diese Schwachstelle in einer ihrer direkten Abhängigkeiten besitzen. Bei 69 % der Java-Projekte ist hingegen eine transitive Abhängigkeit von einer Schwachstelle betroffen. Nach Durchführung einer ähnlich gearteten Analyse, wie diese Arbeit sie vorstellt, konnten von den 1,1 Millionen Java-Paketen ausschließlich 14.150 Pakete als wirklich vulnerabel identifiziert werden. Es existieren Methodenaufrufe, die die als vulnerabel identifizierten Funktionen aufrufen. Das entspricht umgerechnet einer Falsch-Positiv-Rate von 98,8 %. Ferner untersucht die Studie die Bedeutung der Tiefe einer Schwachstelle innerhalb der Abhängigkeitsstruktur (Abbildungen 1.1). Von insgesamt 14.192 ausnutzbaren Schwachstellen befinden sich 7.647 in direkten Abhängigkeiten (Tiefe 1), 3864 in Tiefe 2, 1821 in Tiefe 3, 612 in Tiefe 4 und 97 in Tiefe 5.

Einige Ökosysteme haben begonnen, symbolbasierte Schwachstellendatenbanken aufzubauen, die Schwachstellen nicht pauschal auf Paketebene, sondern gezielt auf der Ebene von Funktionssignaturen dokumentieren, um die Arbeit von Softwareentwicklern zu vereinfachen, indem weniger Fehlmeldungen produziert werden. Ein bekanntes Beispiel hierfür ist die *GO Vulnerability Database*, die

versucht, verwundbare Funktionen in GO-Projekten eindeutig zu benennen und damit eine wesentlich präzisere Sicherheitsbewertung ermöglicht. „Govulncheck analysiert Ihren Code und zeigt nur Sicherheitslücken an, die Sie tatsächlich betreffen, basierend darauf, welche Funktionen in Ihrem Code transitiv anfällige Funktionen aufrufen“ [46] (ins Deutsche übersetzt). Auch die Open Source Vulnerabilities (OSV) Datenbank erlaubt in ihrem Metamodell-Standard das Hinzufügen von solchen Informationen. Im Golang-Ökosystem wurden 806 von 4219 Schwachstellen um Informationen der vulnerablen Funktionen angereichert. Im Rust-Ökosystem sind es nur noch 172 von 1681 Schwachstellen. Kein anderes in der OSV gelistetet Ökosystem, NPM, NuGet, RubyGems, Maven, Packagist, PyPi, enthält Informationen auf Symbolebene [42] (Analyse durchgeführt am 27. April 2025).

1.2. Forschungsfrage

Softwareentwickler stehen heute vor der Herausforderung, eine Vielzahl von Schwachstellenmeldungen bewerten und adressieren zu müssen. Moderne Schwachstellenscanner generieren dabei häufig eine große Anzahl von Warnungen, von denen viele False-Positives darstellen, also Schwachstellen in transitiven Abhängigkeiten, deren vulnerabler Code im konkreten Anwendungsfall nicht erreichbar ist. Während für statisch typisierte Sprachen wie Java und Golang bereits Ansätze zur Erreichbarkeitsanalyse existieren, die False-Positive-Raten von bis zu 98 % auf wenige Prozent reduzieren können [31], fehlen vergleichbare automatisierte und skalierbare Lösungen für das dynamische JavaScript-Ökosystem. Dies ist besonders problematisch, da NPM aufgrund seiner komplexen Abhängigkeitsstruktur besonders stark unter hohen False-Positive-Raten leidet und gleichzeitig, wie die jüngsten Angriffswellen zeigen, zunehmend im Fokus von Angreifern steht [22, 27].

Vor diesem Hintergrund setzt die vorliegende Arbeit an der folgenden Forschungsfrage an:

Wie propagieren Schwachstellen im NPM-Ökosystem über Abhängigkeitsketten, und in welchem Umfang lassen sich False-Positives durch automatisierte Erreichbarkeitsanalysen reduzieren?

Diese übergeordnete Forschungsfrage lässt sich in die folgenden drei Teilfragen untergliedern, die verschiedene Aspekte der automatisierten Schwachstellenanalyse adressieren:

1. **RQ1: Reduktionspotenzial**: Welcher Anteil der durch metadatenbasierte Scanner gemeldeten Schwachstellen ist tatsächlich über die öffentliche Application programming interface (API) einer JavaScript-Bibliothek erreichbar?

2. **RQ2: Propagierungsmuster**: Welche Muster zeigen sich bei der Vererbung von Schwachstellen über transitive Abhängigkeitsketten?

3. **RQ3: Integration**: Wie lassen sich die gewonnenen Erreichbarkeitsinformationen standardisiert bereitstellen und in bestehende Entwicklungsworkflows integrieren?

Zur Beantwortung der Forschungsfrage entwickelt diese Arbeit einen Ansatz zur automatisierten Anreicherung von Schwachstelleninformationen im NPM-Ökosystem. Dabei wird untersucht, wie sich durch systematische Analyse unterscheiden lässt, welche gemeldeten Schwachstellen tatsächlich das analysierte Paket betreffen und welche aufgrund fehlender Erreichbarkeit keine Bedrohung darstellen. Ein zusätzlicher Aspekt ist dabei die Frage der Kommunikation und Bereitstellung dieser Informationen für die praktische Nutzung.

2. Grundlagen

Die präzise Bewertung von Schwachstellen in modernen Softwareökosystemen erfordert ein Verständnis mehrerer zusammenhängender Konzepte. Dieses Kapitel führt zunächst in die grundlegenden Begriffe und Herausforderungen der Schwachstellenerkennung im NPM-Ökosystem ein. Anschließend werden drei zentrale Verfahren erläutert, die in dieser Arbeit kombiniert werden, um eine automatisierte und kontextbezogene Schwachstellenbewertung zu ermöglichen: Security-Patch-Commit-Mining zur Identifikation des vulnerablen Codes, Callgraph-Analyse zur Bestimmung der Erreichbarkeit und der VEX-Standard zur strukturierten Kommunikation der Ergebnisse.

2.1. Schwachstellen und ihre Dokumentation

Die systematische Erfassung und Dokumentation von Schwachstellen in Softwareprodukten ist ein zentraler Baustein moderner IT-Sicherheit. Sobald Sicherheitslücken entdeckt und öffentlich bekannt gemacht werden, erfolgt ihre strukturierte Aufzeichnung in spezialisierten Schwachstellendatenbanken, die als zentrale Referenzquellen für Sicherheitsanalysen und die automatisierte Überprüfung von Softwareprojekten dienen [3]. Diese Datenbanken ermöglichen es Entwicklern, Sicherheitsverantwortlichen und automatisierten Tools, potenzielle Risiken zu erkennen und Gegenmaßnahmen einzuleiten.

Eine besonders bekannte, in der Forschung verwendete und umfangreiche Schwachstellendatenbank ist die im Jahre 1999 gegründete NVD in den USA, die von der National Institute of Standards and Technology (NIST) betrieben wird [38]. Diese Datenbank enthält mehr als 300.000 unterschiedliche Schwachstellen, sogenannte Common Vulnerabilities and Exposures (CVE), und reichert jede dieser um weitere Informationen, wie den Schweregrad oder die Aussage zur Betroffenheit von Produkten, an. Die NVD führt dabei keine aktiven Schwachstellentests durch, sondern stützt sich ausschließlich auf Informationen, die von Herstellern, unabhängigen Sicherheitsforschern und Koordinierungsstellen für Schwachstellen bereitgestellt werden [38].

Der Schweregrad einer Schwachstelle wird dabei als Common Vulnerability Scoring System (CVSS)-Score quantifiziert, der auf einer Skala von 0 bis 10 vergeben wird. Ab einem Wert von 9,0 werden Schwachstellen als kritisch eingestuft. Die Bewertung erfolgt entweder durch das National Institute of Standards and Technology (NIST) oder durch den Anmelder der Schwachstelle selbst. Die Metriken für die Erstellung des CVSS-Werts der Version 3 sind in drei Metrikgruppen

untergliedert: die Base Metric Group, die Temporal Metric Group und die Environmental Metric Group.

Die Base Metric Group erfasst die grundlegenden Charakteristika einer Schwachstelle, darunter die Einfachheit ihrer Ausnutzung und die potenziellen Auswirkungen eines erfolgreichen Angriffs. Die Temporal Metric Group hingegen berücksichtigt zeitlich variable Faktoren wie die Verfügbarkeit von Exploits oder das Vorhandensein von Patches durch den Hersteller. Die Environmental Metric Group fokussiert sich auf die spezifische Umgebung der vulnerablen Komponente und die dort existierenden Sicherheitsanforderungen [18]. Sowohl die NVD als auch die Organisation Forum of Incident Response and Security Teams (FIRST), die den CVSS-Standard entwickelt, bieten für die Berechnung dieser Werte sogenannte Common Vulnerability Scoring System Calculator an, unter anderem unter `https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator`.

Aus diesen Metrikgruppen lassen sich vier verschiedene Scores ableiten: CVSS-B (Base), CVSS-BT (Base + Temporal), CVSS-BE (Base + Environmental) und CVSS-BTE (Base + Temporal + Environmental). Die NVD wie auch die OSV listen jedoch ausschließlich den CVSS Base Score, da dieser für alle Anwender gleich ist und sich zeitlich nicht ändert.

Um eine Schwachstelle, mit ihrem CVSS-Base-Score nun einem konkreten Softwareprodukt zuzuordnen, nutzt die NVD den Common Platform Enumeration (CPE)-Standard. CPE stellt eine standardisierte Methode zur Beschreibung und Identifikation von Klassen von Anwendungen, Betriebssystemen und Hardwaregeräten dar, wobei nicht individuelle Installationen, sondern abstrakte Produktklassen wie etwa XYZ Visualizer Enterprise Suite 4.2.3 oder XYZ Visualizer (alle Versionen) referenziert werden [50]. Solche CPE–Identifikatoren haben folgendes Format: `cpe:<cpe_version>:<part>:<vendor>:<product>:<version>:<update>:<edition>:<language>:<sw_edition>:<target_sw>:<target_hw>:<other>`. Ist etwa eine Schwachstelle mit folgendem CPE-Eintrag versehen: `cpe:2.3:o:microsoft:windows_10_21h2:*:*:*:*:*:*:*:*`, bedeutet dies, dass die Schwachstelle alle Windows-10-21h2-Versionen betrifft. Das CPE-Schema eignet sich, aufgrund von Informationen über Hersteller, Edition oder Sprache, jedoch primär für kommerzielle Softwareprodukte und Betriebssysteme, führt jedoch bei der Referenzierung von Softwarebibliotheken zu strukturellen Problemen [40]. Open-Source-Bibliotheken verwenden häufig andere Namenskonventionen und folgen Versionierungsschemata, die sich nicht ohne Weiteres in das CPE-Format übertragen lassen. Um diese Lücke zu schließen, und die Referenzierung von Schwachstellen in Open-Source-Bibliotheken zu erleichtern, hat das Unternehmen Google die OSV-Datenbank gegründet [40].

Die OSV-Datenbank ist speziell auf die Anforderungen paketbasierter Ökosysteme ausgerichtet und nutzt Package URLs (PURLs) zur eindeutigen Identifikation von Softwarepaketen [40]. Eine Package URL (PURL) folgt dem Schema `pkg:<type>/<namespace>/<name>@<version>`, beispielsweise `pkg:npm/lodash@4.17.21` für das NPM-Paket lodash in Version 4.17.21. Dieses Format erleichtert eine präzise und ökosystemübergreifende Referenzierung

von Bibliotheken und vermeidet die Komplexität wie auch Mehrdeutigkeit des CPE-Formats.

Ein weiterer Vorteil der OSV-Datenbank besteht in der Bereitstellung eines einheitlichen JavaScript Object Notation (JSON)-basierten Formats für alle unterstützten Ökosysteme, darunter NPM, Maven, PyPI, Go und Debian. Diese Standardisierung und die Bereitstellung über JSON-APIs erleichtern die automatisierte Verarbeitung von Schwachstelleninformationen [42]. Die in Schwachstellendatenbanken wie der NVD und OSV erfassten Informationen bilden die Grundlage für automatisierte Schwachstellenscanner. Diese Tools gleichen die in einem Softwareprojekt installierten Pakete gegen die Datenbanken ab und identifizieren dabei bekannte Sicherheitslücken. Der Abgleich erfolgt typischerweise auf Basis der Paketidentifikatoren (CPE oder PURL) und der Versionsnummern, um festzustellen, ob eine installierte Version von einer dokumentierten Schwachstelle betroffen ist. Bei einem Treffer geben die Scanner eine Warnung aus, die die Entwickler auf die Notwendigkeit von Updates oder Patches hinweist.

Das Tool `npm audit` beispielsweise liest die `package-lock.json`-Datei eines Projekts, die alle installierten Pakete samt ihrer exakten Versionen und transitiven Abhängigkeiten auflistet. Anschließend werden diese Informationen mit der Schwachstellendatenbank abgeglichen. Wird eine Übereinstimmung gefunden, meldet `npm audit` die betroffenen Pakete zusammen mit Metadaten wie dem Schweregrad (Base-Score) der Schwachstelle und gegebenenfalls Empfehlungen zur Behebung, etwa durch ein Update auf eine gepatchte Version [37]. Dieser metadatenbasierte Ansatz ist effizient und skaliert gut für große Projekte mit umfangreichen Abhängigkeitsbäumen. Allerdings weist er eine fundamentale Limitierung auf: Er prüft ausschließlich die Präsenz vulnerabler Pakete, ohne zu berücksichtigen, ob der vulnerable Code im konkreten Anwendungsfall tatsächlich genutzt wird.

2.2. Bewertung von Schwachstellenmeldungen

Nachdem Schwachstellen in einem Softwareprojekt identifiziert wurden, müssen diese bewertet werden, um fundierte Entscheidungen über deren Behandlung treffen zu können. Diese Bewertung ist nicht nur eine technische Notwendigkeit, sondern wird auch durch verschiedene Compliance-Frameworks und Sicherheitsstandards gefordert. Der Cyber Resilience Act, der Payment Card Industry Data Security Standard (PCI DSS), die ISO 27001 oder der IT-Grundschutz des BSI verlangen explizit, dass identifizierte Schwachstellen systematisch bewertet werden müssen [5, 13, 23, 43].

Nach ISO 27001 umfasst dieser Bewertungsprozess zwei zentrale Aspekte: Erstens muss das Risiko betrachtet werden, das bei einer erfolgreichen Ausnutzung der Schwachstelle entsteht. Zweitens muss die Wahrscheinlichkeit eingeschätzt werden, mit der die Schwachstelle tatsächlich ausgenutzt werden kann. Dieser Prozess muss reproduzierbar sein und dokumentiert werden [23]. Dies führt unwei-

gerlich zu der praktischen Frage, ob eine gemeldete Schwachstelle im konkreten Anwendungsfall überhaupt ausnutzbar ist. Die Beantwortung dieser Frage stellt Entwickler vor erhebliche Herausforderungen. Schwachstellenbeschreibungen in CVE-Einträgen und Security Advisories, wie die NVD und OSV sie listet, sind häufig kurz und untechnisch gehalten. Für die fundierte Bewertung des Risikos werden jedoch tiefe technische Details benötigt: Welche spezifischen Funktionen sind betroffen? Unter welchen Bedingungen kann die Schwachstelle ausgenutzt werden? Wird der vulnerable Code im eigenen Projekt überhaupt aufgerufen? Diese Informationen sind in den Standarddatenbanken typischerweise nicht verfügbar [45].

Viele Unternehmen verwenden ausschließlich den CVSS-Base-Score, der häufig direkt von den Schwachstellendatenbanken gelistet wird, als Risiko-Metrik. Einige Compliance-Frameworks, wie der PCI DSS, schreiben die Berücksichtigung dieser Metrik sogar vor [43]. Die Organisation FIRST allerdings rät eindringlich davon ab, da der CVSS Base Score kein vollständiges Risiko darstellt. Der User Guide für den CVSS 3.1 enthält hierzu einen expliziten Abschnitt: „Concerns have been raised that the CVSS Base Score is being used in situations where a comprehensive assessment of risk is more appropriate. The CVSS v3.1 Specification Document now clearly states that the CVSS Base Score represents only the intrinsic characteristics of a vulnerability which are constant over time and across user environments. The CVSS Base Score should be supplemented with a contextual analysis of the environment, and with attributes that may change over time by leveraging CVSS Temporal and Environmental Metrics. More appropriately, a comprehensive risk assessment system should be employed that considers more factors than simply the CVSS Base Score. Such systems typically also consider factors outside the scope of CVSS such as exposure and threat.“ Ein vergleichbarer Hinweis findet sich auch im User Guide für die neuere Version CVSS 4.0 [15, 16].

Es gibt auch grundlegende Kritik am CVSS-Scoring-Algorithmus, da dieser weder formal noch empirisch ausreichend validiert ist [49]. Als Alternative oder Ergänzung zum CVSS wird häufig der Exploit Prediction Scoring System (EPSS) herangezogen, der die Wahrscheinlichkeit einer Ausnutzung auf Basis realer Exploit-Daten vorhersagen soll. Der EPSS wird ebenfalls von der Organisation FIRST bereitgestellt und gibt die Wahrscheinlichkeit an, mit der diese konkrete Schwachstelle innerhalb der nächsten 30 Tage ausgenutzt wird [17]. Allerdings zeigt die aktuelle Studie *Conflicting Scores, Confusing Signals: An Empirical Study of Vulnerability Scoring Systems* aus dem Jahr 2025 [26], dass der EPSS die tatsächliche Ausnutzung von Schwachstellen nur unzureichend vorhersagt:

- Nur 19,9 % der später aktiv ausgenutzten Schwachstellen (gemäß dem Cybersecurity and Infrastructure Security Agency (CISA) Known Exploited Vulnerabilities (KEV)-Katalog) wiesen einen EPSS-Score $> 0{,}5$ (50 % Wahrscheinlichkeit) auf.
- 22 % der ausgenutzten Schwachstellen hatten überhaupt keinen EPSS-Sco-

re.

- Insgesamt korrelieren verschiedene Scoring-Systeme (CVSS-B, Exploitability-Index, SSVC-H und EPSS) kaum miteinander, was ihre Aussagekraft infrage stellt.

Die Autoren der Studie empfehlen daher, Scoring-Systeme zwar als Hilfsmittel zu nutzen, jedoch zusätzliche unternehmensspezifische Kriterien einzubeziehen, wie etwa die Kritikalität betroffener Assets oder die geschäftlichen Auswirkungen einer erfolgreichen Ausnutzung. Ein effektiver Ansatz besteht darin, interne Heuristiken oder Overlays zu entwickeln, die Scores im Kontext der eigenen Infrastruktur interpretieren [26]. Diese Anforderung ähnelt stark der auch von der FIRST geforderten Nachpriorisierung des CVSS-B durch die Anwendung von Threat- und Environmental-Metriken zur Erzeugung des CVSS-BTE.

Die Bewertung von Schwachstellen ist dabei untrennbar mit der Frage nach deren Behebung verbunden, was zusätzliche praktische Schwierigkeiten mit sich bringt. Die naheliegendste Maßnahme zur Behebung einer Schwachstelle besteht im Update der betroffenen Bibliothek auf eine gepatchte Version. In frühen Entwicklungsphasen eines Projekts ist dies häufig unproblematisch, da sich die Software noch in einem experimentellen Stadium befindet und Änderungen an den Abhängigkeiten leicht integriert werden können. Je reifer ein Softwareprojekt jedoch wird, desto problematischer werden Updates von Abhängigkeiten. Jedes Bibliotheksupdate kann Release-Pläne verzögern, erfordert zusätzlichen Test- und Integrationsaufwand und birgt das Risiko, neue Bugs oder Breaking Changes in die Software einzuführen [45]. Dies führt zu einem Spannungsfeld: Einerseits erfordern Compliance-Vorgaben und Sicherheitsüberlegungen eine zeitnahe Behebung von Schwachstellen, andererseits können hastige Updates die Stabilität und Zuverlässigkeit der Software gefährden.

Dieses Dilemma wird durch die hohe Rate an False-Positives bei metadatenbasierten Scannern verschärft. Wenn ein Großteil der gemeldeten Schwachstellen das Projekt faktisch nicht betrifft, weil der vulnerable Code nicht erreichbar ist, führt dies zu einer ineffizienten Ressourcenallokation: Entwickler investieren Zeit in die Bewertung und Behebung von Schwachstellen, die keine tatsächliche Bedrohung darstellen, während potenziell kritische Sicherheitslücken in der Masse der Meldungen übersehen werden können. Eine präzisere Methodik zur Unterscheidung zwischen tatsächlich relevanten und nicht relevanten Schwachstellen ist daher sowohl aus technischer als auch aus organisatorischer Sicht erstrebenswert.

2.3. Security-Patch-Commit-Mining

Einen vielversprechenden Ansatz zur Lösung dieses Problems bietet das Security-Patch-Commit-Mining an. Während viele Einträge in Schwachstellendatenbanken wie der NVD oder OSV lediglich grundlegende Metadaten wie CVE-IDs,

Schweregrade oder betroffene Versionen enthalten, verfügen einige Einträge über zusätzliche Referenz-Links zu konkreten Änderungen in Versionskontrollsystemen wie GitHub oder GitLab. Diese Links verweisen auf tatsächliche Code-Änderungen, sogenannte Security-Patches, die von Entwicklern vorgenommen wurden, um die dokumentierte Schwachstelle zu beheben. Durch die Analyse dieser Patches lassen sich weitere Informationen über die Schwachstelle gewinnen, welche sich auch automatisiert verarbeiten lassen. Diese Informationen bilden im Rahmen der Methodik dieser Arbeit den Grundstein für die automatisierte Betroffenheitsanalyse.

Ein zentraler Ansatz in diesem Bereich ist die Studie *CVEFixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software* [3], welche CVE-Einträge aus der in 2.1 eingeführten NVD-Datenbank nach Security--Patch-Commit-Referenzen untersucht, diese extrahiert und analysiert, mit dem Ziel, ein Datenset mit vulnerablem Code und dessen Mitigation aufzustellen [56].

Diese Commits (Code-Änderungen) enthalten präzise Informationen über die vorgenommenen Änderungen, einschließlich betroffener Dateinamen, Zeilennummern sowie die hinzugefügten oder entfernten Code-Zeilen. Die Autoren der Studie *Vulnerability discovery based on source code patch commit mining: a systematic literature review* zeigen einen automatisierbaren Prozess auf, solche Informationen in großem Umfang zu extrahieren [56]. Durch die Analyse der Unterschiede zwischen der vulnerablen und der gepatchten Version lassen sich die tatsächlich problematischen Funktionen oder Codezeilen approximieren. Hierfür wird folgende Heuristik herangezogen: Codezeilen oder Funktionen, die in einem Security-Patch modifiziert werden, werden als vulnerabel klassifiziert [31, 44, 45].

Diese Annahme folgt dem Prinzip, dass eine Änderung zur Behebung einer Schwachstelle direkt auf die Lokalisierung des problematischen Codes schließen lässt: Eine Schwachstelle ist definiert als ungewünschtes Systemverhalten, das durch spezifische Codeausführung hervorgerufen wird. Um dieses Verhalten zu korrigieren, muss zwingend der Code modifiziert werden, der das problematische Verhalten verursacht oder ermöglicht. Dies führt zu einer direkten kausalen Beziehung zwischen der Codeänderung und der Schwachstelle. Wenn eine Schwachstelle in Zeile V durch Änderung einer Zeile Z behoben wird, dann muss eine der folgenden kausalen Beziehungen vorliegen.

Direkte Verursachung Zeile Z implementiert das unerwünschte Verhalten direkt. Es gilt: $Z = V$.

Ermöglichung Zeile Z schafft die notwendigen Voraussetzungen, unter denen das unerwünschte Verhalten auftreten kann. Es existiert ein Aufrufpfad $Z \rightarrow^* A$, sodass die Ausführung von Z mit spezifischen Eingabewerten deterministisch zur Erreichung des unerwünschten Codes in Zeile V führt. Dies schließt die imperative Ausführung der nächsten Funktionszeilen mit ein.

Fehlende Kontrolle Zeile Z unterlässt notwendige Sicherheitsprüfungen, die das

unerwünschte Verhalten verhindern würden. Ohne die durch Z fehlende Kontrolle existiert ein exploitierbarer Aufrufpfad $Z \rightarrow^* V$, der bei ordnungsgemäßer Implementierung von Z blockiert worden wäre.

Da die Schwachstelle ohne die Änderung von Zeile Z weiter bestand hätte, muss also gelten: $Z = V \vee Z \rightarrow^* V$.

Das Security-Patch-Commit-Mining liefert somit den ersten notwendigen Schritt für eine präzise Schwachstellenanalyse: die Identifikation der konkret betroffenen Codezeilen und Funktionen. Diese können anschließend auf ihre Erreichbarkeit im Projektkontext untersucht werden (vgl. 2.4). Erst durch diese Kombination aus Patch-basierter Lokalisierung und kontextsensitiver Erreichbarkeitsanalyse lässt sich eine automatisierte Betroffenheitsanalyse realisieren, die False-Positives reduziert und die Priorisierung kritischer Schwachstellen ermöglicht.

2.4. Callgraphen

Sobald der vulnerable Teilbereich einer Bibliothek identifiziert wurde, stellt sich die Frage, welche Funktionen diesen vulnerablen Bereich direkt oder indirekt aufrufen. Zur Beantwortung dieser Frage eignet sich die Callgraph-Analyse, eine zentrale Methode der Programmanalyse, die die Aufrufbeziehungen zwischen Funktionen oder Methoden eines Programms systematisch abbildet. Ein Callgraph ist ein gerichteter Graph, dessen Knoten die Funktionen eines Programms repräsentieren, während die gerichteten Kanten die Aufrufbeziehungen zwischen diesen Funktionen modellieren. Eine Kante von Knoten A zu Knoten B bedeutet, dass die Funktion A im Programmablauf die Funktion B aufruft [48].

Die Konstruktion von Callgraphen kann sowohl statisch als auch dynamisch erfolgen, wobei beide Ansätze unterschiedliche Stärken und Schwächen aufweisen [20].

Die statische Callgraph-Analyse basiert auf der Quellcodeanalyse ohne Programmausführung. Hierfür wird typischerweise der Abstract Syntax Tree (AST) des Programms verwendet, eine hierarchische Repräsentation der syntaktischen Struktur des Quellcodes. Der AST abstrahiert die konkrete Syntax der Programmiersprache und stellt die logische Struktur des Programms dar, einschließlich aller Funktionsdefinitionen, Kontrollstrukturen und Aufrufe (siehe Abbildung 2.1). Durch die Traversierung des AST können Aufrufbeziehungen zwischen Funktionen extrahiert und in einen Callgraphen überführt werden [7, 48].

Ein Vorteil der statischen Analyse liegt in ihrer Vollständigkeit: Da sie alle theoretisch möglichen Aufrufpfade erfasst, eignet sie sich besonders für die frühe Erkennung potenzieller Schwachstellen, selbst wenn bestimmte Code-Pfade zur Laufzeit nie durchlaufen werden. Allerdings können statische Callgraphen bei dynamischen Sprachkonstrukten (z. B. Reflection, dynamische Codegenerierung oder Polymorphie) unpräzise werden, da diese erst zur Laufzeit aufgelöst und daher nicht immer korrekt im statischen Graphen abgebildet werden können [20].

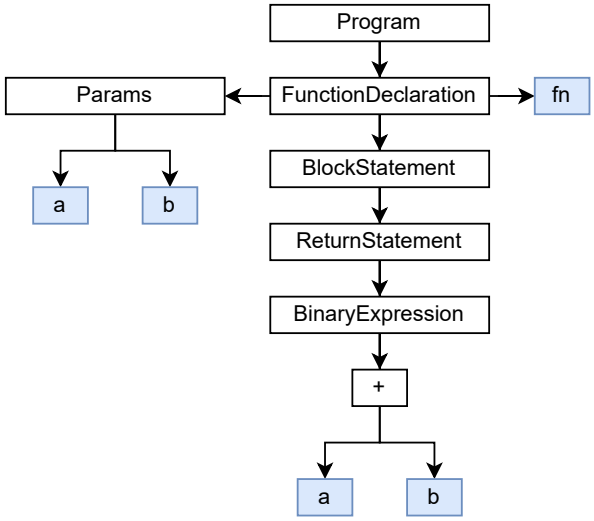

Abbildung 2.1.: Beispiel Abstract-Syntax-Tree des Programms: `function fn(a,b) {return a+b}`. Erstellt mit *acorn* (`https://www.npmjs.com/package/acorn`) und lose in Diagram überführt, ausschließlich zur Symbolisierung, ohne Anspruch auf Korrektheit

Die dynamische Callgraph-Analyse hingegen wird zur Laufzeit des Programms durchgeführt, indem die tatsächlichen Aufrufbeziehungen während der Programmausführung beobachtet werden. Dieser Ansatz erfordert die Ausführung des Codes, etwa durch eine Test-Suite oder gezielte Programmläufe. Im Java-Umfeld kann hierfür beispielsweise das JVM-Eventsystem genutzt werden, das über das `JVMTI_EVENT_METHOD_ENTRY`-Event den Eintritt in Methoden erfasst und somit die Aufrufbeziehungen dynamisch protokollieren kann [20].

Der Hauptvorteil der dynamischen Analyse liegt in ihrer Präzision: Da sie nur die tatsächlich ausgeführten Pfade erfasst, spiegeln dynamische Callgraphen das reale Laufzeitverhalten des Programms wider. Allerdings ist ihre Abdeckung auf die durch Tests abgedeckten Pfade beschränkt, was bedeutet, dass ungetestete Code-Bereiche nicht im Graphen erscheinen [20]. Dies kann zu falsch-negativen Ergebnissen führen, wenn kritische Aufrufpfade nicht durch die Testfälle aktiviert werden.

Idealerweise sollte ein Callgraph sowohl *sound* (vollständig) als auch *präzise* sein [20, 48]. Soundness bedeutet, dass alle zur Laufzeit möglichen Aufrufbeziehungen im Graph erfasst sind, während Präzision verlangt, dass keine falschen Kanten existieren. Ein Graph mit einer fehlenden Kante $A \rightarrow B$ wäre unsound, wenn zur Laufzeit A tatsächlich B aufrufen kann. Ein Graph mit einer zusätzlichen Kante $C \rightarrow D$ wäre unpräzise, wenn C niemals D aufruft. Diese beiden Eigenschaften stehen in einem Spannungsverhältnis: Höhere Soundness geht ty-

Listing 2.1: Pseudocode für den Algorithmus einer Taint-Analyse zur Schwachstellenvererbung

```
Algorithm: VulnerabilityPropagation(CallGraph G, Set<
    Function> VulnerableFunctions)
Initialize WorkList = VulnerableFunctions
Initialize VulnerableSet = VulnerableFunctions
while WorkList is not empty do
     current = WorkList.pop()
     for each predecessor p of current in G do
         if p not in VulnerableSet then
             VulnerableSet.add(p)
             WorkList.add(p)
         end if
    end for
end while
return VulnerableSet
```

pischerweise mit geringerer Präzision einher.

Callgraphen bieten unabhängig von der gewählten Konstruktionsmethode eine wertvolle Grundlage für die Erreichbarkeitsanalyse von Schwachstellen, wie sie auch in dieser Arbeit eingesetzt wird. Eine zentrale Eigenschaft besteht darin, dass sowohl Knoten als auch Kanten eines Callgraphen mit zusätzlichen Metadaten angereichert werden können. Besonders relevante Metadaten sind dabei die Start- und Endzeilen der Funktionen im Quellcode. Diese Informationen ermöglichen es, in Kombination mit Techniken zur Identifizierung vulnerabler Codebereiche, wie dem bereits vorgestellten Security-Patch-Commit-Mining, den exakten Codebereich einer Funktion zu lokalisieren. Dadurch kann präzise bestimmt werden, ob eine als vulnerabel markierte Codezeile innerhalb einer bestimmten Funktion liegt. Auf dieser Basis lässt sich ein initialer Satz vulnerabler Funktionen ableiten, der als Ausgangspunkt für eine systematische Erreichbarkeitsanalyse dient.

Nach Aufbau der Graphstruktur des Programm erfolgt die eigentliche Erreichbarkeitsanalyse oder auch Schwachstellenpropagation mittels Taint-Analyse [35], eine systematische Traversierung des konstruierten Callgraphen. Ausgehend von den als vulnerabel identifizierten Funktionen wird eine Rückwärtssuche durchgeführt, um alle Funktionen zu markieren, die direkt oder indirekt mit diesen Funktionen in Beziehung stehen.

Auf Basis dieser Analyse lässt sich jede Funktion ebenso als vulnerabel markieren, die eine vulnerable Funktion aufruft [35]. Hierfür kann der in Listing 2.1 dargestellte Algorithmus verwendet werden. Diese Propagation erfolgt iterativ: Eine Funktion wird als potenziell vulnerabel markiert, wenn sie über einen beliebig langen Aufrufpfad zu einer bestätigten vulnerablen Funktion führen kann. Die Komplexität dieses Algorithmus ist $O(V + E)$, wobei V die Anzahl der Knoten

Listing 2.2: Beispiel für einen statischen und dynamischen require-Aufruf

```
1 // --- Statischer require-Aufruf ---
2 const fs = require("fs"); // Modulname ist bekannt
3 fs.readFileSync("example.txt", "utf8");
4
5 // --- Dynamischer require-Aufruf ---
6 const moduleName = process.argv[2]; // Modulname kommt zur
     Laufzeit
7 const mod = require(moduleName);
8 mod.run();
```

(Funktionen) und E die Anzahl der Kanten (Aufrufbeziehungen) darstellt.

2.4.1. Limitierungen

Die Qualität und Vollständigkeit eines Callgraphen hängen maßgeblich von der verwendeten Analysemethode ab. Statische Analyseverfahren können durch indirekte Funktionsaufrufe, Polymorphismus oder dynamische Bindung erschwert werden. In objektorientierten Sprachen führen virtuelle Methodenaufrufe zu zusätzlicher Komplexität, da zur Compile-Zeit nicht eindeutig bestimmbar ist, welche konkrete Implementierung zur Laufzeit aufgerufen wird.

Konkret ergeben sich für eine statische Analyze zwei zentrale zu lösende Problemklassen:

1. **Welche Funktionen existieren?** Dynamische Sprachkonstrukte können zur Laufzeit neue Funktionen oder Module erzeugen, die im statischen Code nicht sichtbar sind. Dadurch bleibt die Menge der tatsächlich existierenden Funktionen unvollständig, was unmittelbar die Knotenmenge des Callgraphen beeinflusst.

2. **Welche Aufrufpfade existieren?** Auch wenn alle Funktionen bekannt sind, ist nicht immer eindeutig bestimmbar, welche Aufrufbeziehungen tatsächlich bestehen. Dynamisch registrierte Aufrufe verändern die Aufrufstruktur zur Laufzeit. Dies führt zu Unsicherheiten in der Kantenmenge des Graphen und damit zu einer unvollständigen Darstellung der tatsächlichen Programmstruktur.

Im JavaScript-Ökosystem treten diese Probleme besonders deutlich auf. Wesentliche dynamische Konstrukte, welche in die erste Problemklasse fallen sind:

- **Dynamischer `require`-Aufruf:** Die `require`-Funktion ist das zentrale Modulsystem in Node.js (CommonJS) und ermöglicht das Einbinden externer Module. Bei statischen Aufrufen ist der Modulname als String-Literal festgelegt, sodass der Analyzer die geladenen Module eindeutig bestimmen

Listing 2.3: Beispiel für eine Event-getriebene Architektur

```
1 const EventEmitter = require('events');
2 const emitter = new EventEmitter();
3
4 function handlerA(data) {
5     // ...
6 }
7 emitter.on('process', handlerA);
8 // Event-Dispatch
9 emitter.emit('process', 'Ereignis gestartet'); // ruft
    handlerA auf
```

kann. Bei dynamischen Aufrufen wird der Modulname zur Laufzeit konstruiert, z.B. aus Variablen oder Funktionsparametern, wodurch der Analyzer die geladenen Module nicht vorhersagen kann (siehe Listing 2.2). Dies führt zu unvollständigen Graphdarstellungen.

- **Eval-basierte Codegenerierung:** JavaScript-Code, der durch `eval()`, `Function()` oder Template-Systeme zur Laufzeit erzeugt wird, ist für statische Analysewerkzeuge unsichtbar. Solcher Code kann neue Aufrufpfade schaffen, die in keinem statisch konstruierten Callgraphen erscheinen.

Die zweite Problemklasse in JavaScript wird insbesondere durch folgende Konstrukte bestimmt:

- **Event-getriebene Architekturen:** Event-Listener werden oft dynamisch registriert und durch externe Ereignisse ausgelöst. Die Zuordnung von Aufrufer zu Aufgerufenen ist zur Compile-Zeit häufig nicht erkennbar, was zu unvollständigen Kanten im Callgraphen führt. Listing 2.3 zeigt die beispielhafte Verwendung eines EventEmitters. Der Aufruf zu `handlerA` ist für einen statischen Analyzer nicht erkennbar.
- **Monkey-Patching:** Techniken, bestehende Objekte, Klassen oder Prototypen zur Laufzeit zu modifizieren, können die Aufrufstruktur verändern. Statische Analyzer können solche dynamischen Änderungen nicht zuverlässig erfassen, wodurch Aufrufpfade übersehen werden.

In Kapitel 4.9 wird erläutert, wie mit diesen Limitierungen im Rahmen dieser Arbeit verfahren wird und welche konkreten Maßnahmen zur Erkennung dieser getroffen wurden.

2.5. VEX-Format für standardisierte Schwachstellenkommunikation

Die traditionelle Schwachstellenkommunikation über CVE-Identifikatoren und CVSS-Scores ist, wie bereits beschrieben, oft zu grobgranular für die komplexen Abhängigkeitsverhältnisse moderner Softwaresysteme. Ein CVE-Eintrag gibt lediglich an, dass eine Schwachstelle in einer bestimmten Version einer Bibliothek existiert, nicht jedoch, ob diese in einem konkreten Anwendungskontext tatsächlich ausnutzbar ist. Das VEX-Format adressiert dieses Problem durch die Einführung von Zustandsinformationen, die präzise beschreiben, wie eine Schwachstelle im Kontext eines bestimmten Produkts zu bewerten ist. Statt einer binären *betroffen/nicht betroffen*-Klassifikation ermöglicht der VEX eine granularere Betrachtung der tatsächlichen Auswirkungen [11].

Die praktische Umsetzung des VEX-Standards erfolgt vorwiegend über drei maschinenlesbare Formate, die jeweils spezifische Anwendungsbereiche und technische Anforderungen adressieren:

CycloneDX CycloneDX stellt einen Standard für Software Bill of Materials (SBOM) dar, der eine strukturierte Erfassung von Softwarekomponenten und deren Schwachstellen ermöglicht. Das Format integriert VEX-Informationen direkt in die SBOM-Struktur.

Common Security Advisory Framework (CSAF) CSAF bietet einen standardisierten Ansatz für die Erstellung und insbesondere die Verteilung von Sicherheitsadvisories. Neben dem konkreten Format definiert der Standard auch Anforderungen an die Bereitstellung des VEX-Dokuments [39], beispielsweise die Veröffentlichung unter *.well-known* URLs. Ein praktisches Beispiel hierfür ist die Bereitstellung von VEX-Dokumenten und Security Advisories durch Red Hat Product Security unter folgender URL: `https://security.access.redhat.com/data/csaf/v2/provider-metadata.json`.

OpenVEX OpenVEX konzentriert sich ausschließlich auf den Austausch von VEX-Informationen und implementiert in seinem Standard eine minimale Version eines gültigen VEX-Dokuments, wie es in [51] gefordert wird [41].

Die Struktur eines VEX-Dokuments ähnelt sich in den benannten drei Standards stark. Jeder der Standards fokussiert die Übertragung folgender Informationen [11].

Produkt-Identifier Der Produkt-Identifier ist eine eindeutige Zuordnung der analysierten Softwarekomponente zu einem spezifischen Produkt oder einer definierten Produktfamilie.

Version Die Versionen des Produktes, welche mit der Schwachstellenanalyseinformation addressiert werden.

Component-Identifier Der Component-Identifier spezifiziert die Herkunft der identifizierten Schwachstelle, typischerweise in Form einer externen Bibliothek, eines Frameworks oder einer Drittanbieter-Komponente.

Mitigation Das Mitigationsfeld dokumentiert konkrete, implementierbare Maßnahmen zur Risikominderung für den Fall, dass das analysierte Produkt nachweislich von der identifizierten Schwachstelle betroffen ist.

Analysestatus Der Analysestatus kategorisiert den aktuellen Bewertungsstand der Schwachstelle anhand eines standardisierten Klassifikationsschemas: `affected` signalisiert eine bestätigte Betroffenheit des Produkts, `not affected` markiert das Fehlen einer Sicherheitsbedrohung, `under investigation` kennzeichnet eine bislang nicht abgeschlossene Bewertung, während `fixed` die erfolgreiche Behebung der Schwachstelle dokumentiert.

Status-Begründung Zusätzlich gehört zu dem Analysestatus eine weitere, nicht notwendigerweise maschinenlesbare, detaillierte Begründung für die Klassifizierung eines Produkts als `not affected`.

Timestamp Der Timestamp dokumentiert den präzisen Zeitpunkt der initialen Analyse oder der letzten Aktualisierung der Bewertung. Diese zeitliche Referenz ist wichtig für die Gewährleistung der Aktualität der Sicherheitsinformation und ermöglicht eine Nachverfolgung von Änderungen.

Schwachstellen-Identifier Eine eindeutige Kennung der Schwachstelle. Häufig eine CVE-ID oder eine GitHub Security Advisory ID. Zusätzlich sind hier auch Bewertungen in Hinblick auf die Kritikalität, bspw. mittels CVSS-Score, zu finden.

Listing A zeigt ein VEX-Dokument im CycloneDX Format. Es dokumentiert die Bewertung der Schwachstelle CVE-2025-50181 in der urllib3-Bibliothek für ein OCI-Container-Image. Das Dokument identifiziert das betroffene Produkt durch die Package URL im `metadata.component`-Bereich und spezifiziert die vulnerable Komponente `pkg:pypi/urllib3@1.26.20` im `affects`-Array. Der entscheidende Teil ist die `analysis`-Sektion: Der Status `not_affected` signalisiert, dass trotz Vorhandensein der vulnerablen Bibliothek keine tatsächliche Bedrohung vorliegt. Die Begründung erklärt, dass die betroffenen Funktionen in diesem spezifischen Anwendungskontext nicht aufgerufen werden.

Diese VEX-Meldung verhindert nicht nur Fehlalarme in automatisierten Sicherheitsscans, sondern bietet Entwicklern eine präzise und kontextbezogene Risikobewertung anstelle einer pauschalen Warnung vor der CVE-2025-50181. Damit stellt der VEX eine gezielte Lösung für das Problem transitiver Schwachstellen

dar. Er ermöglicht eine schrittweise Präzisierung von Schwachstelleninformationen entlang der Abhängigkeitsketten: Während ein CVE zunächst nur die Existenz einer Schwachstelle dokumentiert, kann jede Downstream-Komponente ihren Abhängigkeiten standardisiert mitteilen, ob und wie die Schwachstelle in ihrem spezifischen Kontext tatsächlich relevant ist.

3. Forschungsstand

Die Untersuchung der Schwachstellenpropagation im NPM-Ökosystem sowie die Reduktion von False-Positives durch Erreichbarkeitsanalysen basiert auf der Integration zweier zentraler Forschungsbereiche: dem Security-Patch-Commit-Mining zur präzisen Identifikation vulnerabler Code-Bereiche und der Callgraph-Analyse zur Bestimmung der Erreichbarkeit dieser Bereiche innerhalb des Programms. Die Kombination beider Ansätze ermöglicht eine Vulnerability-Reachability-Analyse, die eine fundierte Bewertung der tatsächlichen Ausnutzbarkeit von Schwachstellen erlaubt.

Im Folgenden wird der Forschungsstand zu diesen Teilgebieten strukturiert dargestellt.

3.1. Security-Patch-Identifikation

Die Identifikation von Security-Patches stellt eine zentrale Herausforderung in der automatisierten Schwachstellenanalyse dar und ist Gegenstand aktiver Forschung. Die Grundidee, Security-Patches zur Identifikation von vulnerablem Code zu nutzen, basiert auf einem einfachen, aber wirkungsvollen Rückschluss: Ein Security-Patch behebt eine Schwachstelle durch gezielte Code-Änderungen. Indem man die im Patch geänderten Zeilen identifiziert, lässt sich der ursprünglich vulnerable Code lokalisieren. Dieser Ansatz wurde bereits in mehreren Arbeiten erfolgreich angewendet, darunter die beiden Arbeiten der Autoren S.Ponta, H. Plate und A. Sabetta *Detection, assessment and mitigation of vulnerabilities in open source dependencies* [45] und die Studie *Impact assessment for vulnerabilities in open-source software libraries* [44]. Im Hinblick auf die Identifikation von vulnerablem Code bildet die Arbeit *CVEFixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software* und der darin beschriebene Prozess auf Basis von Patch-Referenzen die Grundlage für diese Arbeit [3]. Der Ansatz extrahiert Security-Patches direkt aus den in CVE-Einträgen referenzierten Links und ermöglicht damit eine automatisierte Zuordnung von Schwachstellen zu den entsprechenden Code-Änderungen.

Der Ansatz der Studie *CVEFixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software* wird von den Autoren J. Akhoundali, S. R. Nouri, K. Rietveld und O. Gadyatskaya in der Arbeit *MoreFixes: A Large-Scale Dataset of CVE Fix Commits Mined through Enhanced Repository Discovery* erweitert [1]. In dieser Arbeit betrachten die Autoren nicht ausschließlich Referenz-Links, sondern analysieren Quellcode-Repositories tiefergehend. Dies um-

fasst beispielsweise die Analyse von Commit-Messages, um weitere Security-Patches zu identifizieren, die aus verschiedenen Gründen nicht dem CVE-Record hinzugefügt wurden. Die Autoren zeigen auf, dass sie durch diesen Ansatz 2,95-mal mehr Patch-Commits identifizieren können als die CVEFixes-Datenbank.

Die Literaturübersicht *Vulnerability Discovery Based on Source Code Patch Commit Mining*[56] liefert einen umfassenden Einblick in den aktuellen Forschungsstand zur automatisierten Identifikation von Security-Patches. Ein zentraler Fokus der Arbeit liegt auf der Erkennung von Patches, die von Herstellern nicht explizit als sicherheitsrelevant gekennzeichnet wurden. Die vorgestellten Ansätze nutzen vorrangig Machine-Learning-Modelle, die Software-Repositories systematisch analysieren. Die untersuchten Methoden unterscheiden sich dabei vor allem in den herangezogenen Informationsquellen: Einige Studien konzentrieren sich auf die Analyse des Patch-Codes und seiner Metadaten, während andere primär Commit-Messages auswerten oder beide Quellen kombinieren, um die Genauigkeit der Patch-Erkennung zu erhöhen.

Allerdings kommt die Review zu dem Schluss, dass Machine-Learning-basierte Ansätze in diesem Gebiet unter erheblichen False-Positive- und False-Negative-Raten leiden. Die Arbeit E-SPI [54] erreicht eine False-Positive-Rate von 12,5 % und eine False-Negative-Rate von 8,8 %. Die Arbeit PatchRNN [53] schneidet mit 11,6 % False-Positive-Rate und 26,3 % False-Negative-Rate sogar schlechter ab. Die Arbeit von Wang et al. [52] weist mit 41,3 % die höchste False-Positive-Rate auf. Diese hohen Fehlerraten verdeutlichen die Komplexität der automatisierten Security-Patch-Identifikation und die Schwierigkeit, relevante von irrelevanten Code-Änderungen zu unterscheiden.

Weitere Forschungsarbeiten befassen sich mit anderen Aspekten der Security-Patch-Analyse. Die Studie *Patchworking* [6] untersucht beispielsweise den Aufwand von Security-Patches und analysiert, welche Code-Änderungen notwendig waren. Auch diese Arbeit stützt sich ausschließlich auf die Einträge in den CVE-Referenzen, analog zu CVEFixes [3].

Insgesamt zeigt der Forschungsstand der Security-Patch-Identifikation mehrere komplementäre Ansätze. Häufig bilden die Referenzen in CVE-Einträgen die Basis, während andere Arbeiten versuchen, zusätzliche Patches von nicht gemeldeten Schwachstellen zu identifizieren. Da die vorliegende Arbeit die Kommunikation von Schwachstelleninformationen automatisieren und erleichtern soll, sind nicht gemeldete Schwachstellen als außerhalb des Untersuchungsbereichs zu betrachten. Der Fokus liegt auf der präzisen Analyse offiziell dokumentierter Schwachstellen und deren Propagierung durch Abhängigkeitsketten.

3.2. Callgraph-Analyse

Die Konstruktion präziser Callgraphen für JavaScript-Code stellt aufgrund der dynamischen Natur der Sprache eine besondere Herausforderung dar und ist Gegenstand intensiver Forschung. JavaScript erlaubt zur Laufzeit die Manipulation

von Objekten, dynamische Property-Zugriffe und die Erzeugung von Code durch `eval` oder ähnliche Konstrukte, was die statische Analyse erheblich erschwert. Diese Arbeit stützt sich auf das Tool NPM-jelly [8], dessen Entwickler auf mehrere prägende Arbeiten verweisen [14, 28, 29, 32, 35], die jeweils unterschiedliche Aspekte der Callgraph-Konstruktion adressieren.

In der Arbeit [35] entwickeln die Autoren ein Tool für den Aufbau modularer Callgraphen mit dem Namen *JAM*. Der modulare Ansatz ist dabei von besonderer Bedeutung, da moderne JavaScript-Anwendungen typischerweise aus zahlreichen Modulen bestehen, die über verschiedene Mechanismen wie CommonJS oder ES6-Imports eingebunden werden. Die Autoren zeigen, dass der von JAM produzierte Callgraph mit einer Präzision von 84,35 % und einem Recall von 98,62 % dem Tool `js-callgraph` deutlich überlegen ist, welches nur 58,64 % Präzision und 48,16 % Recall erreicht. Diese Metriken sind, wie bereits erwähnt, besonders relevant für die Bewertung der Qualität eines Callgraphen: Die Präzision gibt an, welcher Anteil der identifizierten Kanten tatsächlich zur Laufzeit auftreten kann, während der Recall misst, welcher Anteil aller tatsächlich möglichen Kanten vom Tool erkannt wird.

Die Arbeit [14] stellt einen fundamentalen Algorithmus zur Konstruktion von Callgraphen vor, der auf der Analyse des AST basiert. Der AST repräsentiert, wie in 2.1 dargstellt und bereits kurz beleuchtet, die syntaktische Struktur des Programmcodes in Baumform und bildet die Grundlage für zahlreiche statische Analysetechniken. Auf Basis des AST werden sogenannte Flow-Graphen erzeugt, indem der Algorithmus den AST traversiert und Kanten dem Callgraphen auf Basis eines fixen Regelsets hinzufügt. Kanten werden etwa hinzugefügt, wenn Variablen oder Funktionen definiert werden, wenn Funktionen aufgerufen werden, oder wenn Closures erzeugt werden. Der Algorithmus eignet sich sowohl für intraprozeduralen Flow, also Datenfluss innerhalb einer einzelnen Funktion, als auch für interprozeduralen Flow zwischen verschiedenen Funktionen. Auf Basis dieses Regelsets mit zehn präzise definierten Regeln wird der Callgraph aufgebaut.

Die Autoren erreichen eine Präzision von etwa 80 %, während der Recall zwischen 80 % und bei einigen Projekten fast 100 % liegt. Diese hohen Werte demonstrieren, dass ein regelbasierter Ansatz für viele JavaScript-Programme ausreichend genaue Callgraphen erzeugen kann. Als Hauptursachen für Ungenauigkeiten identifizieren die Autoren Funktionen, die in Eigenschaften mit demselben Namen gespeichert sind und die eine feldbasierte Analyse nicht als unterschiedliche Aufrufziele differenzieren kann. Ein Beispiel hierfür sind Objekte verschiedener Typen, die alle eine Methode `process` implementieren, eine statische Analyse kann ohne zusätzliche Typinformationen nicht eindeutig bestimmen, welche konkrete Implementierung bei einem Aufruf von `obj.process()` ausgeführt wird. Die Experimente fokussierten sich allerdings ausschließlich auf web-basierte JavaScript-Programme, was eine gewisse Einschränkung der Generalisierbarkeit auf Node.js-Umgebungen darstellt.

Die Arbeit [28] adressiert fehlende Vollständigkeit von JavaScript-Callgraphen, die aufgrund dynamischer Zuweisungen von Objekteigenschaften entstehen. Feh-

lende Vollständigkeit bedeutet in diesem Kontext, dass der konstruierte Callgraph nicht alle zur Laufzeit möglichen Aufrufbeziehungen abbildet, was zu einem unvollständigen Bild der tatsächlichen Programmstruktur führt. Der vorgeschlagene Ansatz zur Analyse besteht aus einer innovativen Kombination von dynamischer Voranalyse, die Informationen über die dynamische Objektmanipulation ableitet, und statischer Analyse, die die abgeleiteten Informationen nutzt, um den Grad der Unsicherheit zu reduzieren.

Die entscheidende Ergänzung ist, einzelne Codefragmente durch tatsächliche Ausführung zu untersuchen und so dynamische Objektzugriffe und deren Verhalten zu beobachten. Diese Beobachtungen werden in die statische Analyse integriert und verbessern so deren Genauigkeit. Die Autoren evaluieren diesen Ansatz auf 36 JavaScript-Projekten und zeigen, dass der Recall von 75,9 % auf 88,1 % steigt, ohne dass die Präzision leidet. Die Steigerung um über 12 Prozentpunkte beweist, wie effektiv die Kombination aus statischer und dynamischer Analyse ist.

Die Arbeit [32] stellt ein Pattern-Matching-Tool bereit und formuliert eine formale Grammatik zur Abbildung von JavaScript-Funktionen mittels sogenannter Access Paths. Access Paths sind strukturierte Beschreibungen, wie auf Funktionen und Objekte im Code zugegriffen wird, beispielsweise `<module>.functionName()`. Die Autoren zeigen, dass sich die formulierte Grammatik eignet, um Bibliotheksfunktionen präzise zu beschreiben und von internen Implementierungsdetails zu abstrahieren. Zusätzlich stellen sie das Tool *TAPIR* vor, welches auf Basis einer statischen Code-Analyse API-Nutzungen finden kann, gegeben eines Access-Path-Patterns. Diese Access-Path-Notation bildet eine zentrale Grundlage für die in dieser Arbeit durchgeführte Symbolidentifikation, da sie eine global eindeutige Referenzierung von Funktionen über Paketgrenzen hinweg ermöglicht.

Die umfassende vergleichende Studie *Is JavaScript Call Graph Extraction Solved Yet? A Comparative Study of Static and Dynamic Tools* [2] untersucht verschiedene Callgraph-Werkzeuge sowohl für die statische als auch für die dynamische Analyse. Darunter befindet sich neben `npm-cg`, `Closure`, WALA, TAJS, `NodeProf` und `nodejs-cg` auch das bereits beschriebene Werkzeug ACG [28]. Die Tools werden mittels des SunSpider Benchmarks und der Evaluation auf verschiedenen Node.js-Module verglichen, wobei sowohl Präzision als auch Recall systematisch gemessen werden.

Die Autoren kommen zu mehreren wichtigen Erkenntnissen: Statische Tools können einen großen Anteil aller Kanten im Graphen erkennen, wobei die konkreten Werte stark vom analysierten Code abhängen. Der Großteil der Kanten, die ausschließlich von dynamischen Tools erkannt werden konnten, stammt von dynamischen Aufrufbeziehungen, die für statische Analyzer prinzipiell nicht sichtbar sind. Dies unterstreicht die in Abschnitt 4.9 diskutierten fundamentalen Limitierungen statischer Analyse.

Eine zentrale Erkenntnis der Studie ist, dass ausschließlich ACG [28] sich für die Analyse von Node.js-Modulen aufgrund der umfassenden Sprachunterstüt-

zung und relativen Präzision eignet. ACG erreicht bei Node.js-Modulen eine Präzision von 16,93 % und einen Recall von 63,53 %. Diese vergleichsweise niedrige Präzision bedeutet, dass ein erheblicher Anteil der identifizierten Kanten möglicherweise zur Laufzeit nie auftreten, was zu konservativen Analyseergebnissen führt. Dynamische Ansätze erreichen zwar eine Präzision von 100 %, da sie ausschließlich tatsächlich beobachtete Aufrufe registrieren, finden jedoch aufgrund unzureichender Testsuiten einige Kanten nicht. Dies führt zu einem niedrigeren Recall, da nur die während der Testausführung tatsächlich durchlaufenen Pfade erfasst werden.

Die Autoren kommen zu dem Schluss, dass sowohl dynamische als auch statische Ansätze kombiniert werden sollten, um optimale Ergebnisse zu erzielen. Statische Ansätze bieten eine umfassende Abdeckung aller möglichen Pfade, während dynamische Ansätze eine hohe Präzision für die tatsächlich ausgeführten Pfade liefern. Diese Erkenntnis motiviert hybride Ansätze, wie sie in neueren Arbeiten verfolgt werden, und unterstreicht gleichzeitig die Notwendigkeit, die Limitierungen der gewählten Analysetechnik transparent zu dokumentieren, wie es in dieser Arbeit durch die Konfidenzmetriken geschieht.

3.3. Erreichbarkeitsanalyse von Schwachstellen

Die Analyse der Erreichbarkeit von Schwachstellen stellt einen vielversprechenden Ansatz zur Reduktion von False-Positives in der automatisierten Schwachstellenerkennung dar. Verschiedene Forschungsarbeiten haben diesen Ansatz für unterschiedliche Programmiersprachen und Ökosysteme untersucht.

NPM-jelly stellt grundlegende Funktionalität für die Erreichbarkeitsanalyse auch in einem Sicherheitskontext bereit. Allerdings wird von den Autoren nicht vorgestellt, wie die erforderlichen Access-Path-Patterns erstellt werden können, insbesondere nicht auf automatisierte Weise. Diese Lücke zwischen der theoretischen Möglichkeit der Erreichbarkeitsanalyse und ihrer praktischen Anwendung im großen Maßstab adressiert die vorliegende Arbeit durch die Entwicklung einer vollständig automatisierten Pipeline zur Generierung von Access-Path-Patterns aus Security-Patch-Commits.

Eclipse Steady [45] nutzt einen konzeptionell gleichwertigen Ansatz wie ihn diese Arbeit vorstellt, allerdings für die Analyse von Java-Projekten. Der Ansatz basiert ebenfalls auf der Konstruktion von Callgraphen und der Identifikation vulnerablen Codes durch Analyse von Security-Patches. Die Autoren extrahieren aus Fix-Commits die geänderten Code-Bereiche und nutzen diese als Ausgangspunkt für eine Erreichbarkeitsanalyse. Der Aufbau eines Callgraphen in Java ist allerdings nicht so herausfordernd wie für JavaScript, da Java als statisch typisierte Sprache deutlich weniger dynamische Konstrukte aufweist und die meisten Methodenaufrufe bereits zur Compiliezeit aufgelöst werden können. Dennoch demonstriert Eclipse Steady die grundsätzliche Wirksamkeit des Ansatzes und zeigt, dass eine erhebliche Reduktion von False-Positives durch präzise

Erreichbarkeitsanalyse möglich ist.

Die Autoren der Arbeit *Code-based Vulnerability Detection in Node.js Applications: How Far Are We?* [7] stellen einen sehr ähnlichen Ansatz für das JavaScript- und NPM-Ökosystem vor, welcher auf Eclipse Steady basiert und dieses weiterentwickelt. Die Schwachstellendatenbank wird allerdings manuell aus Security-Patch-Commits extrahiert, was die Skalierbarkeit des Ansatzes erheblich einschränkt. Die Autoren konzentrieren sich in ihrer Evaluation ausschließlich auf das GitHub Enterprise System von SAP, wodurch die Generalisierbarkeit der Ergebnisse auf das öffentliche NPM-Ökosystem begrenzt ist. Für die Identifizierung von vulnerablen Symbolen wurde ANTLR-v4 genutzt, ein Parser-Generator, der die syntaktische Analyse des Codes ermöglicht. Ein wesentlicher Unterschied zur vorliegenden Arbeit besteht darin, dass kein vollständiger Callgraph aufgebaut wird, sondern ausschließlich direkt genutzte Library-Funktionen identifiziert werden. Dies bedeutet, dass transitive Aufrufe und die Propagierung von Schwachstellen über mehrere Funktionsebenen hinweg nicht erfasst werden, was zu einer unvollständigen Analyse der tatsächlichen Erreichbarkeit führen kann.

Die Arbeit *On the Effect of Transitivity and Granularity on Vulnerability Propagation in the Maven Ecosystem* [31] untersucht das Maven-Ökosystem und bietet wichtige Erkenntnisse zur Schwachstellenpropagierung in paketbasierten Ökosystemen. Die Autoren identifizieren Patch-Commits anhand der in CVE-Einträgen hinterlegten Referenzen sowie weiterer Heuristiken, um auch Patches zu erfassen, die nicht explizit referenziert werden. Vulnerabler Code wird auf Basis der Änderungen im Patch-Commit identifiziert, analog zum in dieser Arbeit verfolgten Ansatz. Das Werkzeug OPAL [10] wird von den Autoren genutzt, um den Java-Callgraphen aufzubauen und die Erreichbarkeit vulnerabler Methoden zu analysieren.

Die zentrale Erkenntnis dieser Arbeit ist, dass etwa 98 % der Schwachstellenmeldungen im Java-Ökosystem False-Positives sind, wenn ausschließlich auf der Präsenz vulnerabler Abhängigkeiten geachtet wird, ohne die tatsächliche Erreichbarkeit zu berücksichtigen. Diese hohe False-Positive-Rate unterstreicht die Notwendigkeit präziser Erreichbarkeitsanalysen und motiviert die Entwicklung automatisierter Ansätze wie der in dieser Arbeit vorgestellten Methodik. Die Autoren zeigen zudem, dass der exponentielle Zusammenhang zwischen der Tiefe der Abhängigkeit und der Wahrscheinlichkeit der Schwachstellenpropagierung auch im Maven-Ökosystem besteht, was die in dieser Arbeit für das NPM-Ökosystem gefundenen Ergebnisse bestätigt.

Zusammenfassend zeigt der Forschungsstand, dass Erreichbarkeitsanalysen ein wirksames Mittel zur Reduktion von False-Positives darstellen, jedoch bisher primär für statisch typisierte Sprachen wie Java umgesetzt wurden. Für das dynamische JavaScript-Ökosystem existieren zwar konzeptionelle Ansätze, diese wurden jedoch bisher nicht in dem Umfang automatisiert und skaliert, wie es für die praktische Anwendung auf das gesamte NPM-Ökosystem erforderlich wäre.

Ein wesentlicher Aspekt, der die praktische Umsetzung solcher Analysen in der Vergangenheit erschwerte, war das Fehlen eines standardisierten Formats zur

Kommunikation der gewonnenen Erkenntnisse. Selbst wenn präzise Erreichbarkeitsanalysen durchgeführt wurden, existierte keine etablierte Möglichkeit, diese Informationen strukturiert und maschinenlesbar an andere Stakeholder weiterzugeben. Erst durch das heutige Aufkommen des VEX-Standards werden diese Schwachstelleninformationen skalierbar kommunizierbar. Es ist von begrenztem Nutzen, aufwendig Informationen über die tatsächliche Betroffenheit von Paketen zu sammeln, wenn diese nicht auf standardisiertem Weg anderen Entwicklern, Security-Tools und Organisationen bereitgestellt werden können. VEX schafft hier einen gemeinsamen Standard, der die Integration in bestehende Security-Pipelines und Software Composition Analysis (SCA) Tools ermöglicht und damit erstmals eine ökosystemweite Nutzung solcher Analyseergebnisse erlaubt.

Die vorliegende Arbeit adressiert diese Forschungslücke durch die Entwicklung einer vollständig automatisierten Pipeline, die auf Basis von Callgraph-Analysen präzise VEX-Statements generiert und diese über eine standardisierte Schnittstelle bereitstellt. Dadurch werden die gewonnenen Erkenntnisse nicht nur theoretisch fundiert, sondern auch praktisch anwendbar. Die Arbeit baut dabei auf den etablierten Erkenntnissen und Methodiken der bestehenden Forschung auf und führt diese zu einem operationalisierbaren Ansatz zusammen.

4. Methodik

Zur Beantwortung der Forschungsfragen wurde ein systematischer, mehrstufiger Analyseprozess entwickelt. Dieses Kapitel stellt die methodische Vorgehensweise vor, die für die automatisierte Untersuchung der Schwachstellenpropagation im NPM-Ökosystem konzipiert wurde. Der Prozess umfasst dabei alle wesentlichen Schritte: von der initialen Auswahl relevanter Pakete über die Identifikation vulnerabler Code-Bereiche und den Aufbau von Callgraphen bis hin zur Analyse der Schwachstellenvererbung über Paketgrenzen hinweg.

Die Methodik der Arbeit ist in Abbildung 4.1 als Diagramm dargestellt. Für den Ablauf der Methodik ist zunächst auf die Unterscheidung in die grün, blau und gelb markierten Schritte hinzuweisen.

Der gesamte in dieser Arbeit entwickelte Quellcode sowie die generierten Ergebnisse und Datensätze sind unter `https://gitlab.com/timbastin/masterarbeit` öffentlich zugänglich.

4.1. Auswahl der zu analysierenden Pakete

Um die Propagierung von Schwachstellen über Abhängigkeitsketten im NPM-Ökosystem zu untersuchen und das Reduktionspotenzial von False-Positives durch Erreichbarkeitsanalysen zu bewerten, ist eine repräsentative und praxisrelevante Auswahl der zu analysierenden Pakete und Schwachstellen entscheidend. Dieser Schritt (Schritt ❶ im Methodik-Diagramm) legt den Grundstein für die statistische Aussagekraft der späteren Ergebnisse.

Da die tatsächliche Verbreitung und Nutzung eines Pakets entscheidend dafür ist, ob eine darin enthaltene Schwachstelle praktische Auswirkungen entfaltet und überhaupt erkannt sowie gemeldet wird, orientiert sich die Auswahl der zu analysierenden Pakete an den Downloadzahlen im NPM-Ökosystem. Pakete mit hohen Downloadzahlen werden nicht nur häufiger in realen Projekten eingesetzt, sondern bergen auch ein erhöhtes Risikopotenzial für die Verbreitung von Schwachstellen über Abhängigkeitsketten. Diese Fokussierung auf stark genutzte Pakete ermöglicht zudem eine effiziente Ressourcenverteilung, da die Analyse auf diejenigen Komponenten beschränkt wird, die für die Mehrzahl der Entwickler und Anwendungen von direkter Relevanz sind.

Als primäre Datenquelle dient der offizielle `/downloads`-Endpunkt der NPM-Registry, der Downloadstatistiken für alle Projekte bereitstellt. Zur effizienten Datenbeschaffung wird das NPM-Paket `download-counts` (`https://www.npmjs.com/package/download-counts`) genutzt, das diese Informa-

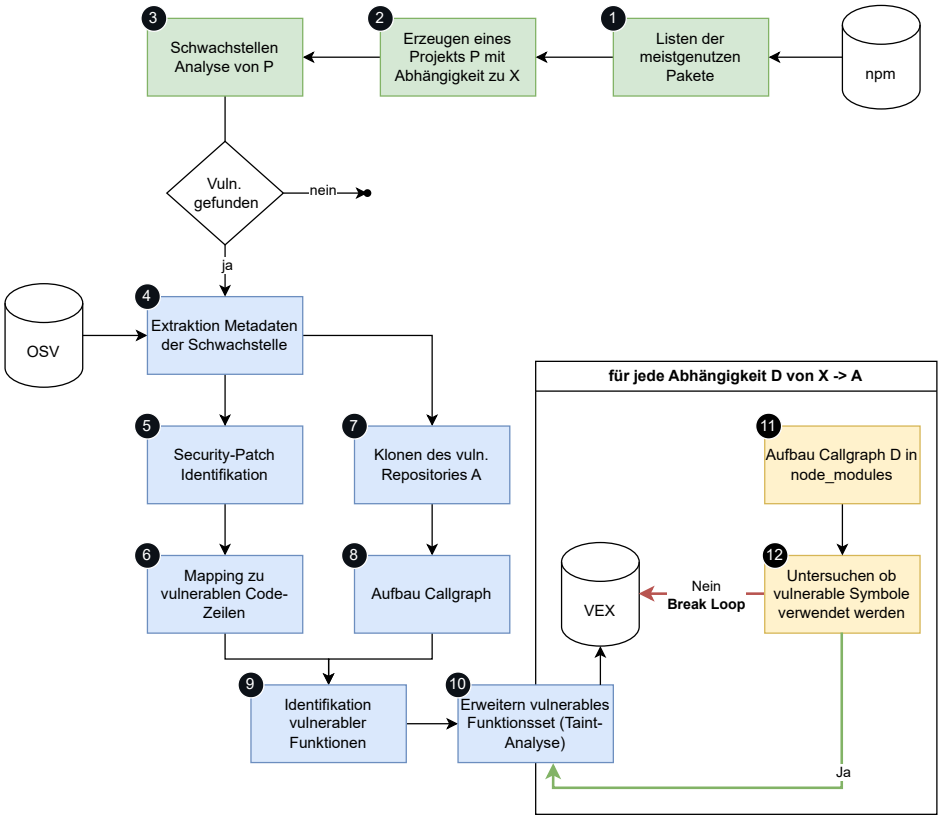

Abbildung 4.1.: Flowchart-Diagramm der Analyse-Pipeline

tionen in einer konsolidierten JSON-Datei zur Verfügung stellt. Das Download-Counts-Projekt nutzt eine GitHub-Action um die Daten zweimal monatlich zu aktualisieren und eine neue Version des download-counts-Pakets zu veröffentlichen. Dies stellt eine aktuelle Datenbasis dar und bildet die Grundlage für die Paketauswahl. Zum Zeitpunkt der Datenerhebung (15.08.2025) umfasst das NPM-Register insgesamt 3.530.446 Pakete [34].

Für die Analyse von Quellcode sind jedoch nicht Pakete als solche relevant, sondern das versionsspezifische Tupel `(Paket, Version)`, das als PURL im Format `pkg:\gls{NPM}/<paketname>@<version>` repräsentiert wird. Der Download-Endpunkt der NPM-Registry liefert die Statistiken allerdings nur pro Projekt, nicht pro konkrete Version. Da EntwicklerInnen in der Praxis stets eine bestimmte Version eines Pakets installieren und nicht ein gesamtes Projekt in allen Versionen, ist ein mehrstufiger Filterprozess zur Identifikation relevanter Paketversionen nötig.

Im ersten Schritt werden alle Pakete mit mehr als 10.000 Downloads pro Monat ausgewählt. Dies ergibt eine Teilmenge von 81.199 Paketen, was etwa 2,3

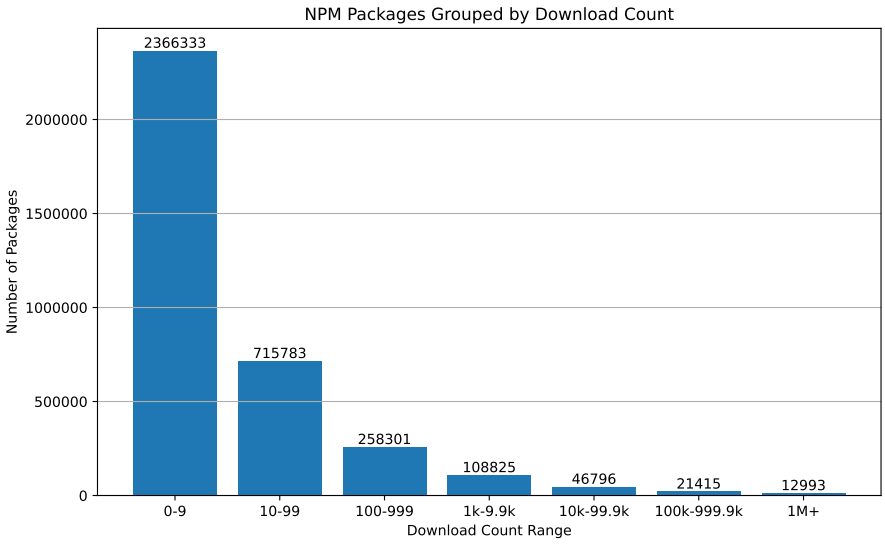

Abbildung 4.2.: Verteilung der Downloads im NPM-Ökosystem (Stand 15.08.2025)

% des gesamten NPM-Registers entspricht. Diese Konzentration auf eine verhältnismäßig kleine Gruppe ist charakteristisch für die Downloadverteilung im NPM-Ökosystem: Während wenige Pakete extrem hohe Downloadzahlen aufweisen, verzeichnet die überwiegende Mehrheit nur sehr geringe Downloadzahlen (siehe Abbildung 4.2). Diese Verteilung folgt einem typischen Long-Tail-Muster.

Im zweiten Schritt werden für jedes der 81.199 ausgewählten Pakete die meistgenutzten Versionen identifiziert. Hierzu wird der NPM-API-Endpunkt `https://api.npmjs.org/versions/<paketname>/last-week` verwendet, der die Downloadzahlen einzelner Versionen für die vergangene Woche liefert.

Ausschließlich Versionen mit mehr als 10.000 Downloads in der vergangenen Woche werden in die finale Auswahl aufgenommen. Dieser Filterungsprozess führt zur Identifikation von 178.775 `(Paket, Version)`-Tupeln, die als Grundlage für die nachfolgende, möglichst praxisnahe Analyse dienen. Es ist dabei anzumerken, dass durch diese Fokussierung nur noch 40.979 unterschiedliche Projekte berücksichtigt werden, da lediglich diese mindestens eine Version mit 10.000 Downloads aufweisen. Im Vergleich dazu hatten 81.199 Pakete über alle Versionen hinweg aggregiert mindestens 10.000 Downloads.

4.2. Untersuchen jedes Tupels (Paket, Version) auf Schwachstellen

Im zweiten und dritten Schritt der Methodik, dargestellt durch ❷ und ❸ im Diagramm, erfolgt die systematische Untersuchung jedes identifizierten `(Paket,`

`Version)`-Tupels auf bekannte Schwachstellen. Dieser Prozess gliedert sich in mehrere aufeinanderfolgende Schritte.

Zunächst wird für jedes zu analysierende Tupel eine isolierte Testumgebung geschaffen (❷). Dazu wird mittels `npm init` ein neues, leeres NPM-Paket initialisiert. In diese Umgebung wird anschließend das zu analysierende Paket in der spezifischen Version mittels `npm install --ignore-scripts --omit=dev --ignore-platform <paket>@<version>` installiert. Die verwendeten Flags dienen der Sicherheit und Effizienz:

- `--ignore-scripts` unterdrückt die Ausführung von Post-Install-Skripten, die in der Vergangenheit wiederholt für Software-Supply-Chain-Angriffe auf das NPM-Ökosystem missbraucht wurden [22, 27]. Durch diese Maßnahme wird das Risiko minimiert, dass schädlicher Code während der Installation ausgeführt wird.

- `--omit=dev` schließt Entwicklungsabhängigkeiten (z. B. Testframeworks oder Build-Tools) von der Installation aus, da diese für die Analyse der Produktionsabhängigkeiten nicht relevant sind und die Komplexität unnötig erhöhen würden.

- `--ignore-platform` ermöglicht die Installation selbst dann, wenn die Paketmetadaten Plattforminkompatibilitäten angeben (z. B. für bestimmte Betriebssysteme oder Architekturen). Dies ist besonders wichtig, um die Analyse nicht durch plattformspezifische Einschränkungen zu behindern, die für die statische Codeanalyse irrelevant sind.

Durch diesen Installationsprozess wird nicht nur das primäre Paket in der gewünschten Version installiert, sondern auch sämtliche transitiven Abhängigkeiten aufgelöst. Dadurch entsteht ein vollständiger Abhängigkeitsbaum, der alle für die Analyse relevanten Pakete und Versionen umfasst.

Nach erfolgreicher Installation wird eine metadatenbasierte Schwachstellenanalyse mittels `npm audit` durchgeführt (❸). Dieser Standardbefehl des NPM-Ökosystems prüft alle installierten Pakete gegen die offizielle NPM-Schwachstellendatenbank und identifiziert bekannte Sicherheitslücken, indem er die `package-lock.json`-Datei analysiert, die exakte Versionsinformationen aller installierten Abhängigkeiten enthält. Das Ergebnis dieser Prüfung bestimmt das weitere Vorgehen: Werden keine Schwachstellen gefunden, wird der Analyseprozess für das aktuelle `(Paket, Version)`-Tupel beendet. Werden jedoch Schwachstellen identifiziert, erfolgt eine detaillierte, kontextsensitive Analyse der konkreten Sicherheitslücken, wie in Abschnitt 4.3 beschrieben. Die Ergebnisse des `npm audit` dienen ebenfalls als repräsentativer Wert für metadatenbasierte Schwachstellenscanner und ermöglichen später einen Vergleich mit den in dieser Arbeit entwickelten Analysemethoden. Dies erlaubt eine Bewertung der möglichen Reduktion von False-Positives.

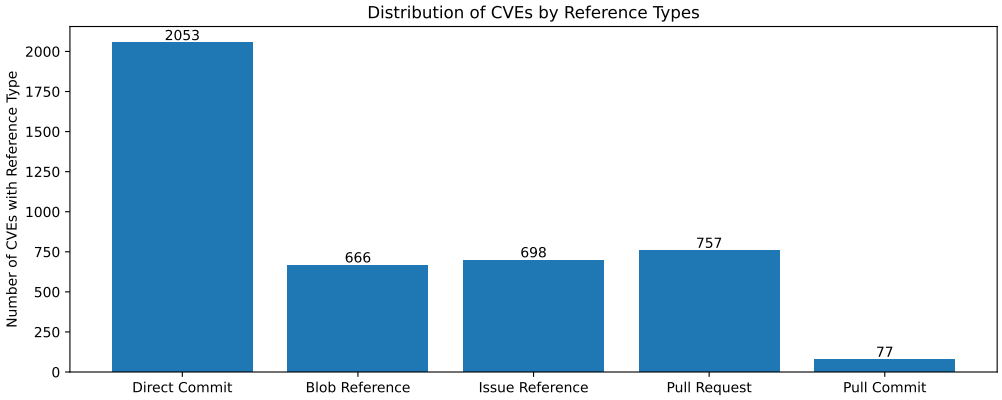

Abbildung 4.3.: Verteilung der einzelnen Referenztypen

4.3. Extraktion von Metadaten

Die in Abbildung 4.1 blau markierten Arbeitsabläufe behandeln die Analyse einer gefundenen Schwachstelle. Sie werden für jede identifizierte Schwachstelle eines jeden `(Paket, Version)`-Tupels durchgeführt und dienen der systematischen Extraktion relevanter Metadaten.

Zunächst wird geprüft, ob für die gefundene Schwachstelle bereits ein Eintrag in der CVEfixes-Datenbank [3] existiert. Falls kein Eintrag vorhanden ist, wird die OSV Datenbank per API abgefragt, um die relevanten Schwachstelleninformationen zu erhalten (❹). Für diese Abfrage ist die PURL des betroffenen Pakets erforderlich, die sich jedoch einfach und fehlerfrei durch eine String-Ersetzung gemäß dem Schema `pkg:npm/<paketname>@<version>` erzeugen lässt [42]. Die OSV-Datenbank aggregiert Schwachstellendaten, wie bereits beschrieben, aus verschiedenen Quellen und stellt diese in einem strukturierten Format bereit [42].

Anschließend werden die Metadaten der Schwachstelle systematisch nach Referenzen untersucht, die auf potenzielle Security-Patches hinweisen (❺). Dabei werden sowohl Web- als auch Fix-Referenzen analysiert, die auf GitHub-Ressourcen verweisen. Konkret wird nach folgenden Referenztypen gesucht:

Direkte Commit-Referenzen URLs, die die Zeichenketten `github.com` und `/commit/` enthalten. Diese verweisen direkt auf einen einzelnen Commit, der die Schwachstelle behebt. Dies ist die direkteste Form der Referenz, da sie unmittelbar die Code-Änderungen zeigt.

Blob-Referenzen URLs, die die Zeichenketten `github.com` und `/blob/` enthalten. Diese verweisen auf eine spezifische Datei in einem bestimmten Zustand (Commit oder Branch). Aus solchen Referenzen lässt sich der zugehörige Commit extrahieren, in dem die Datei in diesem Zustand vorliegt.

Commit-Listen von Pull Requests URLs, die `/commits/` und `github.com` enthalten, verweisen auf die Liste aller Commits innerhalb eines Pull-Requests.

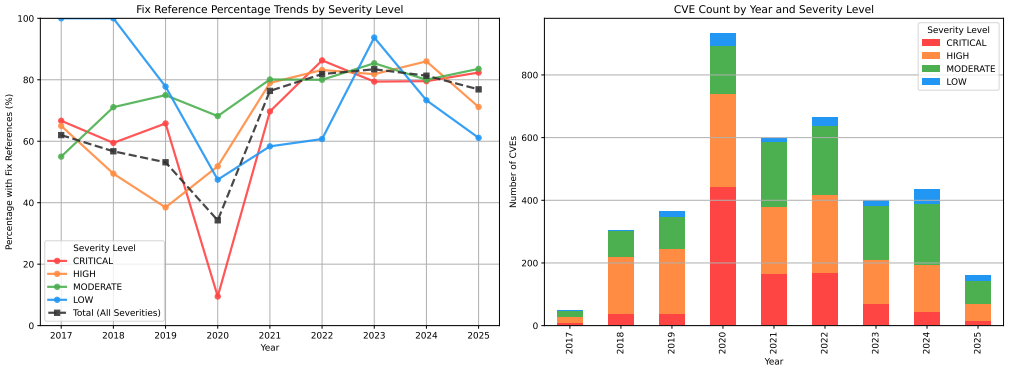

Abbildung 4.4.: Analyse bekannter Schwachstellen im NPM-Ökosystem (durchgeführt 27. April)

Ein Pull-Request repräsentiert einen formalisierten Änderungsvorschlag, der typischerweise zur Diskussion, Überprüfung und schließlich zur Integration von Code-Änderungen in das Hauptrepository dient. Ein Pull-Request bündelt eine Reihe von Code-Änderungen. Dabei kann einer oder mehrere dieser Commits die eigentliche Behebung einer Schwachstelle enthalten. Die Analyse dieser Commit-Listen ermöglicht es, die relevanten Code-Änderungen zu identifizieren, die zur Behebung einer Sicherheitslücke geführt haben.

Direkte Pull-Request-Referenzen URLs, die die Zeichenketten `github.com` und `/pull/` enthalten. Diese verweisen auf den Pull-Request als Ganzes, der zur Behebung der Schwachstelle erstellt wurde. Über diese Referenz lassen sich nicht nur die Metadaten des Pull Requests (z. B. Titel, Beschreibung, Erstellungsdatum, beteiligte EntwicklerInnen) einsehen, sondern auch sämtliche zugehörige Commits extrahieren.

Issue-Referenzen mit assoziierten Pull-Requests URLs, die die Zeichenketten `/issues/` und `github.com` enthalten, verweisen auf GitHub-Issues, in denen Sicherheitslücken diskutiert und dokumentiert werden. Da Issues oft mit Pull-Requests verknüpft sind, die das Problem beheben, lassen sich über diese Verknüpfung die tatsächlichen Fix-Commits identifizieren.

Die Verteilung der identifizierten Referenztypen ist in Abbildung 4.3 dargestellt. Dieser leicht erweiterte Ansatz identifiziert mehr Referenzen als die von [3] vorgestellte Methodik. Während direkte Commit-Referenzen lediglich bei 1.763 CVEs im NPM-Ökosystem gefunden werden, können durch die Berücksichtigung zusätzlicher Referenztypen 765 weitere CVEs mit Fix-Informationen angereichert werden. Dies wird erreicht, ohne die Komplexität des in [1] vorgestellten Prozesses mit seiner tiefen Analyse von Commit-Messages und CVE-Beschreibungen zu übernehmen.

Die Effektivität dieses Ansatzes wird durch die zunehmende Professionalisierung der Schwachstellendokumentation im JavaScript-Ökosystem zusätzlich gestützt. Wie in Abbildung 4.4 ersichtlich, ist der Anteil neuer CVE-Einträge mit Fix-Referenzen zwischen 2017 und 2021 kontinuierlich gestiegen und hat sich seitdem auf einem stabilen Niveau von etwa 80 % eingependelt. Besonders kritische Schwachstellen (CVSS $\geq$ 9.0) weisen im Jahr 2025 sogar zu über 80 % entsprechende Referenzen auf. Diese positive Entwicklung, mit der Ausnahme des Jahres 2020, unterstreicht die Praktikabilität des gewählten referenzbasierten Ansatzes. Die hohe Verfügbarkeit von Fix-Referenzen, insbesondere bei sicherheitskritischen Schwachstellen, rechtfertigt die Konzentration auf diese Informationsquelle, da sie für die überwiegende Mehrheit der relevanten Fälle verlässliche Analysedaten bereitstellt.

In der Praxis enthalten Security-Patches allerdings nicht ausschließlich Änderungen, die isoliert die Schwachstellenbehandlung betreffen. Entwickler können etwa Änderungen an Dokumentationsdateien oder zusätzlich Refactoring-Maßnahmen, die nicht nur in direkter Beziehung zur Schwachstelle stehen, durchführen. Dennoch gilt: Jeder Security-Patch **muss** mindestens eine Änderung enthalten, die in kausalem Zusammenhang zur Schwachstelle steht, sonst wäre die Referenz im CVE-Eintrag fehlplatziert. Auf solche Fehlplatzierungen wird in dieser Arbeit nicht weiter eingegangen.

Diese Arbeit klassifiziert alle Änderungen in identifizierten Security-Patches als vulnerabel, um False-Negatives zu minimieren. Diese Strategie akzeptiert bewusst eine Erhöhung der False-Positives.

4.4. Identifikation des direkt vulnerablen Quellcodes

Die Identifikation des direkt vulnerablen Quellcodes erfolgt nach der Identifikation der Security Patches und ist in Abbildung 4.1 im Schritt ❻ zu finden. Dabei kommt die in Abschnitt 2.3 beschriebene Heuristik zur Anwendung: Jeder im Security-Patch geänderte Quellcode wird als vulnerabel klassifiziert, während jeder nicht geänderte Quellcode als **nicht** vulnerabel eingestuft wird.

Im ersten Schritt werden die identifizierten Security Patches detailliert analysiert. Hierzu werden die konkreten Code-Änderungen über die GitHub REST API abgefragt [19]. Die API-Antwort enthält den vollständigen *Git-Diff* des jeweiligen Commits, der die exakten Änderungen zwischen der vulnerablen und der gepatchten Version dokumentiert.

Mittels eines regulären Ausdrucks werden aus diesem *Diff* die veränderten Zeilenbereiche extrahiert. Für jede Änderung werden die Start- und Endzeile sowie der zugehörige Dateiname erfasst. Diese Informationen definieren präzise, welche Codebereiche durch den Security Patch modifiziert wurden und folglich als vulnerabel anzusehen sind. Darüber hinaus enthält die API-Antwort den Parent-Commit-Hash. Dieser Parent-Commit-Hash repräsentiert somit den Stand des Repositories vor der Behebung der Schwachstelle und ermöglicht die exakte

Rekonstruktion der vulnerablen Codeversion.

4.5. Aufbau des Callgraphen

Nach der Identifikation des direkt vulnerablen Quellcodes erfolgt im nächsten Schritt der Aufbau eines Callgraphen, der die Grundlage für die weitere Analyse der Erreichbarkeit bildet. Dieser Prozess beginnt mit dem Download der vulnerablen Repository-Version (❼) und umfasst anschließend die statische Analyse des Codes mittels des Tools jelly (`https://github.com/cs-au-dk/jelly`) (❽, ❾) [35].

Zunächst wird das mit einer Schwachstelle behaftete Repository von GitHub in der vulnerablen Version heruntergeladen. Dies erfolgt mittels Git unter Verwendung des zuvor identifizierten Parent-Commit-Hash, der die Version des Repositories unmittelbar vor der Behebung der Schwachstelle repräsentiert. Da alle analysierten Referenzen auf `github.com` verweisen, ist die Identifikation des korrekten Repositories trivial: Die Repository-URL lässt sich direkt aus der Referenz-URL extrahieren. Durch den Checkout des Parent-Commit-Hash wird der exakte Zustand des Codes wiederhergestellt, wie er vor der Behebung der Schwachstelle existierte. Dies gewährleistet, dass die nachfolgende Analyse auf dem tatsächlich vulnerablen Code basiert und nicht auf einer bereits gepatchten Version. Der gesamte Prozess stellt sicher, dass für jede analysierte Schwachstelle der präzise Zustand des Quellcodes vorliegt, der die Sicherheitslücke enthält.

Für die statische Analyse wird das Tool `jelly` verwendet, welches von Anders Møller und Oskar Haarklou Veileborg, speziell für die Analyse von JavaScript- und TypeScript-Code entwickelt wurde. Es eignet sich für die Konstruktion eines Callgraphen, die statische Identifikation von genutzen Bibliotheken sowie für die Analyse, ob eine Schwachstelle exploitierbar ist [8]. Laut der README-Datei des Projekts fußt es auf den wissenschaftlichen Arbeiten [35], [29] [28], [14] und [32].

Listing 4.1: Beispiel eines jelly Callgraphen

```
{
    "entries": ["app.js", "lib.js"],
    "files": ["app.js","lib.js"],
    "functions": {
        "0": "1:1:1:4:2",
        "1": "1:6:1:9:2",
    },
    "calls": {
        "4": "0:1:13:1:29",
        "5": "0:3:1:3:19",
    },
    "fun2fun": [[2, 1]],
```

```
13     "call2fun": [[4, 5]],
14 }
```

jelly konstruiert einen umfassenden Callgraphen des analysierten Quellcodes, der nicht nur die tatsächlichen Funktionsaufrufe erfasst, sondern auch die Definitionsbereiche einzelner Funktionen sowie deren zugehörige Dateien dokumentiert. Der generierte Callgraph wird in einem strukturierten JSON-Format von dem Werkzeug jelly bereitgestellt. Ein exemplarischer Callgraph ist in Listing 4.1 dargestellt.

Die Struktur des Callgraphen umfasst mehrere wesentliche Komponenten: Das Array `files` listet alle analysierten Dateien auf, wobei jedem Dateinamen ein Index zugeordnet wird. Das Objekt `functions` enthält für jede identifizierte Funktion einen Eintrag im Format `"FunctionIndex":"FileIndex:StartLine:StartColumn:EndLine:EndColumn"`, der die Position und Ausdehnung der Funktion präzise definiert. Das Objekt `calls` dokumentiert alle Funktionsaufrufe im Code in analoger Notation. Die Arrays `fun2fun` und `call2fun` beschreiben schließlich die Aufrufbeziehungen zwischen Funktionen respektive zwischen Aufrufstellen und aufgerufenen Funktionen.

Die zentrale Herausforderung besteht nun darin, die zuvor identifizierten vulnerablen Codebereiche, die durch Tupel der Form `(Dateiname, Startzeile, Endzeile)` beschrieben werden, mit den Funktionen im Callgraphen zu verknüpfen. Hierzu wird in Schritt ❾ für jeden vulnerablen Codebereich mittels Dateiname, Startzeile und Endzeile geprüft, welche Funktionsindizes im Callgraphen eine räumliche Überlappung mit diesem Bereich aufweisen. Eine Funktion wird als überlappend klassifiziert, wenn ihr Definitionsbereich mindestens eine Zeile des vulnerablen Codebereichs umfasst.

Dieser Abgleich ermöglicht eine wichtige Abstraktion: Von der zeilenbasierten Repräsentation vulnerabler Codebereiche gelangt man zur funktionsbasierten Abstraktion mittels der Funktionsindizes im Callgraphen (`FunctionIndex`). Alle Funktionen, die eine Überlappung mit einem vulnerablen Codebereich aufweisen, werden als vulnerabel markiert.

Es ist zu beachten, dass diese Klassifikation rein statisch erfolgt und keine Aussage über die tatsächliche Erreichbarkeit des vulnerablen Codes zur Laufzeit trifft. Es wird nicht überprüft, ob der vulnerable Code überhaupt erreicht werden kann. Selbst wenn vulnerabler Code innerhalb einer nie ausgeführten Bedingung steht (beispielsweise `if (false) vulnerable code`), wird die umschließende Funktion dennoch als vulnerabel klassifiziert. Dies ist, wie bereits beschrieben, eine grundsätzliche Limitierung der statischen Code-Analyse.

4.6. Taint-Analyse zur Identifikation indirekt betroffener Funktionen

Nachdem die direkt vulnerablen Funktionen identifiziert wurden, erfolgt im nächsten Schritt (10) eine Taint-Analyse, um auch indirekt betroffene Funktionen zu ermitteln. Die Taint-Analyse nutzt die im Callgraphen erfassten Aufrufbeziehungen (`fun2fun`), um alle Funktionen zu identifizieren, die direkt oder transitiv vulnerable Funktionen aufrufen.

Hierbei wird das Konzept der Taint-Propagierung angewendet: Eine Funktion gilt als indirekt vulnerabel, wenn sie eine direkt vulnerable Funktion aufruft oder eine andere indirekt vulnerable Funktion invoziert. Durch eine rekursive Traversierung des Callgraphen werden somit alle Funktionen markiert, über die ein Ausführungspfad zu einer direkt vulnerablen Funktion existiert [25, 35].

Es ist hervorzuheben, dass diese Analyse ausschließlich auf der Aufrufstruktur basiert und nicht berücksichtigt, ob tatsächlich Daten von einer aufrufenden Funktion zu einer aufgerufenen Funktion fließen. Die Taint-Analyse klassifiziert eine Funktion bereits dann als indirekt vulnerabel, wenn strukturell ein Aufrufpfad existiert, unabhängig davon, ob vulnerable Daten über diesen Pfad propagiert werden.

Der verwendete Algorithmus ist durch Pseudocode in Listing 4.2 dargestellt.

Listing 4.2: Pseudocode der Taint-Analyse

```
function taintAnalysis(callgraph,
    directlyVulnerableFunctions):
tainted = set(directlyVulnerableFunctions)
worklist = list(directlyVulnerableFunctions)
while worklist is not empty:
    currentFunction = worklist.pop()

    for caller in getCallers(callgraph, currentFunction):
        if caller not in tainted:
            tainted.add(caller)
            worklist.add(caller)

return tainted
```

Dieser Algorithmus startet mit der Menge der direkt vulnerablen Funktionen und propagiert die Taint-Markierung schrittweise rückwärts durch den Callgraphen. Für jede markierte Funktion werden alle aufrufenden Funktionen identifiziert und ebenfalls als vulnerabel markiert, sofern sie nicht bereits in der Taint-Menge enthalten sind. Dieser Prozess wird iterativ fortgesetzt, bis keine weiteren Funktionen mehr hinzugefügt werden können. Das Ergebnis ist eine vollständige Erfassung aller direkt und indirekt vulnerablen Funktionen, die die Angriffsoberfläche des analysierten Pakets definieren.

4.7. Zuweisung von Funktionsindizes zu global eindeutigen Symbolnamen

Der im Callgraphen verwendete Funktionsindex ist ausschließlich innerhalb eines spezifischen Callgraphen eindeutig. Für den Aufbau einer VEX-Datenbank und einer Datenbank vulnerabler Symbole sind allerdings möglichst global eindeutige Identifikatoren erforderlich, die das Tupel `(PURL, eindeutiger Symbolname im Paket)` repräsentieren. Für die praktische Anwendbarkeit der VEX-Datenbank sind insbesondere die exportierten Funktionen eines Pakets relevant, da nur diese von Konsumenten des Pakets tatsächlich verwendet werden können. Uninteressant sind hingegen interne Funktionen wie beispielsweise Test-Funktionen, die zwar vulnerablen Code ausführen, aber nicht Teil der öffentlichen API des analysierten Pakets sind. Nur wenn vulnerable Funktionen über die Paket-API erreichbar sind, können sie sich auf abhängige Pakete auswirken.

NPM-jelly bietet eine Funktion zur Ermittlung der exportierten Funktionen eines Pakets, die diese mittels sogenannter *Access Paths* darstellt [32]. Ein exemplarischer Access Path hat folgende Form:

`/path/to/lib.js:2:7:5:4: <vexing-npm/lib>.default.fn`

Dieser Access Path kodiert mehrere Informationen über den Zugriffspfad zu einer exportierten Funktion. Die Komponenten lassen sich wie folgt interpretieren: Der erste Teil (`/path/to/lib.js:2:7:5:4`) beschreibt die präzise Position im Quellcode. Dabei bezeichnet`/path/to/lib.js` die Datei, in der die Funktion definiert ist. Die nachfolgenden Zahlenpaare `2:7` und `5:4` spezifizieren die Zeilen- und Spaltenpositionen im Code, typischerweise den Start- und Endpunkt der Funktionsdefinition.

Der zweite Teil (`<vexing-npm/lib>.default.fn`) repräsentiert den semantischen Zugriffspfad auf die Funktion aus Sicht eines Paketnutzers. Der Ausdruck `<vexing-npm/lib>` kennzeichnet das importierte Modul, wobei `vexing-npm` der Paketname und `lib` das spezifische Submodul ist. Die Notation `.default.fn` beschreibt die Kette von Eigenschaftszugriffen: Zunächst wird auf den Default-Export des Moduls zugegriffen (`.default`), von diesem aus dann auf die Eigenschaft `fn`.

Ein Entwickler würde auf diese Funktion wie in Listing 4.3 dargestellt zugreifen.

Listing 4.3: Zugriff auf das AccessPath Beispiel

```
1 import * as lib from "vexing-npm/lib";
2 lib.default.fn();
3
4 // oder alterniv durch Nutzen des default exports
5 import lib from "vexing-npm/lib";
6 lib.fn();
7
```

```
8  // in CommonJS
9  const lib = require("vexing-npm/lib");
10 lib.default.fn();
```

Access Paths werden in der Arbeit [32] vorgestellt und folgen der in Abbildung 4.7 beschriebenen formalen Gramatik, die verschiedene Arten von Zugriffen auf Module und Funktionen beschreibt:

$$
\begin{aligned}
\mathit{AccessPath} ::= \quad & \langle \mathit{ImportPath} \rangle \\
| \quad & \mathrm{Fun}(f, l) \\
| \quad & \mathrm{Fun}(f, l).\mathrm{Param}[i] \\
| \quad & \mathit{AccessPath}.\mathit{Prop} \\
| \quad & \mathit{AccessPath}(\mathcal{P}(\mathit{AccessPath}), \ldots) \\
| \quad & \mathrm{U}
\end{aligned}
$$

Abbildung 4.5.: Gramatik-Definition des *AccessPath*-Patterns [32, 35]

Die einzelnen Bestandteile haben nach den Autoren der Arbeiten [32, 35] folgende Bedeutung:

- $\langle m \rangle$ kennzeichnet das Laden eines Moduls m, beispielsweise durch `require("m")`.
- $\mathrm{Fun}\langle f, l \rangle$ bezeichnet eine Funktionsdefinition in Datei f in Zeile l.
- $\mathrm{Fun}\langle f, l \rangle.\mathit{Param}[i]$ bezeichnet den i-ten Parameter der Funktionsdefinition in Datei f in Zeile l.
- $\mathit{ap}.P$ bezeichnet Zugriffe auf Properties mit Namen P von Objekten, die durch den Access Path ap beschrieben werden.
- $\mathit{ap}(S_1, \ldots, S_n)$ bezeichnet Aufrufe von Funktionen, die durch ap beschrieben werden, wobei das i-te Argument durch einen Access Path in S_i beschrieben wird.
- U wird für Ausdrücke verwendet, denen die statische Analyse keinen anderen Access Path zuweisen kann.

Durch die Verwendung von Access Paths gelangt man auf eine höhere Abstraktionsebene, die eine global eindeutige Identifikation ermöglicht. Das Tupel `(PURL, AccessPath)` ist eindeutig und kann effizient referenziert und in der VEX-Datenbank abgespeichert werden. Die PURL identifiziert das Paket und die spezifische Version eindeutig, während der Access Path das konkrete Symbol innerhalb des Pakets beschreibt.

Die Zuweisung von Funktionsindizes aus dem Callgraphen zu exportierten API-Funktionen mittels Access Paths erfolgt über die Dateinamen und Zeilennummern. Da sowohl der Callgraph als auch die Access Paths Positionsinformationen in der Form `(Dateiname, Zeile)` enthalten, lässt sich die Zuordnung direkt vornehmen: Eine exportierte Funktion mit einem bestimmten Access Path kann den im Callgraphen als vulnerabel markierten Funktionsindizes zugeordnet werden, indem die jeweiligen Positionsangaben abgeglichen werden. Dies ermöglicht eine Abbildung von der internen Callgraph-Repräsentation zu global eindeutigen, exportierten Symbolen.

Die exportierten Symbole werden in das Datenbankschema, dargestellt in Listing 4.4, transformiert und abgespeichert.

Listing 4.4: Datenbankschema für vulnerable Symbole in der VEX-Datenbank

```
CREATE TABLE
  vuln_symbols (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    signature TEXT,
    purl TEXT,
    cve_id TEXT,
    filename TEXT,
    start_line INTEGER,
    end_line INTEGER,
    start_column INTEGER,
    end_column INTEGER,
    evidence TEXT,
    calls_id INTEGER,
    UNIQUE (signature, purl, cve_id)
  )
```

4.8. Vererbungsanalyse

Die bisherige Analyse konzentriert sich ausschließlich auf die Identifizierung vulnerabler Symbole innerhalb des Pakets, in dem die Schwachstelle gemeldet wurde. Im Rahmen der Vererbungsanalyse wird nun untersucht, wie sich diese Schwachstelle auf sogenannte Downstream-Komponenten auswirkt, also auf Komponenten, die von der vulnerablen Bibliothek abhängen. Dabei werden deren vulnerable, exportierte Symbole identifiziert und ebenfalls in der VEX-Datenbank persistiert.

Abbildung 4.6 illustriert einen exemplarischen Abhängigkeitsgraphen. Paket A wurde während der Paketselektionsphase aufgrund hoher Downloadzahlen für die automatische VEX-Erzeugung ausgewählt. Der `npm audit`-Scan ergibt, dass sich eine Schwachstelle in der transitiven Abhängigkeit G befindet. In den Schritten ❶ bis ❿ wurden die konkreten vulnerablen Symbole von G identifiziert. In den Schritten ⓫ bis ⓬ wird nun analysiert, wie sich die Schwachstelle entlang

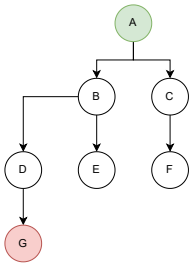

Abbildung 4.6.: Beispielabhängigkeitsgraph. Analysiert wird Paket A (ausgewählt während der Paket-Auswahl aufgrund hoher Downloadzahlen). Eine Schwachstelle befindet sich in G.

der Abhängigkeitskette vererbt, zunächst auf D, bei fortgesetzter Nutzung der vulnerablen Funktionen auf B und schließlich auf das ursprünglich analysierte Paket A.

4.8.1. Iterative Analyse der Abhängigkeitspfade

Zunächst werden mittels `npm ls --all`, das den vollständigen Abhängigkeitsbaum inklusive aller transitiven Pakete ausgibt, alle Pfade ermittelt, über die eine vulnerable Abhängigkeit im Abhängigkeitsbaum erreicht werden kann. Ein Paket kann über multiple Pfade von verschiedenen Abhängigkeiten aus eingebunden werden, was zu unterschiedlichen Vererbungsszenarien führt. Für jeden identifizierten Pfad wird anschließend eine separate Vererbungsanalyse durchgeführt, um festzustellen, ob die Schwachstelle tatsächlich über diesen spezifischen Pfad propagiert wird.

Die Analyse eines Pfades erfolgt iterativ, beginnend beim direkten Downstream der vulnerablen Komponente und fortschreitend entlang des Pfades entgegen der Kantenrichtung bis zum Root-Paket. Für jede Abhängigkeit im Pfad wird untersucht, ob sie die als vulnerabel identifizierten Funktionen ihrer Abhängigkeit tatsächlich nutzt. Im Gegensatz zur Analyse der ursprünglich vulnerablen Bibliothek ist hier kein `git clone` erforderlich, da alle Abhängigkeiten bereits im `node_modules`-Ordner des Root-Pakets vorliegen. Dieser wurde durch Ausführung des Befehls `npm install --no-scripts --ignore-platform --omit=dev` während der initialen Schwachstellenanalyse (Schritt ❷) befüllt und enthält somit den vollständigen Abhängigkeitsbaum in der korrekten Versionierung.

Für jede zu untersuchende Abhängigkeit im Pfad wird jelly genutzt um die extern genutzen Bibliotheksfunktionen zu identifizieren. Hierzu wird jelly mit der `---api-usage`-Flag ausgeführt. Die Ausgabe enthält dabei präzise Informationen darüber, welche Funktionen aus welchen Abhängigkeiten an welchen Stellen im Code aufgerufen werden, einschließlich Dateinamen, Zeilen- und Spaltenan-

gaben. Anschließend erfolgt ein Abgleich zwischen den genutzten API-Funktionen und den zuvor identifizierten vulnerablen Symbolen der Abhängigkeit. Die verwendeten Access Paths ermöglichen einen direkten Vergleich, da sowohl die vulnerablen Symbole als auch die API-Nutzung in diesem Format repräsentiert sind.

Hierbei werden zwei Szenarien unterschieden. Wenn keine der als vulnerabel identifizierten Funktionen durch die analysierende Abhängigkeit genutzt wird, ist der gesamte Pfad nicht von dieser Schwachstelle betroffen. Dies wird in der Datenbank persistiert, und die Analyse dieses Pfades wird beendet. Dieses Ergebnis wird als VEX-Statement mit dem Status `not_affected` dokumentiert, wobei als Begründung angegeben wird, dass die vulnerablen Funktionen nicht aufgerufen werden. Wenn hingegen eine oder mehrere vulnerable Funktionen tatsächlich genutzt werden, vererbt sich die Schwachstelle potenziell auf die analysierende Abhängigkeit, und die Analyse wird mit der Identifikation der neu betroffenen Symbole fortgesetzt.

4.8.2. Propagierung und Identifikation vulnerabler Symbole

Wenn die Nutzung vulnerabler Funktionen festgestellt wurde, erfolgt eine analoge Analyse wie bei der ursprünglich vulnerablen Bibliothek. Die konkreten Zeilen, in denen vulnerable Funktionen aufgerufen werden, werden als vulnerable Codebereiche markiert. Die Informationen über die Aufrufstellen sind im NPM-jelly API-Usage-Report enthalten und umfassen Dateinamen sowie Zeilen- und Spaltenangaben. Anschließend wird für die aktuell analysierte Bibliothek ein vollständiger Callgraph aufgebaut.

Daraufhin wird analog zu den in den Abschnitten 4.5 und 4.7 beschriebenen Schritten verfahren. Zunächst werden durch Abgleich der als vulnerabel markierten Zeilen (Aufrufstellen) mit den Funktionsdefinitionen im Callgraphen alle Funktionen identifiziert, deren Definitionsbereich mit den vulnerablen Zeilen überlappt. Mittels der in Abschnitt 4.5 beschriebenen Taint-Analyse werden dann alle Funktionen identifiziert, die direkt oder transitiv die vulnerablen Funktionen aufrufen. Schließlich wird ermittelt, welche der als vulnerabel markierten Funktionen Teil der exportierten API der Bibliothek sind. Dies erfolgt analog zum in Abschnitt 4.7 beschriebenen Mapping von Funktionsindizes zu Access Paths.

Die so identifizierten, exportierten vulnerablen Symbole bilden die Grundlage für die Analyse des nächsten Elements im Abhängigkeitspfad. Dieser Prozess wird iterativ fortgesetzt, bis entweder das Root-Paket erreicht oder festgestellt wird, dass die Schwachstelle nicht weiter propagiert wird. Im ersten Fall wird ein VEX-Statement mit dem Status `affected` für das Root-Paket erzeugt, im zweiten Fall endet die Propagierung an der Stelle, an der keine Nutzung vulnerabler Funktionen mehr festgestellt werden kann. Dies lässt sich auf den VEX-Status `not_affected` übersetzen.

Durch diese systematische Vererbungsanalyse entlang aller Abhängigkeitspfade entsteht ein vollständiges Bild darüber, welche Schwachstellen transitiver Ab-

hängigkeiten tatsächlich das Root-Paket betreffen und über welche API-Pfade sie erreichbar sind.

Ein vereinfachtes Beispiel verdeutlicht diesen Prozess: Angenommen, Paket `my-web-framework@2.1.0` wurde für die Analyse ausgewählt und besitzt folgenden vereinfachten Abhängigkeitgraphen: my-web-framework → express → cookie. Der `npm audit`-Scan identifiziert eine Schwachstelle in der transitiven Abhängigkeit `cookie@0.5.0`. Die Schritte ❶ bis ❿ ermitteln, dass die Funktion `cookie.parse()` vulnerabel ist. Nun beginnt die Vererbungsanalyse: Zunächst wird untersucht, ob `express` die vulnerable Funktion `cookie.parse()` direkt oder indirekt aufruft. Falls nein, ist `express` nicht betroffen, und es wird ein `not_affected` VEX-Statement erstellt. Falls jedoch `express` diese Funktion nutzt, wird ermittelt, welche Funktionen von `express`, beispielsweise `express.cookieParser()`, den vulnerablen Code verwenden. Diese Funktionen werden als indirekt vulnerabel markiert.

Die Analyse endet jedoch nicht bei `express`. Die Vererbungsanalyse prüft nun, ob `my-web-framework` die zuvor als vulnerabel identifizierte Funktion `express.cookieParser()` aufruft. Falls dies der Fall ist, wird wiederum ermittelt, welche exportierten Funktionen von `my-web-framework`, beispielsweise `framework.handleRequest()`, betroffen sind. Erst wenn keine weitere Propagierung festgestellt wird oder das ursprünglich analysierte Paket erreicht ist, endet die Vererbungsanalyse. Alle identifizierten vulnerablen Symbole entlang dieser Kette werden in der VEX-Datenbank dokumentiert, wodurch Entwickler präzise nachvollziehen können, ob und über welche API-Pfade die Schwachstelle ihr Projekt betrifft.

4.9. Umgang mit dynamisch erzeugtem Code

Wie in Kapitel 2.4 beschrieben, können dynamische JavaScript-Konstrukte die Ergebnisse einer statischen Code-Analyse beeinträchtigen. Die dort eingeführten zwei zentralen Problemklassen, die Identifikation aller existierenden Funktionen (Knotenmenge) und die Bestimmung aller Aufrufpfade (Kantenmenge), sind durch dynamische Sprachkonstrukte fundamental betroffen. Während diese Limitierungen statischer Analyse nicht vollständig umgangen werden können, ist es möglich, die Präsenz solcher dynamischer Konstrukte zu erkennen und entsprechend zu dokumentieren. Dies ermöglicht eine Bewertung der Verlässlichkeit der Analyseergebnisse.

Zu diesem Zweck wurden Semgrep-Regeln entwickelt, die systematisch nach problematischen dynamischen Konstrukten im analysierten Code suchen. Semgrep ist ein statisches Analysetool, das regelbasierte Mustersuche in Quellcode ermöglicht und sich besonders für die Identifikation spezifischer Code-Muster eignet. Die entwickelten Regeln adressieren beide Problemklassen und decken drei Kategorien dynamischer Konstrukte ab, die die Qualität der Callgraph-Konstruktion beeinträchtigen können.

Die erste Regel identifiziert dynamische `require`-Aufrufe, bei denen der Modulname nicht als String-Literal vorliegt, sondern zur Laufzeit berechnet wird. Solche Konstrukte der Form `require(variableName)` betreffen beide Problemklassen gleichermaßen, da weder alle existierenden Funktionen, noch deren Aufrufbeziehung erkannt werden können. Der tatsächlich geladene Modulpfad wird erst zur Laufzeit bestimmt, wodurch die Knoten- und Kantenmenge des Callgraphen unvollständig bleibt. Die Regel unterscheidet explizit zwischen statischen Aufrufen wie `require("modulename")` und dynamischen Varianten.

Die zweite Regel erkennt Monkey-Patching-Operationen, bei denen importierte Bibliotheken zur Laufzeit modifiziert werden. Durch Zuweisungen der Form `$LIB.$PROP = $VALUE` können Funktionen überschrieben oder neue Eigenschaften hinzugefügt werden. Zum einen kann die Knotenmenge verändert werden, wenn neue Funktionen hinzugefügt werden, zum anderen werden bestehende Aufrufbeziehungen umgeleitet, wodurch die Kantenmenge nicht mehr der statisch ermittelten Struktur entspricht. Die Regel erfasst verschiedene Import- und Require-Varianten und identifiziert alle Zuweisungen auf importierte Module, unabhängig davon, ob diese mittels ES6-Imports oder CommonJS-`require` eingebunden wurden.

Die dritte Regel detektiert dynamische Code-Ausführung mittels `eval`, `new Function` oder verwandter Konstrukte. Diese Mechanismen erlauben die Ausführung von zur Laufzeit konstruiertem Code in String-Form. Durch `eval` können beliebige neue Funktionen erzeugt werden (Knotenmenge), und es können beliebige neue Aufrufbeziehungen etabliert werden (Kantenmenge), ohne dass diese im statischen Code erkennbar wären. Die Regel deckt verschiedene Varianten dieser Konstrukte ab, einschließlich der Verwendung in unterschiedlichen globalen Kontexten wie `window`, `global` oder `globalThis`.

Die Ergebnisse dieser Semgrep-Analyse werden für jedes untersuchte Paket dokumentiert und fließen in die Berechnung einer Konfidenzmetrik ein, die angibt, wie verlässlich die Resultate der statischen Analyse für das jeweilige Paket sind. Pakete ohne dynamische Konstrukte erhalten dabei eine hohe Konfidenz. Es ist jedoch zu beachten, dass die Regeln keinesfalls vollständig sind und nicht alle möglichen Probleme erfassen können. Die Konfidenzinformation wird in den VEX-Statements mitgeführt und erlaubt es, die Aussagekraft der Analyse einzuschätzen und bei Bedarf zusätzliche manuelle Überprüfungen vorzunehmen.

Die vorgestellte entwickelte Methodik ermöglicht eine detaillierte Analyse der Vererbung von Schwachstellen über transitive Abhängigkeiten hinweg im NPM-Ökosystem. Durch die systematische Extraktion von Fix-Referenzen (Abschnitt 4.3) und die statische Analyse mittels Callgraph wird nicht nur die direkte Betroffenheit eines Pakets ermittelt, sondern auch die Ausbreitung von Schwachstellen über Abhängigkeitsketten nachverfolgt. Die Callgraph-basierte Taint-Analyse (Abschnitt 2.1) identifiziert dabei vulnerable Pfade und zeigt auf, wie sich Schwachstellen von einer Bibliothek auf abhängige Projekte übertragen. Dies liefert empirische Daten, um Erkenntnisse zu den Propagierungsmustern im NPM-Ökosystem zu gewinnen.

5. Ergebnisse

Die folgenden Abschnitte präsentieren die konkreten Ergebnisse der in Abbbildung 4.1 dargestellten Methodik und zeigen, inwiefern die Forschungsfragen durch die gesammelten Daten beantwortet werden können.

Für die Analyse wurden 178.775 `(Paket, Version)`-Tupel, wie in Kapitel 4.1 beschrieben, ausgewählt. Von diesen Paketen ließen sich 172.177 mittels `npm install` installieren. Bei den anderen Paketen kam es zu Fehlermeldungen während des Installationsprozesses. Es lassen sich folgende zentrale Erkenntnisse ableiten, die einen umfassenden Einblick in die Schwachstellenlandschaft des NPM-Ökosystems sowie die Effektivität und Notwendigkeit der entwickelten Methodik geben.

5.1. Reduktionspotential metadatenbasierter Schwachstellenscanner (RQ1)

Die in dieser Arbeit verfolgte Methodik analysiert die NPM-Pakete absteigend nach ihrer Popularität, beginnend bei den am häufigsten heruntergeladenen Paketen. Dieser Ansatz dient primär der Untersuchung der Propagierung von Schwachstellen über Abhängigkeitsketten, wie in der ersten Teilfrage der Forschungsfrage formuliert. Da diese Pakete als grundlegende Bausteine des NPM-Ökosystems fungieren, kommt ihnen eine zentrale Bedeutung für die Sicherheit des gesamten Ökosystems zu. Ihre weite Verbreitung bedeutet, dass Schwachstellen in diesen Paketen ein hohes Potenzial für eine transitive Verbreitung aufweisen. Gestützt wird diese Annahme zusätzlich durch die Downloadverteilung, wie in Abbildung 4.2 dargestellt.

Die Abhängigkeitsstruktur der untersuchten NPM-Paketen variiert stark und zeigt interessante Zusammenhänge mit der Popularität der Pakete. Abbildung 5.1 visualisiert eine schwach negative Korrelation zwischen der Paket-Popularität, gemessen anhand der Downloadzahlen, und der Größe des Abhängigkeitsgraphen. Dieses Phänomen lässt sich leicht erklären: Wenn ein Paket A, welches von Paket B abhängt, installiert wird, wird bei jeder Installation von A auch B heruntergeladen. Die Downloadzahl von B ist daher mindestens so hoch wie die von A, in der Regel sogar höher, da B möglicherweise von mehreren anderen Paketen als Abhängigkeit verwendet wird. Daraus folgt, dass das am häufigsten heruntergeladene Paket im gesamten Ökosystem keine Abhängigkeiten besitzen kann, da es andernfalls nicht die höchste Downloadzahl aufweisen würde. Seine eigenen Abhängigkeiten hätten mindestens dieselbe oder eine höhere Downloadzahl. Diese strukturelle Eigenschaft führt dazu, dass populäre Pakete, gemessen an

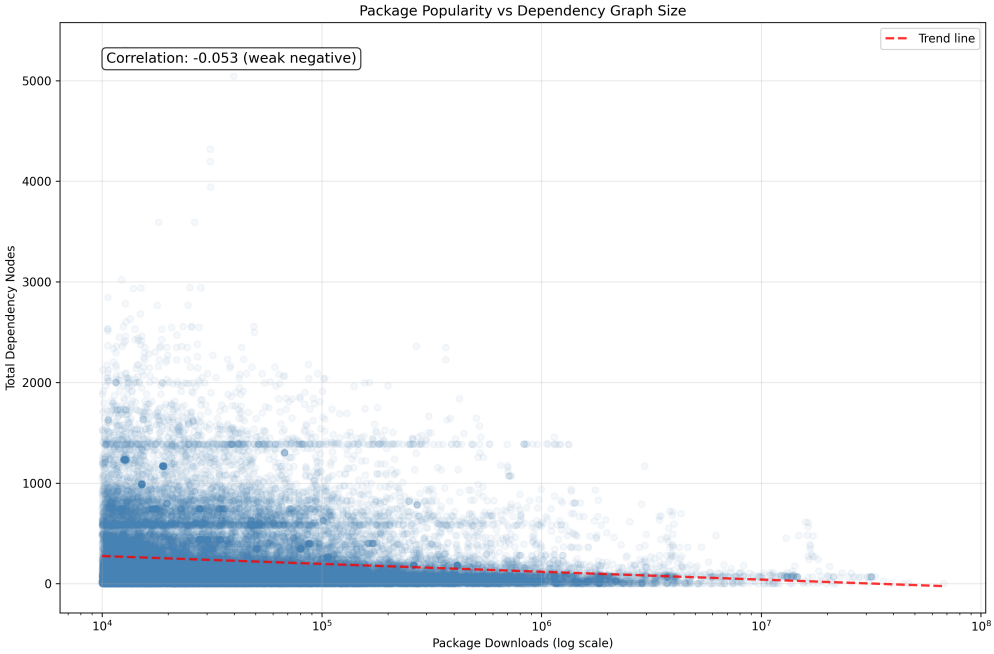

Abbildung 5.1.: Korrelation zwischen Downloadzahlen und Größe des Abhängigkeitsgraphen

den Downloadzahlen, tendenziell weniger Abhängigkeiten aufweisen als weniger verbreitete Pakete.

Da diese Arbeit beginnend bei den populärsten Paketen absteigend NPM-Pakete analysiert, können die erhobenen Daten keine repräsentative Aussage über eine durchschnittliche JavaScript-Anwendung treffen. Es ist anzunehmen, dass typische Anwendungspakete, die nicht primär als wiederverwendbare Bibliotheken konzipiert sind, größere und komplexere Abhängigkeitsbäume aufweisen. Die hier untersuchten Pakete stellen die fundamentalen Bausteine des NPM-Ökosystems dar, die aufgrund ihrer breiten Verwendung eine besondere Rolle für die Sicherheit des gesamten Ökosystems spielen.

Die untersuchten 172.177 Pakete weisen im Durchschnitt 97 Abhängigkeiten auf, wobei der Median mit nur 12 Abhängigkeiten deutlich niedriger liegt. Diese Diskrepanz zwischen Mittelwert und Median deutet auf eine rechtsschiefe Verteilung hin, die in Abbildung 5.2 visualisiert wird. Eine relativ kleine Anzahl von Paketen mit sehr großen Abhängigkeitsbäumen verschiebt den Durchschnitt nach oben. Die Tiefe der Abhängigkeitsbäume beträgt im Median 3 Ebenen, im Durchschnitt allerdings 3,5 Ebenen.

Die Analyse der direkten Abhängigkeiten, also derjenigen Pakete, die explizit in der `package.json` eines Projekts deklariert werden, offenbart charakteristische strukturelle Merkmale des NPM-Ökosystems. Die untersuchten Pakete weisen im Durchschnitt fünf direkte Abhängigkeiten auf, während, wie bereits erwähnt,

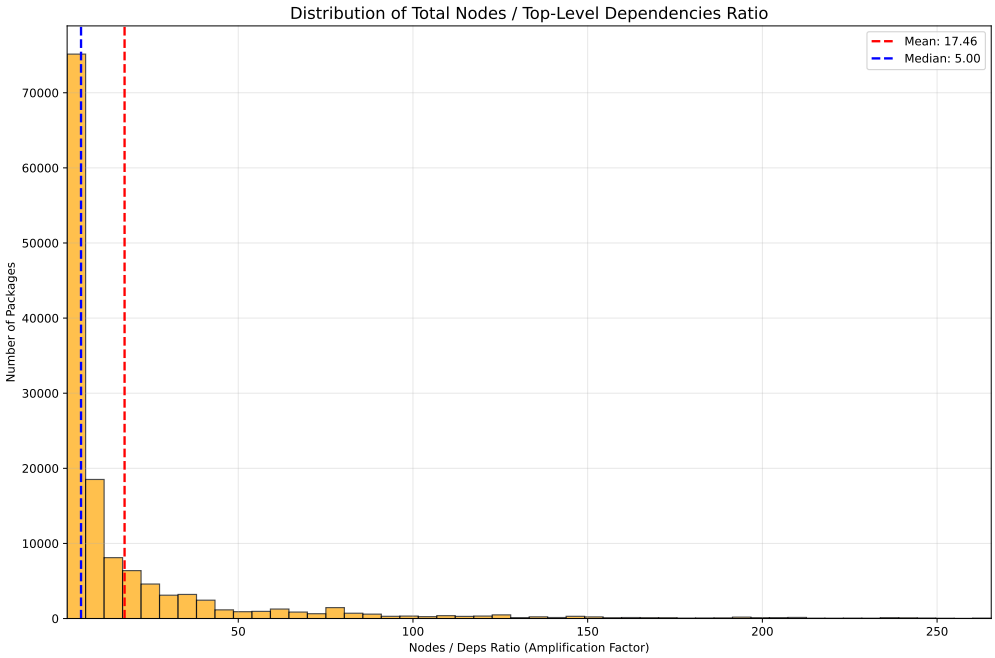

Abbildung 5.2.: Verteilung der Anzahl der Abhängigkeiten der 178.775 untersuchten Pakete

die Gesamtzahl der Abhängigkeiten inklusive der transitiven Abhängigkeiten bei durchschnittlich 97 liegt. **Daraus ergibt sich ein Verhältnis von 1:19, das die ausgeprägte Dominanz transitiver Abhängigkeiten im NPM-Ökosystem verdeutlicht**. Im Vergleich zu den von Zimmermann et al. im Jahr 2018 durchgeführten Untersuchungen hat sich dieses Ungleichgewicht weiter verstärkt [55]. Die Dominanz transitiver Abhängigkeiten führt dazu, dass trotz der vergleichsweise geringen Anzahl direkt deklarierter Abhängigkeiten eine deutlich größere und vor allem undurchsichtigere Angriffsoberfläche entsteht.

In den analysierten Paketen wurden insgesamt 618 unterschiedliche Schwachstellen identifiziert. Das entspricht etwa 15 % aller bekannten Schwachstellen im NPM-Ökosystem. Die Verteilung der 497 mit einem CVSS-Wert bewerteten Schwachstellen nach Schweregraden zeigt dabei eine Konzentration im mittleren bis hohen Bereich: 23 Schwachstellen wurden als Low eingestuft, 205 als Medium, 195 als High und 74 als Critical klassifiziert. Insgesamt besitzen 17.804 der analysierten Pakete mindestens eine Schwachstelle. Dies entspricht etwas mehr als 10 % der insgesamt untersuchten Pakete. Einige Pakete sind von mehreren Schwachstellen betroffen. Gezählt wurden 39.833 unterschiedliche Instanzen von Schwachstellen.

Andere Arbeiten ermitteln hingegen für das gesamte NPM-Ökosystem einen Anteil von 40 % vulnerabler Pakete [55]. Dieser deutliche Unterschied lässt sich durch das in Abschnitt 4.1 beschriebene Auswahlverfahren erklären, bei dem

gezielt die populärsten Pakete analysiert wurden. Wie in den vorherigen Abschnitten gezeigt wurde, weisen populäre Pakete tendenziell weniger und kleinere Abhängigkeitsbäume auf, was die Wahrscheinlichkeit reduziert, dass sie von Schwachstellen in ihren Abhängigkeitsbäumen betroffen sind. Es ist anzunehmen, dass bei Einbeziehung weniger populärer Pakete der Anteil vulnerabler Pakete sich dem für das Gesamtökosystem ermittelten Wert von 40 % annähern würde [55].

Von den 39.833 Schwachstelleninstanzen ließen sich 30.574 mittels Erreichbarkeitsanalyse analysieren, da für sie in den Metadaten der Schwachstelle eine Referenz direkt zu einem Security-Patch oder aber, wie in Abschnitt 4.3 beschrieben, etwas Vergleichbarem, vorhanden war. Dies entspricht einer Quote von 76 % und bestätigt die in Abschnitt 4.3 beschriebene hohe Verfügbarkeit von solchen Security-Patch-Referenzen, insbesondere für neuere und kritische Schwachstellen. Die verbleibenden 9.259 Schwachstellen-Instanzen konnten nicht analysiert werden, da keine Metadaten für die Identifikation des Patches verfügbar waren, oder es bei der Analyse zu einem Fehler gekommen ist.

Von den 30.574 analysierbaren Schwachstellen wurden durch die entwickelte Methodik 20.799 Schwachstellen als False-Positives klassifiziert, während 9.775 als True-Positives identifiziert wurden. Dies entspricht einer False-Positive-Rate von etwa 68 %, was sich mit Erkenntnissen der Studie [12] deckt und die Notwendigkeit einer präzisen Erreichbarkeitsanalyse unterstreicht. Ohne die durchgeführte statische Analyse würden alle 30.574 Schwachstellen-Instanzen als relevant gemeldet, was potenziell zu einer erheblichen Überlastung der Entwickler mit irrelevanten Warnmeldungen führen würde.

Antwort auf RQ1 (Reduktionspotenzial): Die Analyse zeigt ein erhebliches Reduktionspotenzial: Von 30.574 durch metadatenbasierte Scanner (`npm audit`) gemeldeten Schwachstellen sind nur 9.775 (32 %) tatsächlich erreichbar. Die verbleibenden 20.799 Schwachstellen (68 %) stellen False-Positives dar, bei denen kein Aufrufpfad zu den vulnerablen Funktionen identifiziert wurde.

5.2. Muster in der Schwachstellenvererbung (RQ2)

Die identifizierten 9.775 True-Positives lassen sich weiter in zwei Kategorien unterteilen: 4.663 direkte Schwachstellen, bei denen das `(Paket, Version)`-Tupel selbst die Schwachstelle beinhaltet, und 5.112 vererbte Schwachstellen, bei denen die Schwachstelle durch die Nutzung vulnerabler Funktionen aus transitiven Abhängigkeiten propagiert wurde.

Insgesamt wurden 1.328.336 vulnerable Symbole für die 9.775 True-Positive-Schwachstellen identifiziert. Diese Zahl umfasst sowohl die direkt vulnerablen Funktionen in den ursprünglich betroffenen Paketen als auch alle Funktionen, die durch die Taint-Analyse und Vererbung als vulnerabel klassifiziert wurden. Die hohe Anzahl verdeutlicht, dass eine einzelne Schwachstelle in einer grundlegenden Funktion sich auf zahlreiche aufrufende Funktionen im selben Paket und in abhängigen Paketen auswirken kann. Dennoch ist anzumerken, dass die vulnerablen Symbole jeden *AccessPath* einer Funktion beinhalten. Das bedeutet, dass etwa eine Funktion, welche häufiger über verschiedene Wege exportiert wird, häufiger gezählt wird. Die Annahme, eine Schwachstelle verteilt sich auf im Durchschnitt auf 135 unterschiedliche Funktionen, ist nicht zutreffend.

Von besonderem Interesse ist die Analyse der 603.925 vulnerablen Symbole, die eine Vererbung einer Schwachstelle über Paketgrenzen hinweg beinhalten. Diese Symbole repräsentieren Funktionen, bei denen der Aufrufpfad Paketgrenzen überschreitet und somit eine Schwachstelle von einem Paket in ein anderes propagiert wird. Die Verteilung nach der Anzahl überschrittener Paketgrenzen, wie in Abbildung 5.3 dargestellt, zeigt ein deutliches Muster: 535.096 dieser Symbole überspringen eine einzelne Paketgrenze, 48.006 überspringen zwei Paketgrenzen, 16.126 überspringen drei Grenzen, 1.765 überspringen vier Grenzen und 1.287 überspringen fünf Paketgrenzen.

Diese Verteilung folgt einem exponentiellen Abklingverhalten, das sich durch die Funktion $A \cdot \text{decay}^d$ modellieren lässt (visualisiert durch die rote Kurve in Abbildung 5.3), wobei d die Anzahl der überschrittenen Paketgrenzen darstellt und der Decay-Wert bei etwa 0,094 liegt. Dieses Verhalten lässt sich dadurch erklären, dass mit jeder zusätzlichen Paketgrenze die Wahrscheinlichkeit sinkt, dass die vulnerablen Funktionen tatsächlich genutzt werden. Viele Pakete verwenden nur einen begrenzten Teil der API ihrer Abhängigkeiten, wodurch die Propagierung von Schwachstellen über mehrere Ebenen hinweg zunehmend unwahrscheinlicher wird. **Die überwiegende Mehrheit der Schwachstellenvererbungen basiert auf direkten oder einstufigen Abhängigkeitsbeziehungen, während tiefere Propagierungen exponentiell seltener auftreten.** Der exponentielle Zusammenhang zwischen Ausnutzbarkeit und Tiefe der Abhängigkeit wurde bereits in der Arbeit *On the Effect of Transitivity and Granularity on Vulnerability Propagation in the Maven Ecosystem* für das Maven-Ökosystem nachgewiesen [31]. Die vorliegenden Ergebnisse zeigen, dass ein analoger Zusammenhang auch im NPM-Ökosystem besteht.

Während der Analyse wurde zudem deutlich, dass sich das Verhältnis zwischen

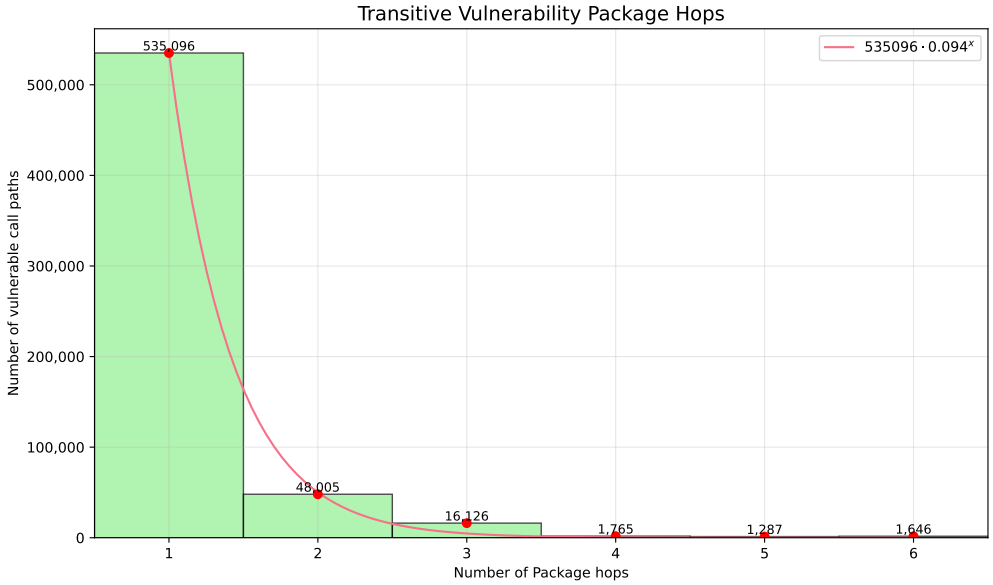

Abbildung 5.3.: Vererbung über Paketgrenzen hinweg transitiver Schwachstellen

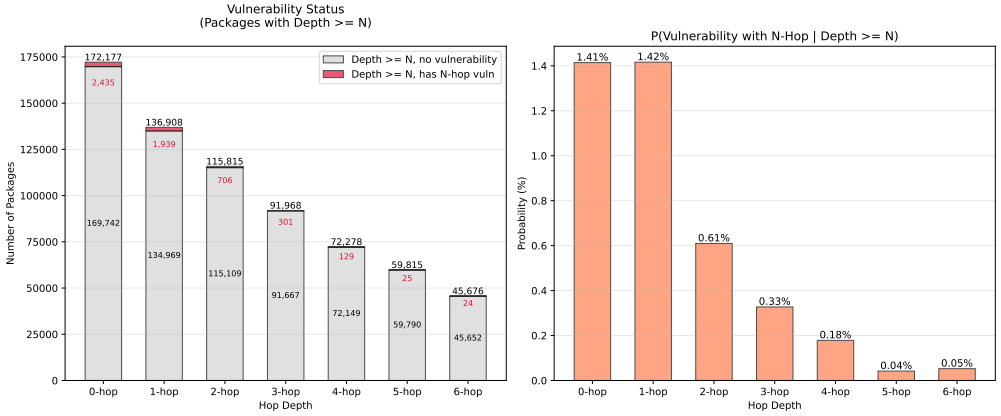

Abbildung 5.4.: Wahrscheinlichkeit von tiefen Schwachstellen, gegeben, diese sind strukturell innerhalb des Abhängigkeitsgraphen möglich. Grafik veranschaulicht exponentiell seltener auftretende Schwachstellen in Relation zur Tiefe.

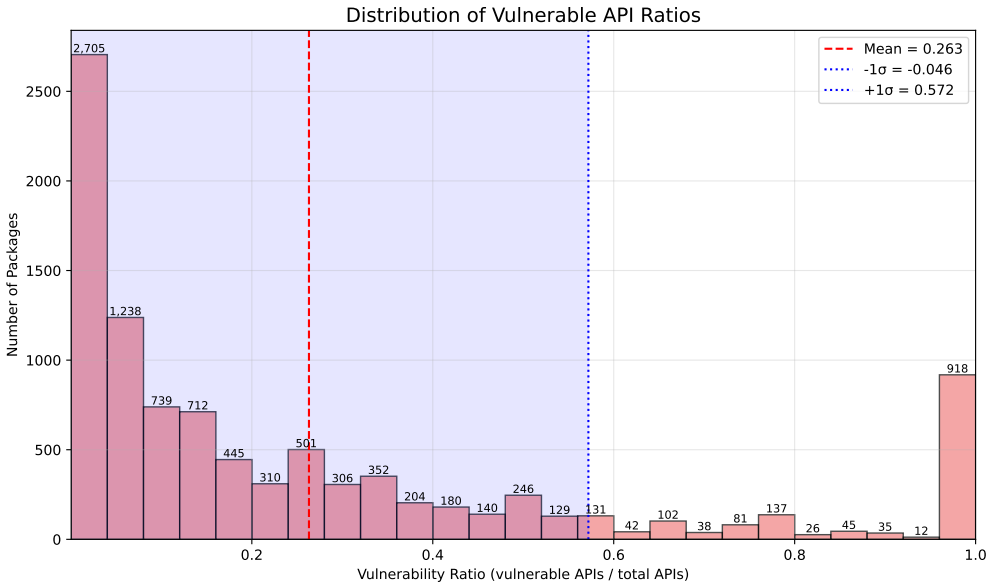

Abbildung 5.5.: Verhältnis zwischen den als vulnerabel identifizierten exportierten Funktionen einer Bibliothek zu allen exportierten Funktionen

exportierten und vulnerablen Funktionen einer Bibliothek bei zunehmend weniger populären Paketen sogar weiter in Richtung Null verschiebt.

Abbildung 5.6 visualisiert diesen Zusammenhang und zeigt eine schwach positive Korrelation zwischen der Popularität eines Pakets (gemessen an den Downloadzahlen) und dem Verhältnis vulnerabler zu exportierten APIs. Je bekannter ein Paket ist, desto höher ist tendenziell der Anteil vulnerabler Funktionen an der Gesamtzahl der exportierten APIs. Dies lässt sich durch die unterschiedliche Größe und Struktur der API-Oberflächen erklären: Populäre Pakete weisen typischerweise eine kompakte, fokussierte API auf, was dem Prinzip der Unix-Philosophie entspricht, wonach erfolgreiche Software-Komponenten eine Sache gut machen sollten [30]. Populäre Utility-Bibliotheken wie `semver` oder `ms` implementieren oft nur wenige, hochspezialisierte Funktionen. Im Gegensatz dazu bieten weniger populäre Pakete tendenziell umfangreichere und diversifiziertere Funktionalität, wodurch der Anteil vulnerabler Funktionen an der Gesamtzahl der exportierten APIs sinkt. Die kompakte API-Oberfläche beliebter Pakete hat zur Folge, dass eine einzelne Schwachstelle einen erheblichen Teil der verfügbaren Funktionalität betreffen kann.

Neben der Betrachtung einzelner vulnerabler Symbole auf Funktionsebene wurde auch die grobgranulare Struktur vulnerabler und als nicht vulnerabel identifizierter Pfade analysiert, also die Aufrufbeziehungen zwischen Paketen. Im Gegensatz zur feingranularen Aufrufstruktur auf Funktionsebene werden hier mehrere Funktionsaufrufe zwischen zwei Paketen nur einmal gezählt, sodass die Analyse die grundlegende Propagierungsstruktur auf Paketebene erfasst. Auch hier ist

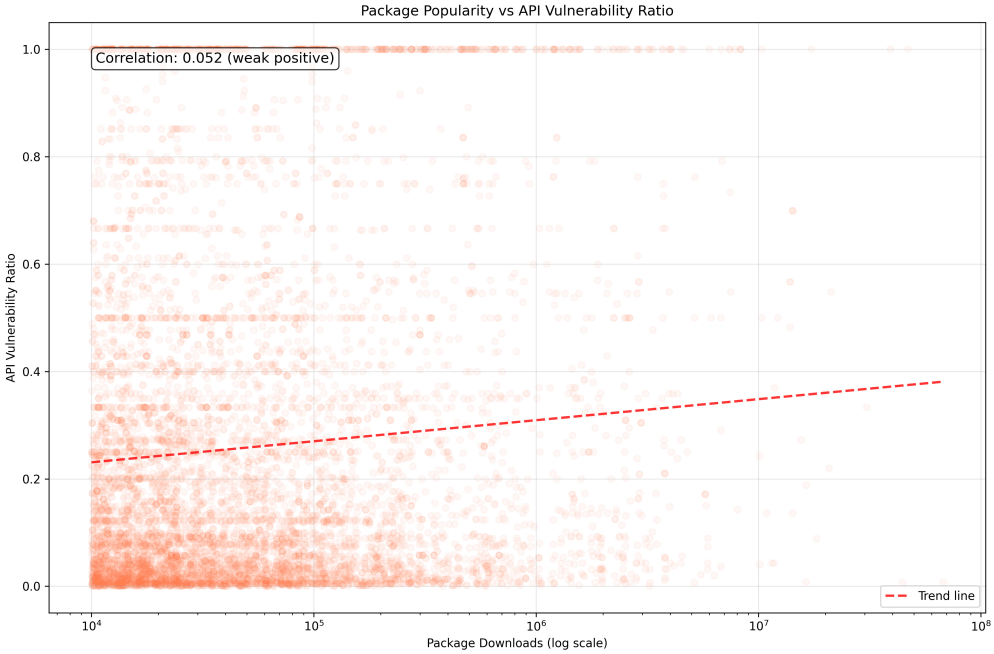

Abbildung 5.6.: Verteilung des Verhältnisses zwischen vulnerablen exportierten Funktionen und allen exportierten Funktionen

eine exponentielle Abnahme der Häufigkeit vulnerabler Pfade mit zunehmender Pfadlänge erkennbar, was die zuvor beschriebenen Beobachtungen auf einer höheren Abstraktionsebene bestätigt und dem erwarteten Verhalten entspricht. Dies wird in Abbildung 5.7 visualisiert.

Die Beobachtung des exponentiellen Abklingverhaltens bei der Schwachstellenpropagierung wird durch die Betrachtung des Verhältnisses von vulnerablen, exportierten APIs zu allen exportierten APIs einer Bibliothek weiter gestützt. Abbildung 5.5 zeigt, dass im Durchschnitt 26,3 % der exportierten APIs einer von einer Schwachstelle betroffenen Bibliothek tatsächlich vulnerabel sind. Dies bedeutet im Umkehrschluss, dass durchschnittlich 73,7 % der exportierten Funktionalität weiterverwendet werden können, ohne dass die Schwachstelle propagiert wird.

Allerdings ist diese durchschnittliche Betrachtung mit Vorsicht zu interpretieren, da die Daten eine hohe Standardabweichung aufweisen. Das Verhältnis vulnerabler zu sicheren APIs variiert stark zwischen verschiedenen Schwachstellen: Einige Schwachstellen betreffen nur wenige spezifische Funktionen einer großen API-Oberfläche, während andere einen Großteil der exportierten Funktionalität kompromittieren. Diese Variabilität ist sowohl auf die unterschiedliche Natur der Schwachstellen als auch auf die genutzte Auswahlstrategie der analysierten Pakete zurückzuführen und erschwert pauschale Aussagen über das NPM-Ökosystem als Ganzes.

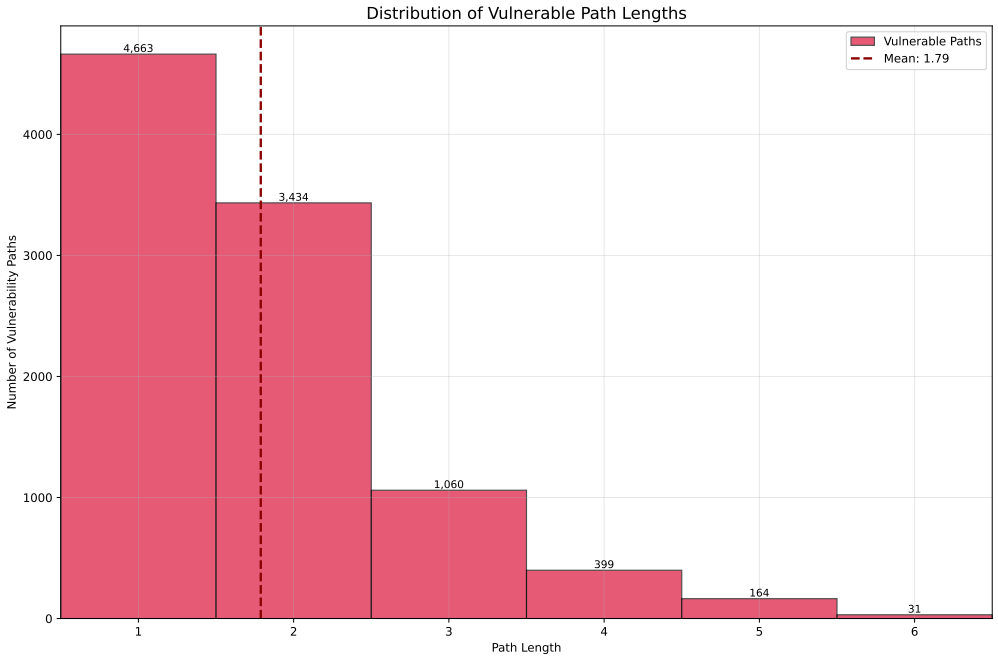

Abbildung 5.7.: Verteilung der Länge vulnerabler Pfade. Pfadlänge 1 bedeutet keine Propagierung zwischen Paketen.

Dennoch lässt sich aus dem durchschnittlichen Verhältnis ein theoretisches Modell für die schwachstellenübergreifende Propagierungswahrscheinlichkeit ableiten. Die Wahrscheinlichkeit einer Vererbung über mehrere Paketgrenzen hinweg kann als bedingte Wahrscheinlichkeit modelliert werden. Gegeben der Abhängigkeitsstruktur $A \to B \to C_{\text{vuln}}$ lässt sich die theoretische Wahrscheinlichkeit, dass Paket A die Schwachstelle von C über B erbt, wie folgt annähern:

$$P(A_{\text{vuln}} \mid C_{\text{vuln}}) = P(A_{\text{vuln}} \mid B_{\text{vuln}}) \cdot P(B_{\text{vuln}} \mid C_{\text{vuln}})$$

Unter Verwendung der durchschnittlichen 26,3 % für beide bedingten Wahrscheinlichkeiten ergibt sich:

$$\begin{aligned} P(A_{\text{vuln}} \mid C_{\text{vuln}}) &= P(A_{\text{vuln}} \mid B_{\text{vuln}}) \cdot P(B_{\text{vuln}} \mid C_{\text{vuln}}) \\ &= 0.263 \cdot 0.263 \\ &= 0.069 \\ &= 6.9\% \end{aligned}$$

Diese theoretische Formel suggeriert ebenfalls ein exponentielles Abklingverhalten der Form $P(\text{n-Paketgrenzen-Vererbung}) = p^n$. Die empirischen Daten bestätigen dieses Muster in den absoluten Zahlen: Von 2.360 direkt betroffenen Paketen sind nach einer Vererbungsstufe noch 1.487 (63%), nach zwei Stufen 512

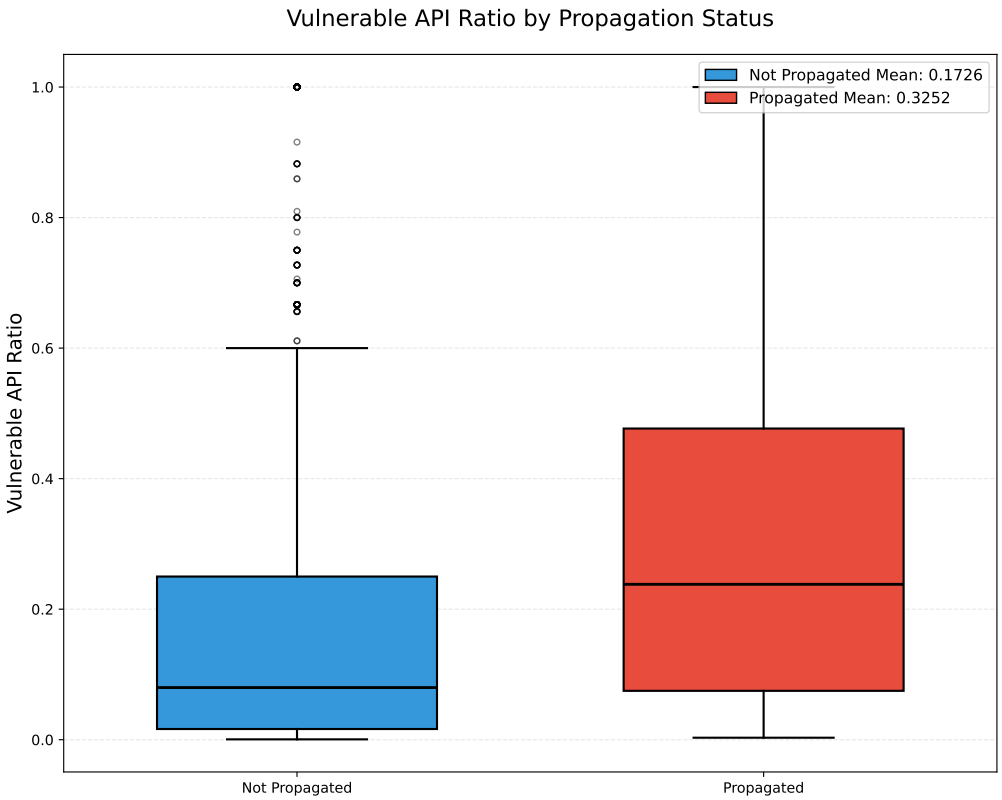

Abbildung 5.8.: Einfluss des Anteils vulnerabler exportierter APIs darauf, ob eine Schwachstelle vererbt wird oder nicht. Die unterschiedlichen Box-Plots zeigen einen deutlichen Unterschied des Durchschnittswerts.

(22%), nach drei Stufen 134 (6%), nach vier Stufen 72 (3%) und nach fünf Stufen 20 (0,8%) betroffen. Abbildung 5.4 visualisiert diesen exponentiellen Abfall.

Eine Analyse der Vulnerable-API-Ratios zeigt, dass die Wahrscheinlichkeit einer Propagierung stark von dem prozentualen Anteil verwundbarer APIs abhängt. Nicht propagierte Schwachstellen weisen einen Mittelwert von 18,4 % und einen Median von 7,5 % vulnerabler APIs auf, während propagierte Schwachstellen deutlich höhere Werte zeigen (Mittelwert: 34,5 %, Median: 24,3 %). Ein Mann-Whitney-U-Test bestätigt, dass dieser Unterschied hochsignifikant ist ($U = 1.480.944$, $p < 0{,}001$).

Abbildung 5.8 illustriert den signifikanten Einfluss des Vulnerable-API-Ratios auf die Wahrscheinlichkeit der Schwachstellenvererbung und unterstützt damit die theoretische Annahme eines exponentiellen Abfalls über mehrere Vererbungsebenen.

Die Korrelationsanalyse ergab folgende Zusammenhänge zwischen den strukturellen Merkmalen von Bibliotheken und der Propagation von Schwachstellen:

- Die Korrelation zwischen dem Anteil vulnerabler APIs an der Gesamt-

zahl exportierter APIs (Vulnerable-API-Ratio) und der Propagation von Schwachstellen beträgt $r = 0.2597$ ($p = 1.04 \times 10^{-72}$). Der positive Korrelationskoeffizient zeigt an, dass ein höherer Anteil vulnerabler APIs mit einer häufigeren Weitergabe von Schwachstellen an abhängige Bibliotheken einhergeht.

- Die Korrelation zwischen der absoluten Anzahl exportierter APIs einer Bibliothek und der Propagation von Schwachstellen beträgt $r = -0.1801$ ($p = 2.86 \times 10^{-35}$). Der negative Korrelationskoeffizient bedeutet in diesem Kontext, dass eine geringere Anzahl exportierter APIs tendenziell mit einer häufigeren Weitergabe von Schwachstellen einhergeht, während eine höhere Anzahl exportierter APIs tendenziell mit einer selteneren Weitergabe von Schwachstellen verbunden ist.

Die Pfadanalyse bildet einen zentralen Bestandteil der in dieser Arbeit entwickelten Methodik zur Bewertung der Erreichbarkeit von Schwachstellen in Abhängigkeitsgraphen. Wie in Abschnitt 2.4.1 diskutiert, unterliegt die statische Code-Analyse grundsätzlichen Einschränkungen, insbesondere beim Umgang mit dynamischen Sprachkonstrukten, die die Vollständigkeit der Ergebnisse beeinflussen können. Dennoch lässt sich eine klare Aussage über die Belastbarkeit der identifizierten Pfade treffen: Sobald ein Aufrufpfad durch die statische Analyse explizit identifiziert wurde, beträgt die Konfidenz in dessen **statische** Existenz 1 (100 %). Dies bedeutet, dass ein im Callgraphen gefundener Pfad mit absoluter Sicherheit als statisch tatsächlich vorhanden betrachtet werden kann. Dies trifft allerdings keine Aussage darüber, ob der Code auch dynamisch erreicht werden kann.

Um die Robustheit der Analyse gegenüber dynamischen Konstrukten für **nicht vulnerable** Pfade zu bewerten, wurden die in Abschnitt 4.9 beschriebenen Semgrep-Regeln auf alle Bibliotheken innerhalb eines nicht vulnerablen Pfades angewendet. Dabei wird die gesamte Bibliothek untersucht, sodass eine Regel ausgelöst wird, sobald ein dynamisches Konstrukt an beliebiger Stelle im Code vorkommt, unabhängig davon, ob es in direktem Zusammenhang mit der analysierten Schwachstelle oder den identifizierten Pfaden steht. Diese konservative Vorgehensweise stellt sicher, dass keine potenziellen Risiken übersehen werden, führt jedoch zu einer generellen Reduktion der Konfidenz für alle Pfade, die eine Bibliothek mit dynamischen Konstrukten enthalten, auch wenn diese nicht direkt mit der Schwachstelle zusammenhängen.

Die Konfidenz eines Pfades wird durch die folgende Formel berechnet:

$$\text{Konfidenz} = 1 - \frac{\text{Bibliotheken im Pfad mit mind. einer ausgelösten Regel}}{\text{Bibliotheken im Pfad}}$$

Diese Metrik quantifiziert den Anteil der Bibliotheken entlang eines nicht vulnerablen Pfades, die keine dynamischen Konstrukte aufweisen, welche die statische Analyse beeinträchtigen könnten. Die Regeln werden auch für das erste

Element, also die von einer Schwachstelle betroffene Bibliothek selbst, im Pfad berücksichtigt. Der Mittelwert aller nicht vulnerabler Pfade liegt bei 0,756, was einer durchschnittlichen Konfidenz von 75,6 % entspricht. Dies bedeutet, dass bei einem typischen Pfad etwa drei Viertel der beteiligten Bibliotheken keine dynamischen Konstrukte enthalten, die die Analyseergebnisse verfälschen könnten.

Ein besonders aufschlussreicher Aspekt der Analyse betrifft die Fälle, in denen ein vulnerabler Pfad unterbrochen wird, also wenn ein Paket B keine vulnerablen Funktionen eines Pakets A aufruft und somit die Schwachstellenpropagierung stoppt. Abbildung 5.9 zeigt die Verteilung ausgelöster Semgrep-Regeln für solche unterbrechenden Pakete. Auch in diesen Fällen liegt der Prozentsatz mit 76,9 % nah an der durchschnittlichen Konfidenz, dass tatsächlich keine vulnerable Funktion aufgerufen wurde. Bei den verbleibenden 23,1 % der Pakete werden im Durchschnitt 1,45 Regeln ausgelöst, was auf das Vorhandensein dynamischer Konstrukte hindeutet und eine manuelle Überprüfung nahelegt, um False-Negatives auszuschließen.

Diese Ergebnisse unterstreichen die Zuverlässigkeit der statischen Pfadanalyse für die Identifikation vulnerabler Aufrufpfade, während gleichzeitig die Notwendigkeit einer konservativen Bewertung dynamischer Konstrukte deutlich wird. Die Konfidenzmetrik bietet dabei eine quantitative Grundlage, um die Belastbarkeit der Analyseergebnisse einzuschätzen und priorisierte Handlungsempfehlungen für die Sicherheitspraxis abzuleiten. Für kritische Anwendungsfälle bleibt eine manuelle Überprüfung insbesondere bei Pfaden mit niedriger Konfidenz unverzichtbar, um die Genauigkeit der Risikobewertung weiter zu erhöhen.

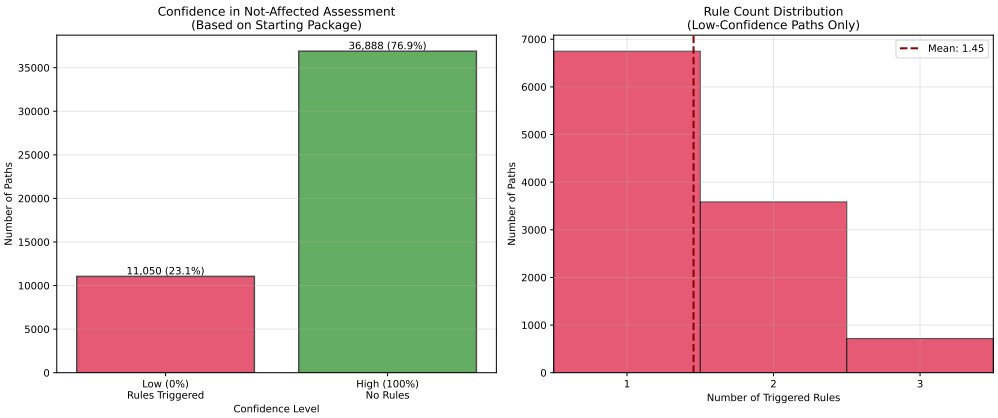

Abbildung 5.9.: Verteilung ausgelöster Semgrep-Regeln in unterbrechenden Paketen. 76,9 % der Pakete lösen keine Regeln aus, was auf eine hohe Konfidenz der Pfadunterbrechung hindeutet.

Der wesentliche Mehrwert dieser Konfidenzanalyse liegt in der Transparenz der methodischen Limitierungen. Die Konfidenz der Analyse wird explizit quantifiziert, was eine fundierte Bewertung der Ergebnisse ermöglicht. Für Fälle, in denen eine manuelle Überprüfung erforderlich ist, bietet die Methodik präzise Ansatz-

punkte. Anstatt die gesamte Bibliothek oder alle Abhängigkeiten untersuchen zu müssen, kann die Überprüfung gezielt auf die identifizierten vulnerablen Funktionen fokussiert werden. Diese Überprüfung lässt sich zudem auf die Stellen mit dynamischem Code konzentrieren, indem zunächst die dynamischen Konstrukte mittels der Semgrep-Regeln lokalisiert werden und anschließend untersucht wird, ob diese die vulnerablen Funktionen aufrufen könnten. Dieser gezielte Ansatz reduziert den manuellen Aufwand und ermöglicht eine Validierung der automatisch generierten Ergebnisse.

5.2.1. Beispiel Schwachstelle CVE-2017-16137

Ein illustratives Beispiel für die Komplexität der Schwachstellenpropagierung bietet die Schwachstelle `GHSA-gxpj-cx7g-858c` (`CVE-2017-16137`), die den höchsten Vererbungsfaktor aller analysierten Schwachstellen aufweist. Für diese Schwachstelle wurde ein Aufrufpfad zu 185 anderen Paketen identifiziert, während sie gleichzeitig in 71 Paketen als False-Positive klassifiziert wurde, weil die Schwachstelle zwar als Abhängigkeit vorhanden ist, aber die vulnerablen Funktionen nicht aufgerufen werden.

Betroffen von dieser Schwachstelle ist das NPM-Paket `debug`, von dem 31 unterschiedliche Versionen untersucht wurden. Abbildung 5.10 visualisiert die Schwachstellenvererbung exemplarisch für die Version `2.2.0` in Form eines Netzwerkgraphen. Die Analyse dieser spezifischen Version zeigt ein differenziertes Bild der Propagierung: Die Schwachstelle vererbt sich über 45 Kanten an Downstream-Pakete, während über 16 weitere Kanten keine Vererbung stattfindet, da die betroffenen Pakete die vulnerablen Funktionen nicht nutzen. Insgesamt sind 46 unterschiedliche Bibliotheken, häufig in verschiedenen Versionen, von dieser Schwachstelle betroffen, während bei 10 Bibliotheken False-Positives auftreten.

Diese Schwachstelle weicht damit erheblich vom durchschnittlichen Verhalten ab. Während über alle analysierten Schwachstellen hinweg eine False-Positive-Rate von etwa 68 % beobachtet wurde, liegt diese für `GHSA-gxpj-cx7g-858c` bei nur etwa 18 % (10 von 56 betroffenen Bibliotheken). Dies lässt sich durch die Natur der Schwachstelle erklären: `debug` ist eine grundlegende Logging-Bibliothek, deren Funktionalität von vielen Paketen aktiv genutzt wird. Die Bibliothek exportiert 198 verschiedene Access-Path-Patterns. Von diesen sind 151 von der Schwachstelle betroffen. Dies entspricht einem prozentualen Anteil von 76 %. Dieses Beispiel unterstreicht, dass die Propagierungswahrscheinlichkeit stark von der Art der betroffenen Funktionalität, sowie der konkreten Bibliothek, in der die Schwachstelle vorkommt, abhängt.

Der folgende Vererbungspfad veranschaulicht die Propagierung über drei Paketgrenzen hinweg. Die Schwachstelle befindet sich ursprünglich im Paket `debug@2.2.0`, konkret in der Hauptexportfunktion (default export) `debug()` in der Datei `src/debug.js` (Zeilen 66-116). Diese Zeilen wurden aufgrund des Security-Patches `https://github.com/debug-js/debug/commit/71169065b5262f9858ac78cc0b688c84a438f290` als vulnerabel markiert. Von dort propagiert

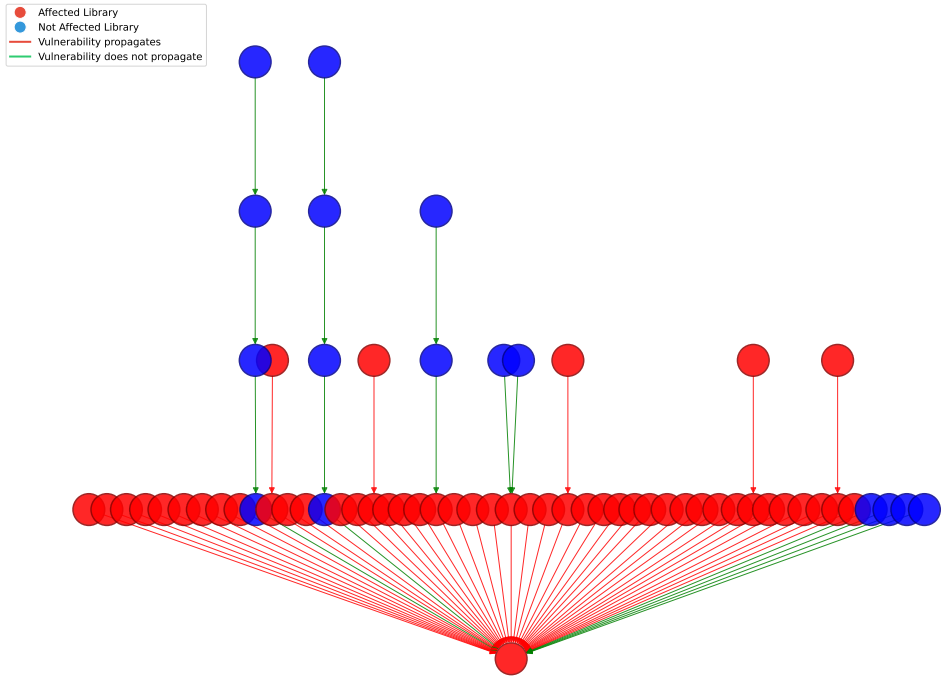

Abbildung 5.10.: Exemplarisches Vererbungsnetzwerk für Schwachstelle GHSA-gx-pj-cx7g-858c der Bibliothek `debug` in Version `2.2.0`. Der Knoten am unteren Rand der Grafik repräsentiert die Bibliothek `debug`. Jede Kante steht für die Nutzung der Bibliothek als Abhängigkeit in einem weiteren Projekt

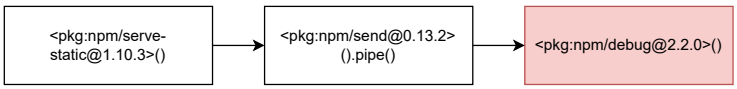

Abbildung 5.11.: Ausschnitt der Vererbung für Schwachstelle GHSA-gxpj-cx7g-858c mit Information über die konkrete Funktionsnutzung

sie zum Paket `send@0.13.2`, wo die Methode `pipe` der `send`-Instanz (definiert in `index.js`, Zeilen 431-505) den vulnerablen default Export aufruft. Schließlich vererbt sich die Schwachstelle weiter zu `serve-static@1.10.3`, dessen Hauptexportfunktion (Zeilen 71-125 in `index.js`) über die interne Funktion `serveStatic` letztlich die vulnerable Funktionalität `send.pipe` nutzt. Dieser Pfad der Schwachstellenvererbung ist in Abbildung 5.11 grafisch dargestellt.

Ein instruktives Gegenbeispiel für einen False-Positive bildet die Bibliothek `socket.io-adapter@0.4.0`. Diese Bibliothek deklariert in ihrer `package.json` eine explizite Abhängigkeit zu `debug`, nutzt die importierte Funktionalität jedoch nicht im Code. Dieser Befund wurde durch eine manuelle Inspektion des Quellcodes der Bibliothek `socket.io-adapter@0.4.0` verifiziert. Dies bedeutet, dass die Bibliothek `debug` mit der enthaltenen Schwachstelle zwar bei der Installation von `socket.io-adapter` heruntergeladen und im `node_modules`-Verzeichnis abgelegt wird, jedoch keinerlei tatsächliche Verwendung findet.

Dennoch weist `npm audit` bei der Installation von `socket.io-adapter` auf das Vorhandensein der Schwachstelle hin, da der metadatenbasierte Ansatz ausschließlich die Präsenz der vulnerablen Abhängigkeit im Abhängigkeitsbaum prüft. Die entwickelte Erreichbarkeitsanalyse klassifiziert diesen Fall korrekt als False-Positive. Diese Nichtbetroffenheit gilt sowohl für `socket.io-adapter` selbst als auch für alle Bibliotheken, die von diesem Paket abhängig sind, sofern keine alternativen Pfade in ihrem Abhängigkeitsgraphen zu `debug` existieren. Existiert etwa eine andere Abhängigkeitskette, über die `debug` tatsächlich genutzt wird, müsste diese separat analysiert werden.

Antwort auf RQ2 (Propagierungsmuster): Schwachstellen propagieren sich mit exponentiell abnehmender Wahrscheinlichkeit über Paketgrenzen hinweg. Die Propagierungswahrscheinlichkeit variiert jedoch stark zwischen Schwachstellen: Während durchschnittlich 26,5 % der exportierten APIs vulnerabel sind, zeigen kleinere Bibliotheken deutlich höhere, größere Bibliotheken deutlich niedrigere, Vererbungsraten. Somit eignet sich die Größe einer Bibliothek als aussagekräftige Kenngröße für die Vererbungswahrscheinlichkeit von Schwachstellen.

5.3. Integration der gewonnenen Erkenntnisse

Um die gesammelten Informationen praktisch nutzbar zu machen und die in Teilfrage 3 aufgeworfene Frage nach der Integration in bestehende Security-Tools zu adressieren, wurde ein HTTP-Server implementiert, der die VEX-Daten über eine standardisierte Schnittstelle bereitstellt. Der Server ist in Node.js implementiert und ermöglicht den Abruf von VEX-Informationen für die analysierten Package-URLs im JSON-Format.

Die Schnittstelle des Servers folgt einer RESTful-Architektur und bietet einen GET-Endpunkt unter `/vex`, der die PURL als Query-Parameter entgegennimmt. Ein exemplarischer Aufruf hat die Form `http://localhost:3000/vex?purl=pkg:npm/chokidar@3.6.0`. Dieser Endpunkt liefert alle verfügbaren VEX-Statements für das spezifizierte Paket in der angegebenen Version zurück, einschließlich Informationen über betroffene und nicht betroffene Schwachstellen sowie der identifizierten vulnerablen Symbole.

Listing A.2 zeigt exemplarisch eine solche Antwort für das Paket `chokidar` in Version 3.6.0, welches von der Schwachstelle GHSA-grv7-fg5c-xmjg in seiner direkten Abhängigkeit `braces` betroffen ist. Die Antwort enthält die Klassifikation der Schwachstelle als `exploitable`, was bedeutet, dass die vulnerablen Funktionen tatsächlich über die API von `chokidar` erreichbar sind. Das `evidence`-Array dokumentiert den vollständigen Vererbungspfad analog zu der in Kapitel 5.2.1 beschriebenen Schwachstelle.

Der Server unterstützt sowohl das OpenVEX-Format als auch das CycloneDX-VEX-Format, was prinzipiell die Integration in bestehende Software Composition Analysis (SCA) Tools und Security-Pipelines ermöglicht. Allerdings ist die praktische Adoption von VEX in verbreiteten Security-Tools derzeit noch begrenzt. `npm audit`, der hauseigene Scanner des NPM-Ökosystems, unterstützt VEX-Dateien zum gegenwärtigen Zeitpunkt noch nicht. Der OSV-Scanner hat angekündigt, VEX-Unterstützung in Zukunft zu implementieren. Unter den etablierten Tools bietet derzeit primär Trivy, ein Security-Scanner, native Unterstützung für VEX-Files.

Diese noch unvollständige Tool-Unterstützung stellt eine Herausforderung für die praktische Nutzbarkeit der generierten VEX-Daten dar. Um dennoch einen unmittelbaren Mehrwert zu schaffen, können die bereitgestellten Daten über die HTTP-API in bestehende CI/CD-Pipelines integriert werden. Entwicklungsteams können eigene Wrapper-Skripte implementieren, die nach einem `npm audit`-Scan die VEX-API konsultieren, um False-Positives automatisiert herauszufiltern. Mit zunehmender Adoption des VEX-Standards durch verbreitete Security-Tools wird die Integration ohne zusätzliche Implementierungsarbeit voraussichtlich möglich.

5.3.1. Beispielhafte Integration in Trivy

Um die praktische Anwendbarkeit der generierten VEX-Daten zu demonstrieren, wurde eine exemplarische Integration mit Trivy durchgeführt, einem weitverbreiteten Security-Scanner, der native VEX-Unterstützung bietet. Als Beispielpaket dient `react-devtools@4.28.5`. `react-devtools@4.28.5` weist eine transitive Schwachstelle in der Bibliothek `got@6.7.1` auf. Diese Schwachstelle ist über den folgenden Abhängigkeitsbaum erreichbar: `react-devtools@4.28.5` → `update-notifier@2.1.0` → `latest-version@3.0.0` → `package-json@4.0.0` → `got@6.7.1`. Ohne die Verwendung eines VEX-Dokuments meldet Trivy diese Schwachstelle standardmäßig.

Der Integrationsprozess beginnt mit der Generierung eines Software Bill of Materials (SBOM) mittels Trivy: `trivy fs -format cyclonedx . > sbom.json`. Anschließend wird das entsprechende VEX-Dokument vom entwickelten VEX-Server im CycloneDX-Format heruntergeladen. Listing 5.1 zeigt einen Ausschnitt des VEX-Dokuments für die Schwachstelle CVE-2022-33987 (GHSA-pfrx-2q88-qq97).

Listing 5.1: VEX-Dokument für Trivy-Integration nach hinzufügen des BOM-Links.

```
"vulnerabilities": [{
    "id": "CVE-2022-33987",
    "description": "Security vulnerability GHSA-pfrx-2
       q88-qq97",
    "affects": [{
        "ref": "urn:cdx:b576e6e0-5914-4332-81c5-
           ef4ccbb6404a/1#pkg:npm/got@6.7.1"
    }],
    "analysis": {
        "state": "not_affected",
        "detail": "Vulnerable code exists but is not
           reachable through execution paths."
    },
    "evidence": [...]
}]
```

Die Integration mit Trivy im CycloneDX-Format erfordert erhebliche manuelle Anpassungen des VEX-Dokuments, da Trivy CycloneDX BOM-Links verwendet, um Schwachstellen mit spezifischen Komponenten in der SBOM zu verknüpfen. Diese BOM-Links müssen im `affects.ref`-Feld des VEX-Dokuments platziert werden und folgen folgender Syntax: `urn:cdx:serialNumber/version#bom-ref`.

Laut der offiziellen CycloneDX-Dokumentation [4] ermöglichen BOM-Links zwar eine modulare und granular verknüpfte Darstellung von Softwarekomponenten, Abhängigkeiten und Schwachstellen, stellen jedoch hohe Anforderungen an die Konsistenz zwischen SBOM- und VEX-Dokument. Die drei essenziellen Ele-

mente des BOM-Links (`serialNumber`, `version` und `bom-ref`) müssen direkt aus der generierten SBOM extrahiert werden. Die enge Kopplung zwischen SBOM und VEX-Dokument bringt mehrere Herausforderungen mit sich. So kann das VEX-Dokument nicht unabhängig von der konkreten SBOM genutzt werden, da die BOM-Links direkt auf die spezifische Struktur und Version der SBOM verweisen. Dies bedeutet, dass jede Änderung in der SBOM, etwa durch Updates oder unterschiedliche Build-Umgebungen, eine Neugenerierung und Anpassung der BOM-Links im VEX-Dokument erfordert, was die Wartbarkeit deutlich erschwert.

Im Kontrast dazu gestaltet sich die Integration über das OpenVEX-Format deutlich einfacher. Hier war lediglich eine Anpassung der Schwachstellen-ID notwendig. Während der entwickelte VEX-Server, analog zu `npm audit`, primär GitHub Security Advisories (GHSA) verwendet, identifiziert Trivy Schwachstellen über CVE-IDs. Nach dieser Konvertierung funktioniert die Integration ohne weitere Anpassungen, da OpenVEX keine expliziten Verknüpfungen zwischen BOMs erfordert und sich daher besser für automatisierte Workflows eignet.

Listing 5.2: VEX-Dokument im OpenVEX-Format

```
"statements": [{
  "@id": "GHSA-pfrx-2q88-qq97-react-devtools@4.28.5",
  "vulnerability": {
    "@id": "CVE-2022-33987",
    "name": "CVE-2022-33987",
  },
  "products": [
    {
      "@id": "pkg:npm/react-devtools@4.28.5",
      "identifiers": {
        "purl": "pkg:npm/react-devtools@4.28.5"
      },
      "subcomponents": [{"@id": "pkg:npm/got@6.7.1"}]
    }
  ],
  "status": "not_affected",
}]
```

Trotz dieser Herausforderungen führen sowohl die Integration über CycloneDX-BOM-Links als auch die Anpassung der CVE-ID im OpenVEX-Format dazu, dass die ursprünglich von Trivy identifizierte Schwachstelle in `got@6.7.1` korrekt als *nicht betroffen* klassifiziert wird und somit nicht länger als aktives Sicherheitsrisiko angezeigt wird. Idealerweise sollte die Konsolidierung von Scanner-Ergebnissen mit VEX-Daten in einem zentralen Schwachstellenmanagement-Tool erfolgen, das die Zuordnung zwischen verschiedenen Schwachstellen-Identifikatoren (GHSA, CVE) transparent handhabt, die entsprechenden VEX-Daten automatisch

abruft und die Ergebnisse verschiedener Scanner aggregiert. Ein solches Tool würde die VEX-Integration von der spezifischen Scanner-Implementierung entkoppeln und eine konsistente Schwachstellenbewertung über verschiedene Werkzeuge hinweg ermöglichen.

Die Unterstützung für das Common Security Advisory Framework (CSAF) wurde im VEX-Server nicht implementiert und kann daher hier nicht betrachtet werden.

Antwort auf Teilfrage 3 (Bereitstellung): Die Integration der Analyseergebnisse in bestehende Security-Tools ist möglich und wurde exemplarisch mit Trivy demonstriert. Während CycloneDX-VEX eine enge Kopplung mit generierten SBOMs erfordert und die Automatisierung erschwert, ermöglicht OpenVEX eine einfachere Integration ohne SBOM-Abhängigkeiten. Die derzeit noch begrenzte VEX-Unterstützung verbreiteter Scanner wie `npm audit` stellt eine Herausforderung für die breite Adoption dar. Idealerweise sollte die Konsolidierung von Scanner-Ergebnissen mit VEX-Daten nicht in den Scannern selbst, sondern in einem zentralen Schwachstellenmanagement-Tool erfolgen, das verschiedene Scanner-Outputs aggregiert und mit VEX-Informationen anreichert.

5.4. Manuelle Verifikation der Ergebnisse und Lessons Learned

Um fundierte Aussagen über die Korrektheit und Präzision der entwickelten Methodik treffen zu können, wurde eine Stichprobe der Analyseergebnisse manuell verifiziert. Die Größe der Stichprobe wurde mittels der Margin-of-Error-Formel bestimmt, die eine statistische Aussage über die Gesamtpopulation anhand eines Konfidenzintervalls ermöglicht. Die Verifikation konzentriert sich ausschließlich auf transitive Schwachstellen, also jene Fälle, bei denen die Methodik eine Vererbungsanalyse durchgeführt hat. Direkte Schwachstellen, bei denen das analysierte Paket selbst die Schwachstelle enthält, werden nicht betrachtet, da deren Klassifikation nicht durch diese Arbeit erfolgte, sondern bereits in den Schwachstellendatenbanken dokumentiert ist. Die relevante Grundgesamtheit umfasst daher 25.911 Klassifikationen transitiver Schwachstellen: Schwachstellen-Instanzen, bei denen überprüft wurde, ob sie sich von einer Abhängigkeit auf das analysierende Paket vererben. Aus dieser Grundgesamtheit wurde eine Stichprobe gezogen, die sowohl `affected` als auch `not_affected` Klassifikationen umfasst.

Wichtig ist dabei die Präzisierung der Verifikationsebene: Es wird nicht jedes einzelne vulnerable Symbol untersucht, sondern ausschließlich die finale Klassifikation auf `(PURL, CVE_ID)`-Ebene. Diese Aggregationsebene ist die praktisch relevante, da sie der Frage entspricht, die Entwickler in der Praxis beantworten müssen: Ist mein Paket in Version X von der transitiven Schwachstelle Y

betroffen oder nicht? Die Anzahl der konkreten vulnerablen Symbole oder die spezifischen Aufrufpfade sind für diese binäre Entscheidung sekundär.

Für das Sampling wurde die in Listing 5.3 dargestellte SQL-Query verwendet, die eine zufällige Auswahl eindeutiger Kombinationen aus `(PURL, CVE_ID, Klassifikation)` erzeugt. Die Query kombiniert vulnerable Symbole und nicht betroffene Pakete mittels `UNION` und wählt anschließend 68 zufällige Einträge aus.

Die gewählte Stichprobengröße von 68 Einträgen entspricht bei einem angenommenen Anteilswert von 50 % (konservative Schätzung bei unbekannter wahrer Fehlerrate) einer Margin of Error von 10 % bei einem Konfidenzniveau von 90 %. Dies bedeutet, dass die aus der Stichprobe ermittelte Fehlerrate mit 90-prozentiger Wahrscheinlichkeit innerhalb eines Intervalls von ±10 Prozentpunkten um den wahren Wert der Gesamtpopulation liegt. Diese Parameter wurden gewählt, um eine Balance zwischen statistischer Aussagekraft und praktischem Verifikationsaufwand zu erreichen.

Listing 5.3: SQL-Query für die Stichprobenziehung

```
SELECT * FROM (
  SELECT DISTINCT purl, cve_id, "affected" as classification
  FROM vuln_symbols WHERE calls_id IS NULL
  UNION
  SELECT DISTINCT purl, cve_id, "not affected" as
      classification
  FROM not_affected
)
ORDER BY RANDOM()
LIMIT 68;
```

5.4.1. Ergebnisse der manuellen Verifikation

Die manuelle Überprüfung der 68 Stichprobenelemente wurde nach bestem Wissen und Gewissen durchgeführt. Jedes Stichprobenelement wurde durch Inspektion des Quellcodes, der Abhängigkeitsstruktur und der relevanten Schwachstellenmetadaten überprüft. Diese Verifikation offenbarte sowohl die Stärken als auch verbesserungswürdige Aspekte der entwickelten Methodik. Die Confusion Matrix in Abbildung 5.12 visualisiert die Klassifikationsergebnisse im Detail.

Zunächst zeigte sich eine bemerkenswert hohe Präzision: Von allen als `affected` klassifizierten Paketen waren 100 % tatsächlich betroffen (16 True Positives, 0 False-Positives). Dies bedeutet, dass die Methodik keine False-Positives produzierte, wenn ein Paket als vulnerabel markiert wurde, war dies stets korrekt. Ebenso wurden alle als `not_affected` klassifizierten Pakete korrekt identifiziert (33 True Negatives, 0 False Negatives in dieser Kategorie). Diese perfekte Präzision und Spezifität (jeweils 100 %) unterstreicht zwar die Zuverlässigkeit der Methodik bei positiven Aussagen, ist jedoch mit Vorsicht zu bewerten, da die

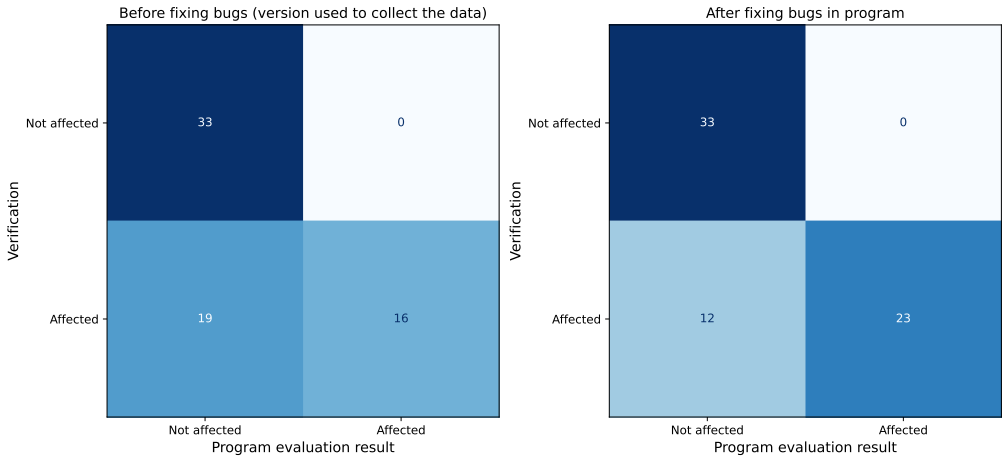

Abbildung 5.12.: Visualisierung der Ergebnisse der manuellen Verifikation. Dargestellt wird die Stichprobe sowohl auf einer verbesserten Version des Programs als auch auf dem ursprünglichen Program

konkrete Konfiguration oder Nutzung einer API durch Sichtung des Quellcodes nicht vollständig statisch erfasst werden kann.

Die initiale Herausforderung lag jedoch im Recall (Sensitivität): Mit 19 False Negatives wurden nur 45,7 % der tatsächlich betroffenen Pakete korrekt identifiziert. Dies resultierte in einer Gesamtgenauigkeit (Accuracy) von 72,1 % und einem F1-Score von 62,8 %. Die False-Negatives sind dabei auf die in den folgenden Unterabschnitten beschriebenen systematischen Limitierungen zurückzuführen, insbesondere auf Probleme beim TypeScript-zu-JavaScript-Mapping, fehlenden Support für gebündelte Abhängigkeiten und CVE-Metadaten-Inkonsistenzen.

Auf Basis der Erkenntnisse aus der manuellen Verifikation wurden gezielte Verbesserungen implementiert. Insbesondere wurde die Behandlung von TypeScript-Paketen verbessert, Support für gebündelte Abhängigkeiten hinzugefügt und die Zuordnung von Default-Exports verfeinert. Eine erneute Evaluation der Stichprobe nach diesen Anpassungen zeigt deutliche Verbesserungen: Die Anzahl der False-Negatives reduzierte sich von 19 auf 12, während True Positives von 16 auf 23 stiegen. Dies führte zu einem Recall von 65,7 % (Verbesserung um 20 Prozentpunkte), einer Gesamtgenauigkeit von 82,4 % und einem F1-Score von 79,3 %. Präzision und Spezifität blieben weiterhin bei 100 %.

Diese Ergebnisse demonstrieren, dass die Methodik bereits eine substanzielle Reduktion von Falschmeldungen ermöglicht, während die verbleibenden False-Negatives primär auf technische Herausforderungen zurückzuführen sind, die durch weitere Verfeinerungen adressierbar sind.

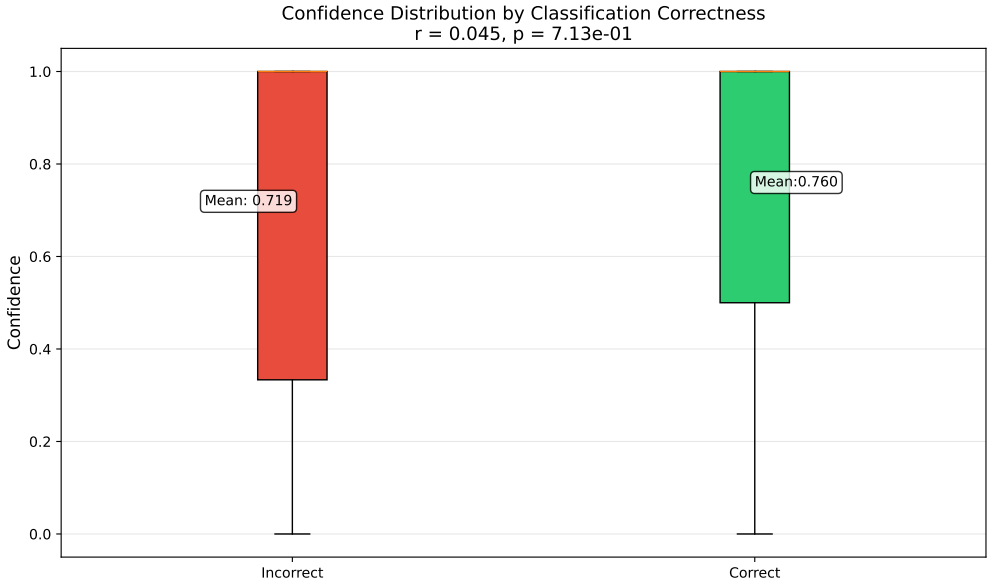

Abbildung 5.13.: Box-Plot visualisiert den Einfluss der Konfidenzmetriken auf die Klassifizierung des Programms. Die Konfidenzmetriken zeigen keine statistische Signifikanz.

5.4.2. Bewertung der Konfidenzmetrik

Die in dieser Arbeit entwickelte Konfidenzmetrik für Pfade, die auf der Präsenz dynamischer Konstrukte basiert, erweist sich als unzureichend für die Vorhersage der Klassifikationsqualität. Die empirische Evaluation zeigt, dass korrekte Klassifikationen eine durchschnittliche Konfidenz von 76 % aufweisen, während inkorrekte Klassifikationen mit 71,9 % nur geringfügig niedrigere Werte erreichen. Diese Differenz ist statistisch nicht signifikant, wie die Point-Biserial-Korrelation belegt ($r = 0{,}045$, $p = 0{,}713$). Abbildung 5.13 visualisiert diese marginale Unterscheidungskraft in einem Box-Plot, der die weitgehend überlappenden Verteilungen beider Gruppen verdeutlicht.

Die mangelnde prädiktive Kraft der Konfidenzmetrik deutet darauf hin, dass die verwendeten Semgrep-Regeln zu unspezifisch sind. Sie detektieren zwar die Präsenz dynamischer Konstrukte, können jedoch nicht differenzieren, ob diese Konstrukte tatsächlich die Analyse der konkreten Schwachstelle beeinträchtigen oder in völlig unabhängigen Codebereichen auftreten. Eine Verbesserung würde eine kontextsensitivere Regelformulierung erfordern, die beispielsweise nur dann eine Konfidenzreduktion vornimmt, wenn dynamische Konstrukte in direkter Nähe zu den identifizierten Aufrufpfaden auftreten. Konkrete Vorschläge zur Verbesserung der Regeln werden in Kapitel 5.4.3 ausgeführt.

5.4.3. Identifizierte Limitierungen und Lessons Learned

Die manuelle Verifikation offenbarte mehrere systematische Herausforderungen, die die Präzision der automatisierten Analyse beeinträchtigen und in zukünftigen Arbeiten adressiert werden sollten.

Eine wesentliche Limitation besteht in der zu groben Granularität bei der Identifikation vulnerabler Code-Bereiche sowie der fehlenden Differenzierung zwischen Modulsystemen. Wenn eine einzelne Methode einer Klasse vulnerabel ist, wird die gesamte Klasse als vulnerabel markiert, wodurch auch nicht-vulnerable Methoden derselben Klasse als betroffen klassifiziert werden. Zudem unterscheidet die Analyse derzeit nicht zwischen ECMAScript Modules (ESM) und CommonJS, sodass identifizierte Aufrufpfade zwischen verschiedenen Modulsystemen hin- und herspringen können, obwohl solche Übergänge in der Praxis nicht direkt möglich sind. Während diese konservativen Ansätze False-Negatives vermeiden, was sich in der perfekten Präzision von 100 % in der manuellen Verifikation widerspiegelt, schränken sie die Granularität der Analyse ein und können theoretisch zu unpräzisen Klassifikationen führen, die in der untersuchten Stichprobe jedoch nicht auftraten.

Eine der bedeutendsten Herausforderungen stellt das Mapping zwischen TypeScript-Quellcode und dem transpilierten JavaScript-Code dar. Viele NPM-Pakete werden in TypeScript entwickelt, veröffentlichen jedoch nur den kompilierten JavaScript-Code. Wenn die Schwachstellenanalyse auf dem TypeScript-Quellcode basiert, während Downstream-Konsumenten den JavaScript-Code verwenden, entstehen Diskrepanzen in Dateinamen, Zeilennummern und teilweise sogar in der Funktionsstruktur. Zur Lösung dieses Problems wurden verschiedene Ansätze untersucht. Ein vielversprechender Ansatz bestand in der Anwendung von Callgraph-Similarity-Metriken, um korrespondierende Funktionen zwischen TypeScript-Source und JavaScript-Build zu identifizieren. Dies erfordert die Lösung des Maximum Common Edge Subgraph Problems, für das eine Implementierung mittels Python NetworkX evaluiert wurde. Allerdings erwies sich dieser Ansatz als unpraktikabel: Die Laufzeit war hoch, und die durch den Transpilierungsprozess entstehenden strukturellen Unterschiede zwischen den Graphen waren zu erheblich, um zuverlässige Mappings zu ermöglichen. Auch heuristische Ansätze, ausschließlich basierend auf Dateinamen lieferten keine zufriedenstellenden Ergebnisse.

Die zuletzt implementierte Lösung basiert auf Source Maps, die von vielen TypeScript-Projekten generiert werden und präzise Mappings zwischen transpiliertem JavaScript und TypeScript-Source bereitstellen. Allerdings sind Source Maps nicht immer in veröffentlichten NPM-Paketen enthalten, was diese Lösung auf eine Teilmenge der analysierten Pakete beschränkt. Mehrere False-Negatives in der Stichprobe, darunter `@asyncapi/modelina` und `vue-i18n`, sind auf fehlendes oder inkorrektes TypeScript-zu-JavaScript-Mapping zurückzuführen. Es bleibt anzumerken, dass nicht nur die Transpilierung von TypeScript-zu-JavaScript sondern jegliche Transpilierung, bspw. zwischen ECMA-Script-Versionen

oder das Minifizieren von JavaScript-Quellcode, zu methodischen Herausforderungen führt.

Das Programm konnte zum Zeitpunkt der Datenerhebung nicht mit gebündelten Abhängigkeiten (`bundledDependencies`) umgehen. Bei regulären NPM-Paketen werden Abhängigkeiten zur Installationszeit aus der NPM-Registry heruntergeladen und im `node_modules`-Ordner abgelegt. Pakete mit `bundledDependencies` hingegen bündeln ausgewählte oder alle Abhängigkeiten direkt im veröffentlichten Paket-Tarball. Dies geschieht häufig, wenn Paketautoren die Verfügbarkeit bestimmter Abhängigkeiten sicherstellen wollen oder wenn ein Paket mit einem Bundler wie webpack oder esbuild kompiliert wurde. Die gebündelten Abhängigkeiten werden nicht separat in `node_modules` installiert, sondern befinden sich bereits kompiliert oder vorinstalliert im Paketverzeichnis selbst. Prominente Beispiele sind `@miniflare/core`, das ein gebündeltes `undici`-Paket enthält, oder Pakete, die mit Bundlern wie webpack oder esbuild erstellt wurden. Die Analyse solcher Pakete führte zu False-Negatives, da die Abhängigkeitsstruktur nicht korrekt aufgelöst werden konnte. Nachträglich wurde Support für gebündelte Abhängigkeiten implementiert, allerdings wurden die Analyseergebnisse nicht neu generiert, weshalb diese Limitation in der präsentierten Datenbasis noch vorhanden ist.

Wie bereits in Abschnitt 4.9 diskutiert, stellen dynamische JavaScript-Konstrukte eine fundamentale Herausforderung für statische Analysen dar. Die manuelle Verifikation bestätigte mehrere Fälle, in denen dynamische Patterns die Analyse beeinträchtigten. Beispielsweise nutzt `@twilio-labs/serverless-runtime-types` `Object.defineProperty` mit Getter-Funktionen, um Properties dynamisch zu definieren, was von der statischen Analyse nicht aufgelöst werden konnte. Ähnlich führte die event-getriebene Architektur in `@wdio/local-runner`, bei der Funktionen über `process.on('message')` dynamisch aufgerufen werden, zu False-Negatives.

Mehrere Fehler waren nicht auf Limitationen der Analysemethodik zurückzuführen, sondern auf unvollständige oder inkorrekte Metadaten in den Schwachstellendatenbanken. Bei `webpack-dev-middleware` beispielsweise war der in der CVE-Beschreibung angegebene Versionsbereich falsch, die tatsächlich installierte Version war nie vulnerabel. Das Programm identifizierte diese Diskrepanz korrekt, indem es feststellte, dass die in den Fix-Commits referenzierten Dateien in der analysierten Version nicht existierten, was zu einer korrekten Klassifikation als `not_affected` führte.

Eine systematischere Herausforderung entsteht, wenn eine Schwachstelle mehrere Pakete oder mehrere Versionsreihen desselben Pakets betrifft. In solchen Fällen existieren häufig unterschiedliche Fix-Commits für verschiedene Versionen oder sogar verschiedene betroffene Pakete. Die aktuelle Implementierung wählt deterministisch einen der in den CVE-Metadaten referenzierten Fix-Commits aus, ohne zu prüfen, ob dieser für die konkret analysierte Paketversion relevant ist. Dies führte zu False-Negatives bei `cypress`, `ms-rest-azure` und `@vercel/routing-utils`, wo die CVE-Einträge Fix-Commits für Version 3.x

enthielten, während die analysierten Pakete Versionen aus anderen Reihen (z.B. 2.x oder 5.x) verwendeten. Die aus dem unpassenden Fix-Commit extrahierten vulnerablen Code-Bereiche stimmten nicht mit der Struktur der tatsächlich verwendeten Version überein, was dazu führte, dass keine vulnerablen Funktionen identifiziert wurden. Dieses Problem wurde auch vor Neuevaluierung der Stichprobe nicht behoben.

Eine robustere Implementierung müsste für jede analysierte Paketversion den zugehörigen Fix-Commit identifizieren, beispielsweise durch Abgleich der Versionsnummern oder durch Analyse aller referenzierten Commits, um denjenigen zu finden, der die tatsächlich vorhandene Code-Struktur modifiziert. Diese Verbesserung würde die False-Negative-Rate bei Schwachstellen mit versionsabhängigen Fix-Commits erheblich reduzieren.

6. Diskussion

Die vorliegende Arbeit belegt eine deutliche Dominanz transitiver Abhängigkeiten gegenüber direkten Abhängigkeiten innerhalb des NPM-Ökosystems. Mit einem durchschnittlichen Verhältnis von etwa 1:19 bestätigt und verstärkt dieser Befund frühere Beobachtungen aus dem Jahr 2019 [55] und unterstreicht die wachsende Komplexität moderner Softwareprojekte, die zunehmend durch indirekte Abhängigkeitsstrukturen geprägt sind. Die empirische Untersuchung zeigt zudem, dass gängige metadatenbasierte Schwachstellenscanner wie `npm audit` eine False-Positive-Rate von etwa 70 % aufweisen. Diese hohe Rate an Fehldetektionen führt zu einer ineffizienten Ressourcenallokation, da Entwickler aufgrund regulatorischer Vorgaben wie der ISO 27001 [23], dem Cyber Resilience Act [13] oder dem PCI-DSS [43] verpflichtet sind, alle gemeldeten Schwachstellen zu bewerten. Da die tatsächliche Relevanz vor der Bewertung unbekannt ist, muss jede Meldung mit gleichem Aufwand untersucht werden, unabhängig davon, ob sie letztlich eine reale Bedrohung darstellt oder nicht.

Sowohl aktuelle Forschungsergebnisse als auch Empfehlungen der Organisation FIRST betonen, dass eine umfassende Schwachstellenbewertung über den reinen CVSS-Base-Score hinausgehen muss [15, 16, 26]. Der Base Score erfasst ausschließlich die intrinsischen Eigenschaften einer Schwachstelle, wie die Komplexität ihrer Ausnutzung oder die potenziellen Auswirkungen, vernachlässigt jedoch zentrale kontextabhängige Faktoren wie die tatsächliche Erreichbarkeit des vulnerablen Codes oder die spezifischen Sicherheitsanforderungen der Einsatzumgebung. Die Berücksichtigung solcher kontextspezifischer Faktoren und dynamischer Risikometriken erhöht zwar den Bewertungsaufwand, ermöglicht jedoch eine realistischere Risikoeinschätzung, die der tatsächlichen Bedrohungslage näher kommt [15, 16].

Ein zentraler Beitrag dieser Arbeit besteht in der Entwicklung einer methodischen Pipeline für automatisierte Erreichbarkeitsanalysen, die den manuellen Prüfaufwand durch Identifikation von False-Positives deutlich reduziert. Die Pipeline integriert dabei mehrere methodische Komponenten: die automatisierte Extraktion von Security-Patches aus CVE-Metadaten auf Basis der vorangegangenen Arbeiten, insbesondere der Studie *CVEfixes: automated collection of vulnerabilities and their fixes from open-source software*[3], die Konstruktion von Callgraphen mittels npm-jelly [8], die Identifikation vulnerabler Symbole durch Überlappungsanalyse von Patch-Zeilen und Funktionsdefinitionen, sowie die systematische Vererbungsanalyse entlang aller Abhängigkeitspfade. Diese End-to-End-Automatisierung, vom initialen Schwachstellen-Scan bis zur Generierung strukturierter VEX-Statements, stellt einen wesentlichen Fortschritt gegen-

über dem aktuellen Stand der Technik dar und reduziert den manuellen Aufwand erheblich. Im Erfolgsfall liefert die Pipeline präzise Zeilenangaben der vulnerablen Aufrufe einschließlich des vollständigen Call-Stacks über Paketgrenzen hinweg, was eine gezielte manuelle Verifikation ermöglicht. Im Fall von False-Positives erfolgt eine automatische Klassifikation als `not_affected`, da kein Aufrufpfad zu den vulnerablen Funktionen identifiziert werden konnte, wodurch diese Schwachstellen ohne weiteren manuellen Aufwand aus der Priorisierungsliste entfernt werden können. Auch bei einer nachgelagerten manuellen Verifikation des `not_affected`-Zustands unterstützen die Ergebnisse der Pipeline die Bewertung erheblich, da die bereits identifizierten vulnerablen Symbole der betroffenen Bibliothek die häufig kurz gehaltenen Schwachstellenbeschreibungen um konkrete technische Details, wie betroffene Funktionen und deren Position im Quellcode, ergänzen, was die manuelle Analyse beschleunigt.

Neben der automatisierten Pipeline ist ein weiterer Beitrag dieser Arbeit die Erhebung von empirischen Daten in Bezug auf die Schwachstellenvererbung. So belegen die erhobenen Daten eine exponentielle Abnahme der Propagierungswahrscheinlichkeit mit zunehmender Tiefe im Abhängigkeitsbaum. Von 603.925 vererbten vulnerablen Symbolen überspringen 88,6 % eine einzelne Paketgrenze, während nur 0,7 % vier oder mehr Grenzen überwinden. Dieses Verhalten bestätigt analoge Beobachtungen im Maven-Ökosystem [31] und deutet auf ein universelles Muster in paketbasierten Softwareökosystemen hin. Die praktische Implikation ist bedeutsam: Schwachstellen in tiefen transitiven Abhängigkeiten stellen statistisch ein geringeres Risiko dar als jene in direkten oder weniger tiefen Abhängigkeiten.

Ein weiterer schwachstellenunabhängiger Prädiktor für die Vererbungswahrscheinlichkeit wurde durch die Korrelation zwischen der Anzahl exportierter APIs einer Bibliothek und der Vererbungswahrscheinlichkeit von Schwachstellen identifiziert. Die empirischen Daten zeigen, dass Bibliotheken mit kompakteren API-Oberflächen ein erhöhtes Vererbungsrisiko aufweisen. Dies lässt sich dadurch erklären, dass vulnerable Funktionen in solchen Bibliotheken proportional häufiger einen signifikanten Teil der Gesamtfunktionalität ausmachen und daher mit größerer Wahrscheinlichkeit von Downstream-Komponenten genutzt werden.

Für die praktische Umsetzung der gewonnenen Erkenntnisse schlägt diese Arbeit vor, die identifizierten Metriken, Abhängigkeitstiefe und API-Größe, in bestehende Analysewerkzeuge und Risikobewertungssysteme zu integrieren. Ein entscheidender Vorteil dieser Metriken liegt in ihrer einfachen Verfügbarkeit und Berechenbarkeit: Die Abhängigkeitstiefe ist typischerweise bereits in einer Software Bill of Materials (SBOM) enthalten, während die API-Größe mit deutlich geringerem Ressourcenaufwand als eine vollständige Erreichbarkeitsanalyse ermittelt werden kann. Beide Metriken lassen sich somit unabhängig von der entwickelten Pipeline nutzen und bieten eine kostengünstige und skalierbare Möglichkeit, die Risikopriorisierung von Schwachstellen zu verbessern. Dies ermöglicht es Organisationen, bereits mit minimalem Aufwand eine grundlegende, kontextsensitive Sicherheitsbewertung durchzuführen und ihre Ressourcen gezielt auf die kritischs-

ten Abhängigkeiten zu konzentrieren. Eine formale Integration dieser Metriken könnte beispielsweise über den CVSS Environmental Score (CVSS-BE) erfolgen, der explizit für die Berücksichtigung umgebungsspezifischer Faktoren konzipiert wurde. Ein Paket mit einer Schwachstelle in einer direkten Abhängigkeit erhielte dann einen höheren Environmental Score als dasselbe Paket mit derselben Schwachstelle in Abhängigkeitstiefe 5.

Während aktuelle Entwicklungen vorwiegend auf die Erweiterung einzelner Schwachstellenscanner um VEX-Unterstützung abzielen, wäre eine Integration in übergeordnete Schwachstellenmanagement-Systeme der nachhaltigere und skalierbare Ansatz. Solche Systeme könnten die Ergebnisse verschiedener Scanner konsolidieren, automatisch mit VEX-Daten anreichern und eine einheitliche Risikobewertung über die gesamte Softwarelandschaft eines Unternehmens hinweg ermöglichen. Besonders vielversprechend ist dabei die unternehmensübergreifende Nutzung von VEX-Daten, da die Erreichbarkeitsanalyse bis zur direkten Abhängigkeit vor der eigentlichen Anwendung unternehmensunabhängig durchgeführt werden kann. Dies eröffnet Potenzial für kooperative Sicherheitsanalysen, bei denen gemeinsame Abhängigkeitspfade bewertet und geteilt werden. Eine solche Zusammenarbeit würde nicht die Effizienz der Sicherheitsbewertung weiter steigern.

Langfristig sollte jedoch auch der Einsatz solcher Analysen, wenngleich sie automatisiert oder kooperativ erfolgen, nur als Übergangslösung dienen. Die optimale Lösung bestünde darin, dass Upstream-Maintainer selbst standardisierte VEX-Dokumente mit Erreichbarkeitsinformationen bereitstellen. Diese Verlagerung der Sicherheitsverantwortung zu den Komponentenentwicklern, dort, wo die größte Expertise über die interne Funktionsweise der Software besteht, würde die Effizienz des gesamten Sicherheitsprozesses erheblich steigern und die festgestellten Limitierungen einer statischen Code-Analyse umgehen. Paketautoren könnten bei der Veröffentlichung neuer Versionen gleichzeitig VEX-Statements für bekannte Schwachstellen erstellen, die präzise dokumentieren, welche exportierten Funktionen betroffen sind. Dies würde nachgelagerte Analysen in vielen Fällen überflüssig machen und zu einer konsistenteren Sicherheitskommunikation im Ökosystem führen.

Die entwickelten Metriken und Erkenntnisse bieten damit sowohl für die Forschung als auch für die Praxis wertvolle Ansatzpunkte zur Verbesserung des Schwachstellenmanagements in modernen Softwareökosystemen.

7. Zusammenfassung und Ausblick

Die vorliegende Arbeit adressiert das Problem hoher False-Positive-Raten von metadatenbasierten Schwachstellenscannern in der Schwachstellenidentifikation des NPM-Ökosystems. Durch die Kombination von Security-Patch-Commit-Mining, Callgraph-Analyse und systematischer Vererbungsanalyse wurde eine automatisierte Pipeline entwickelt, die bestimmen kann, welche Schwachstellen transitiver Abhängigkeiten tatsächlich über die öffentliche API erreichbar sind.

Die empirische Evaluation auf 172.177 Paketversionen zeigt, dass etwa 70 % der durch metadatenbasierte Scanner gemeldeten Schwachstellen nicht erreichbar sind und somit keine reale Bedrohung darstellen. Darüber hinaus wurden zwei zentrale Muster der Schwachstellenpropagierung identifiziert: die exponentielle Abnahme mit zunehmender Abhängigkeitstiefe und die Korrelation zwischen API-Größe und Vererbungswahrscheinlichkeit.

Die entwickelte Pipeline generiert automatisiert VEX-Statements, die über eine HTTP-Schnittstelle in standardisierten Formaten (OpenVEX und CycloneDX) bereitgestellt werden. Die Integration in bestehende Scanner wie Trivy wurde exemplarisch demonstriert, wobei sich OpenVEX aufgrund geringerer Kopplungen als praktikabler für automatisierte Arbeitsabläufe erwies. Die manuelle Verifikation einer Stichprobe offenbarte noch Verbesserungspotenzial, unterstrich jedoch gleichzeitig das Potenzial der Methodik.

Bei der Weiterentwicklung des Prototyps sollten die Optimierung der Patch-Extraktion für Schwachstellen mit versionsabhängigen Fix-Commits, die Verbesserung des TypeScript-zu-JavaScript-Mappings mittels Source Maps und die Erweiterung der Semgrep-Regeln für komplexe dynamische Konstrukte priorisiert werden. Die Integration fortschrittlicher Methoden zur Identifikation von Patch-Commits, wie sie in der Arbeit *MoreFixes: A Large-Scale Dataset of CVE Fix Commits Mined through Enhanced vorgeschlagen werden* [1], könnte die Abdeckung der Schwachstellen, welche von der Pipeline aufgrund vorhandener Security-Patch-Informationen analysiert werden können, weiter erhöhen.

Langfristig könnte diese Arbeit die Entwicklung adaptiver Sicherheitswerkzeuge anregen, die Entwickler bei der gezielten Priorisierung von Schwachstellen unterstützen. Bereits heute lassen sich die gewonnenen Erkenntnisse zur Abhängigkeitstiefe und API-Größe direkt in der Praxis anwenden, um die Risikopriorisierung zu optimieren und zu einem ressourcenschonenderen Schwachstellenmanagement beizutragen.

Literatur

[1] J. Akhoundali, S. R. Nouri, K. Rietveld und O. Gadyatskaya. „MoreFixes: A Large-Scale Dataset of CVE Fix Commits Mined through Enhanced Repository Discovery“. In: *Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering*. 2024, S. 42–51 (siehe S. 26, 38, 78).

[2] G. Antal, P. Hegedűs, Z. Herczeg, G. Lóki und R. Ferenc. „Is JavaScript Call Graph Extraction Solved Yet? A Comparative Study of Static and Dynamic Tools“. In: *IEEE Access* 11 (2023), S. 25266–25284. DOI: 10.1109/ACCESS.2023.3255984 (siehe S. 29).

[3] G. Bhandari, A. Naseer und L. Moonen. „CVEfixes: automated collection of vulnerabilities and their fixes from open-source software“. In: *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE 2021. Athens, Greece: Association for Computing Machinery, 2021, 30–39. ISBN: 9781450386807. DOI: 10.1145/3475960.3475985. URL: https://doi.org/10.1145/3475960.3475985 (siehe S. 12, 17, 26, 27, 37, 38, 75).

[4] *BOM-Link | CycloneDX*. [Online; accessed 2025-12-02]. URL: https://cyclonedx.org/capabilities/bomlink/ (siehe S. 66).

[5] Bundesamt für Sicherheit in der Informationstechnik (BSI). *IT-Grundschutz-Kompendium, Edition 2023*. BSI, 2023. URL: https://www.bsi.bund.de (siehe S. 14).

[6] G. Canfora, A. Di Sorbo, S. Forootani, M. Martinez und C. A. Visaggio. „Patchworking: Exploring the code changes induced by vulnerability fixing activities“. In: *Information and Software Technology* 142 (2022), S. 106745. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2021.106745. URL: https://www.sciencedirect.com/science/article/pii/S0950584921001932 (siehe S. 27).

[7] B. Chinthanet, S. E. Ponta, H. Plate, A. Sabetta, R. G. Kula, T. Ishio und K. Matsumoto. „Code-Based Vulnerability Detection in Node.js Applications: How far are we?“ In: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2020, S. 1199–1203 (siehe S. 18, 31).

[8] *cs-au-dk/jelly: JavaScript/TypeScript static analyzer for call graph construction, library usage pattern matching, and vulnerability exposure analysis*. [Online; accessed 2025-10-05]. 2025. URL: https://github.com/cs-au-dk/jelly (siehe S. 28, 40, 75).

[9] *CVE: Common Vulnerabilities and Exposures*. [Online; accessed 2025-04-27]. URL: https://www.cve.org/About/Metrics (siehe S. 7).

[10] M. Eichberg und B. Hermann. „A software product line for static analyses: the OPAL framework“. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. SOAP '14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, 1–6. ISBN: 9781450329194. DOI: 10.1145/2614628.2614630. URL: https://doi.org/10.1145/2614628.2614630 (siehe S. 31).

[11] Y. C. M. Ekstedt. *Vexed by VEX tools: Consistency evaluation of container vulnerability scanners*. 2025. arXiv: 2503.14388 [cs.CR]. URL: https://arxiv.org/abs/2503.14388 (siehe S. 23).

[12] R. Elizalde Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto und A. Ihara. „Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages“. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018, S. 559–563. DOI: 10.1109/ICSME.2018.00067 (siehe S. 8, 53).

[13] European Commission. *Proposal for a Regulation of the European Parliament and of the Council on horizontal cybersecurity requirements for products with digital elements*. Techn. Ber. COM(2022) 454 final. 2022 (siehe S. 7, 14, 75).

[14] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby und F. Tip. „Efficient construction of approximate call graphs for JavaScript IDE services“. In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, S. 752–761. DOI: 10.1109/ICSE.2013.6606621 (siehe S. 28, 40).

[15] FIRST. *CVSS v3.1 User Guide*. [Online; accessed 2025-12-06]. URL: https://www.first.org/cvss/v3-1/user-guide (siehe S. 15, 75).

[16] FIRST. *CVSS v4.0 User Guide*. [Online; accessed 2025-12-06]. URL: https://www.first.org/cvss/v4-0/user-guide (siehe S. 15, 75).

[17] Forum of Incident Response and Security Teams, Inc. *Exploit Prediction Scoring System (EPSS)*. [Online; accessed 2025-11-08]. URL: https://www.first.org/epss/ (siehe S. 15).

[18] Forum of Incident Response and Security Teams, Inc. *CVSS v3.0 Specification Document*. [Online; accessed 2025-11-08]. Juni 2015. URL: https://www.first.org/cvss/v3-0/specification-document (siehe S. 13).

[19] GitHub, Inc. *GitHub REST API documentation - GitHub Docs*. [Online; accessed 2025-10-05]. Nov. 2022. URL: https://docs.github.com/en/rest (siehe S. 39).

[20] D. Helm, S. Keidel, A. Kampkötter, J. Düsing, T. Roth, B. Hermann und M. Mezini. „Total Recall? How Good Are Static Call Graphs Really?“ In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2024. Vienna, Austria: Association for Computing Machinery, 2024, 112–123. ISBN: 9798400706127. DOI: 10.1145/3650212.3652114. URL: https://doi.org/10.1145/3650212.3652114 (siehe S. 18, 19).

[21] A. Henig. *Breakdown: Widespread npm Supply Chain Attack Puts Billions of Weekly Downloads at Risk - Palo Alto Networks Blog*. [Online; accessed 2025-09-20]. Sep. 2025. URL: https://www.paloaltonetworks.com/blog/cloud-security/npm-supply-chain-attack/ (siehe S. 7).

[22] e. A. Hila Ramati Merav Bar. *Shai-Hulud 2.0 Supply Chain Attack: 25K+ Repos Exposed | Wiz Blog*. [Online; accessed 2025-12-06]. Nov. 2025. URL: https://www.wiz.io/blog/shai-hulud-2-0-ongoing-supply-chain-attack (siehe S. 7, 10, 36).

[23] *Informationssicherheit, Cybersicherheit und Datenschutz - Informationssicherheitsmanagementsysteme - Anforderungen*. Standard. International Organization for Standardization, 2022 (siehe S. 7, 14, 75).

[24] R. Kikas, G. Gousios, M. Dumas und D. Pfahl. „Structure and Evolution of Package Dependency Networks“. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 2017, S. 102–112. DOI: 10.1109/MSR.2017.55 (siehe S. 7, 8).

[25] M. Kluban, M. Mannan und A. Youssef. „On Detecting and Measuring Exploitable JavaScript Functions in Real-world Applications“. In: *ACM Trans. Priv. Secur.* 27.1 (Feb. 2024). ISSN: 2471-2566. DOI: 10.1145/3630253. URL: https://doi.org/10.1145/3630253 (siehe S. 42).

[26] V. Koscinski, M. Nelson, A. Okutan, R. Falso und M. Mirakhorli. *Conflicting Scores, Confusing Signals: An Empirical Study of Vulnerability Scoring Systems*. 2025. arXiv: 2508.13644 [cs.CR]. URL: https://arxiv.org/abs/2508.13644 (siehe S. 15, 16, 75).

[27] R. Lakshmanan. *Self-Replicating Worm Hits 180+ npm Packages to Steal Credentials in Latest Supply Chain Attack*. [Online; accessed 2025-09-20]. Sep. 2025. URL: https://thehackernews.com/2025/09/40-npm-packages-compromised-in-supply.html (siehe S. 7, 10, 36).

[28] M. R. Laursen, W. Xu und A. Møller. „Reducing Static Analysis Unsoundness with Approximate Interpretation“. In: *Proc. ACM Program. Lang.* 8.PLDI (Juni 2024). DOI: 10.1145/3656424. URL: https://doi.org/10.1145/3656424 (siehe S. 28, 29, 40).

[29] P. Maj, S. Muroya, K. Siek, L. Di Grazia und J. Vitek. „The Fault in Our Stars: Designing Reproducible Large-scale Code Analysis Experiments“. In: *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Hrsg. von J. Aldrich und G. Salvaneschi. Bd. 313. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 27:1–27:23. ISBN: 978-3-95977-341-6. DOI: 10.4230/LIPIcs.ECOOP.2024.27. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2024.27 (siehe S. 28, 40).

[30] M. D. McIlroy, E. N. Pinson und B. A. Tague. „UNIX Time-Sharing System: Foreword“. In: *Bell System Technical Journal* 57.6 (1978), S. 1899–1904. DOI: https://doi.org/10.1002/j.1538-7305.1978.tb02135.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.1538-7305.1978.tb02135.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1978.tb02135.x (siehe S. 56).

[31] A. M. Mir, M. Keshani und S. Proksch. „On the Effect of Transitivity and Granularity on Vulnerability Propagation in the Maven Ecosystem“. In: *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2023, S. 201–211. DOI: 10.1109/SANER56733.2023.00028 (siehe S. 9, 10, 17, 31, 54, 76).

[32] A. Møller, B. B. Nielsen und M. T. Torp. „Detecting locations in JavaScript programs affected by breaking library changes“. In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428255. URL: https://doi.org/10.1145/3428255 (siehe S. 28, 29, 40, 43, 44).

[33] A. A. Y. Mussa, Y. Malaiya und I. Ray. „Using Attack Surface Entry Points and Reachability Analysis to Assess the Risk of Software Vulnerability Exploitability“. In: Jan. 2014, S. 1–8. DOI: 10.1109/HASE.2014.10 (siehe S. 8).

[34] *nice-registry/download-counts: Average daily download counts for every npm package. Works offline.* [Online; accessed 2025-10-04]. URL: https://github.com/nice-registry/download-counts (siehe S. 34).

[35] B. B. Nielsen, M. T. Torp und A. Møller. „Modular call graph construction for security scanning of Node.js applications“. In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2021. Virtual, Denmark: Association for Computing Machinery, 2021, 29–41. ISBN: 9781450384599. DOI: 10.1145/3460319.3464836. URL: https://doi.org/10.1145/3460319.3464836 (siehe S. 20, 28, 40, 42, 44).

[36] *npm Blog Archive: Details about the event-stream incident.* [Online; accessed 2025-04-27]. URL: https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident (siehe S. 7).

[37] npm, Inc. *npm-audit*. [Online; accessed 2025-11-04]. 2024. URL: https://docs.npmjs.com/cli/v10/commands/npm-audit (siehe S. 14).

[38] *NVD - National Vulnerability Database*. [Online; accessed 2025-11-02]. URL: https://nvd.nist.gov/ (siehe S. 12).

[39] OASIS CSAF Technical Committee. *Common Security Advisory Framework Version 2.0*. OASIS Standard. OASIS, 2022. URL: https://docs.oasis-open.org/csaf/csaf/v2.0/os/csaf-v2.0-os.html (siehe S. 23).

[40] Oliver Chang and Kim Lewandowski, Google Security Team. *Google Online Security Blog: Launching OSV - Better vulnerability triage for open source*. [Online; accessed 2025-11-02]. Feb. 2021. URL: https://security.googleblog.com/2021/02/launching-osv-better-vulnerability.html (siehe S. 13).

[41] OpenVEX Community. *OpenVEX Specification v0.2.0*. [Online; accessed 2025-09-20]. März 2025. URL: https://github.com/openvex/spec/blob/main/OPENVEX-SPEC.md (siehe S. 23).

[42] *OSV - Open Source Vulnerabilities*. [Online; accessed 2025-04-27]. URL: https://osv.dev/ (siehe S. 10, 14, 37).

[43] PCI Security Standards Council, LLC. *Payment Card Industry Data Security Standard (PCI DSS), Version 4.0*. 2022. URL: https://www.pcisecuritystandards.org (siehe S. 14, 15, 75).

[44] H. Plate, S. E. Ponta und A. Sabetta. „Impact assessment for vulnerabilities in open-source software libraries“. In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2015, S. 411–420. DOI: 10.1109/ICSM.2015.7332492 (siehe S. 8, 17, 26).

[45] S. Ponta, H. Plate und A. Sabetta. „Detection, assessment and mitigation of vulnerabilities in open source dependencies“. In: *Empirical Software Engineering* 25 (Sep. 2020). DOI: 10.1007/s10664-020-09830-x (siehe S. 15–17, 26, 30).

[46] J. Qiu. *Vulnerability Management for Go - The Go Programming Language*. [Online; accessed 2025-09-07]. Sep. 2022. URL: https://go.dev/blog/vuln (siehe S. 10).

[47] nice registry. *GitHub - nice-registry/all-the-package-names: A list of all the public package names on npm. Updated daily.* [Online; accessed 2025-04-27]. URL: https://github.com/nice-registry/all-the-package-names (siehe S. 7).

[48] B. Ryder. „Constructing the Call Graph of a Program“. In: *IEEE Transactions on Software Engineering* SE-5.3 (1979), S. 216–226. DOI: 10.1109/TSE.1979.234183 (siehe S. 18, 19).

[49] J. Spring, E. Hatleback, A. Householder, A. Manion und D. Shick. „Time to Change the CVSS?“ In: *IEEE Security & Privacy* 19.2 (2021), S. 74–78. DOI: 10.1109/MSEC.2020.3044475 (siehe S. 15).

[50] N. I. of Standards und Technology. *Security Content Automation Protocol | CSRC*. [Online; accessed 2025-12-06]. 2025. URL: https://csrc.nist.gov/projects/security-content-automation-protocol/specifications/cpe (siehe S. 13).

[51] VEX Working Group coordinated by the Cybersecurity & Infrastructure Security Agency. *Minimum Requirements for Vulnerability Exploitability eXchange (VEX)*. [Online; accessed 2025-09-20]. Apr. 2023. URL: https://www.cisa.gov/sites/default/files/2023-04/minimum-requirements-for-vex-508c.pdf (siehe S. 23).

[52] X. Wang, K. Sun, A. Batcheller und S. Jajodia. „Detecting "0-Day"Vulnerability: An Empirical Study of Secret Security Patch in OSS“. In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2019, S. 485–492. DOI: 10.1109/DSN.2019.00056 (siehe S. 27).

[53] X. Wang, S. Wang, P. Feng, K. Sun, S. Jajodia, S. Benchaaboun und F. Geck. „PatchRNN: A Deep Learning-Based System for Security Patch Identification“. In: *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*. IEEE, Nov. 2021, 595–600. DOI: 10.1109/milcom52596.2021.9652940. URL: http://dx.doi.org/10.1109/MILCOM52596.2021.9652940 (siehe S. 27).

[54] B. Wu, S. Liu, R. Feng, X. Xie, J. Siow und S.-W. Lin. „Enhancing Security Patch Identification by Capturing Structures in Commits“. In: *IEEE Transactions on Dependable and Secure Computing* (2022), S. 1–15. DOI: 10.1109/TDSC.2022.3192631 (siehe S. 27).

[55] M. Zimmermann, C.-A. Staicu, C. Tenny und M. Pradel. „Smallworld with high risks: a study of security threats in the npm ecosystem“. In: *Proceedings of the 28th USENIX Conference on Security Symposium*. SEC'19. Santa Clara, CA, USA: USENIX Association, 2019, 995–1010. ISBN: 9781939133069 (siehe S. 7, 8, 52, 53, 75).

[56] F. Zuo und J. Rhee. „Vulnerability discovery based on source code patch commit mining: a systematic literature review“. In: *International Journal of Information Security* 23.2 (Jan. 2024), 1513–1526. ISSN: 1615-5270. DOI: 10.1007/s10207-023-00795-8. URL: http://dx.doi.org/10.1007/s10207-023-00795-8 (siehe S. 17, 27).

A. VEX-Dokumente

Listing A.1: Auschnitt eines VEX-Dokuments für ein OCI-Container Image

```
{
 "bomFormat": "CycloneDX",
 "specVersion": "1.6",
 "metadata": {
  "timestamp": "2025-09-20T10:13:55Z",
  "component": {
   "type": "application",
   "author": "L3montree Cybersecurity",
   "publisher": "github.com/l3montree-dev/devguard",
   "name": "pkg:oci/devguard?repository_url=ghcr.io/
      l3montree/devguard",
   "version": "main"
  }
 },
 "vulnerabilities": [{
   "id": "\gls{CVE}-2025-50181",
   "source": {
    "url": "https://nvd.nist.gov/vuln/detail/\gls{CVE
       }-2025-50181"
   },
   "ratings": [{
     "score": 5.3,
     "severity": "medium",
     "method": "CVSSv31"
    }],
   "analysis": {
    "state": "not_affected",
    "detail": "We are not calling the vulnerable
       functions from urllib3",
   },
   "affects": [{ "ref": "pkg:pypi/urllib3@1.26.20" }]
  }
 ]
}
```

Listing A.2: Beispiel-VEX-Statement des Servers

```
{
 "vulnerabilities": [
  {
   "id": "GHSA-grv7-fg5c-xmjg",
   "analysis": {
    "state": "exploitable",
    "detail": "Exploitable vulnerability found. pkg:\gls
       {NPM}/chokidar@3.6.0 -> pkg:\gls{NPM}/braces@3
       .0.3",
    "evidence": [{
      "confidence": 0.5,
      "path": [{
        "id": 1097262,
        "signature": "<chokidar>.watch",
        "filename": "index.js",
        "start_line": 967,
        "end_line": 971,
        "evidence": "[DIST] (<chokidar>.watch at index.
           js:967:15:971:2) -> [...] -> (<chokidar>.
           WatchHelper at index.js:204:1:278:2) -> (
           index.js:258:55:258:74) -> { see calls_id }",
        "purl": "pkg:\gls{NPM}/chokidar@3.6.0",
        "calls_id": 1097205,
        "triggeredRules": []
       },
       {
        "id": 1097205,
        "signature": "<braces>.expand",
        "filename": "index.js",
        "start_line": 120,
        "end_line": 138,
        "evidence": "(<braces>.expand at index.js
           :120:17:138:2) -> [...] -> (<braces/lib/parse
           >.parse at lib/parse.js:31:15:331:2) -> (lib/
           parse.js:308:0:308:Infinity https://github.
           com/micromatch/braces/commit/98414
           f9f1fabe021736e26836d8306d5de747e0d)",
        "purl": "pkg:\gls{NPM}/braces@3.0.3",
        "calls_id": null,
        "triggeredRules": ["limitations.dynamic-require"
           ]
...
}
```