INS | Institut für Netzwerke und Sicherheit

OST Eastern Switzerland University of Applied Sciences

KI Coding Assistenten und Sicherer Code: Möglichkeiten, Grenzen, Angriffe

Studiengang Informatik
OST – Ostschweizer Fachhochschule
Campus Rapperswil-Jona
Herbstsemester 2025

Autor: Eric Horsch

Betreuer:in: Dr. Arno Wagner, Alice Glaus

Contents

I Management Summary und Abstract **1**

II Product Documentation **6**

1 Ausgangslage und Stand der Technik **7**
1.1 Motivation 7
1.2 Stand der Technik 8

2 Zielsetzung und Forschungsfragen **9**
2.1 Übergeordnetes Ziel 9
2.2 Konkrete Forschungsfragen 9
2.3 Kritische Erfolgsfaktoren 9

3 Rahmenbedingungen **11**
3.1 Art der Arbeit 11
3.2 Zeitbudget und ECTS 11
3.3 Beschreibung der Arbeit 11
3.4 Erweiterungen 12

4 Methodik **13**
4.1 Vorgehen bei der Literaturrecherche 13
4.2 Vorgehen bei der Sammlung der Schwachstellen 14
4.2.1 Kleine Codebeispiele: 14
4.2.2 Bewusst unsichere Projekte: 14
4.2.3 Echte CVEs aus Python-Libraries: 15
4.2.4 Sichere Vergleichsprojekte: 15
4.3 Auswahl der LLMs und Coding-Assistenten 16
4.4 Prompt Engineering 16
4.5 Vorgehen bei den Versuchen 18
4.5.1 Dokumentation der Ergebnisse: 18
4.6 Modifikation von Code-Mustern: 18

5 Umsetzung der Tests **20**
5.1 Ergänzungen während der Tests 20
5.2 Fehler und Abweichungen 20

6 Ergebnisse **22**
6.1 Score-Vergleich der Modelle 22
6.2 Ergebnisse pro Modell 23
6.2.1 Amazon Q Developer 23
6.2.2 ChatGPT (GPT-5) 23
6.2.3 Claude Sonnet 4.5 23

6.2.4 Copilot (Claude Sonnet 4.5) . . . 24
6.2.5 Copilot (GPT-5) . . . 24
6.2.6 Qwen3-Coder-30B . . . 25
6.2.7 Tabnine Dev (Protected Model) . . . 25
6.3 Gesamtvergleich . . . 25
6.3.1 LLMs vs. Coding-Assistenten . . . 25
6.3.2 Vergleich gleiches LLM vs. Coding-Assistent . . . 26
6.3.3 Typische Schwächen über alle Modelle . . . 26
6.3.4 Kernaussagen (Ergebnisse) . . . 26
6.4 Ergebnisse zu Code-Mustern und Obfuskation . . . 27
6.4.1 ChatGPT (GPT-5) . . . 27
6.4.2 Copilot (GPT-5) . . . 27

7 Diskussion **28**
7.1 Einordnung der Ergebnisse . . . 28
7.2 LLMs vs. Coding-Assistenten . . . 29
7.3 Vergleich gleiches LLM vs. Copilot-Integration . . . 29
7.4 Typische Schwächen und Ursachen . . . 30
7.5 Umgehung durch Code-Muster und Obfuskation . . . 31
7.6 Implikationen für den praktischen Einsatz . . . 32
7.7 Zusammenfassung . . . 33

8 Schlussfolgerungen und Ausblick **34**
8.1 Schlussfolgerungen . . . 34
8.2 Ausblick . . . 35
8.3 Schlussbemerkung . . . 35

Quellenverzeichnis **36**

Abbildungsverzeichnis **38**

Tabellenverzeichnis **39**

Abkürzungsverzeichnis

.py Python-Dateiendung. 1

AWS Amazon Web Services. 1

CVE Common Vulnerabilities and Exposures. 1

CWE Common Weakness Enumeration. 1

ECTS European Credit Transfer System. 1

IDE Integrated Development Environment. 1

IDOR Insecure Direct Object Reference. 1

JWT JSON Web Token. 1

KI Künstliche Intelligenz. 1

LLM Large Language Model. 1

OST Ostschweizer Fachhochschule. 1

OWASP Open Worldwide Application Security Project. 1

REST Representational State Transfer. 1

SBOM Software Bill of Materials. 1

SSRF Server-Side Request Forgery. 1

SSTI Server-Side Template Injection. 1

XSS Cross-Site Scripting. 1

Glossar

1-1 Match Direkte Übereinstimmung eines Codes mit einem bekannten Muster. 1

Benchmark Standardisierter Test zur Bewertung der Leistungsfähigkeit eines Modells. 1

Best Practice Bewährte Methode oder empfohlene Vorgehensweise zur sicheren Softwareentwicklung. 1

Chat-Funktion Dialogbasierter Interaktionsmodus eines LLMs. 1

Code Sanitation Bereinigung von Code, z. B. Entfernen von Kommentaren oder Hinweisen. 1

Coding-Assistent Werkzeug zur Unterstützung bei Codegenerierung, Analyse und Review. 1

CSRF Cross-Site Request Forgery; Angriff, der ungewollte Aktionen auslöst. 1

CSV-Formula-Injection Angriff, bei dem Tabellenkalkulationsprogramme Formeln ausführen. 1

De-Obfuskation Rückführung verschleierten Codes in eine verständliche Form. 1

Directory Traversal Angriff durch manipulierte Pfadangaben zum Zugriff auf fremde Dateien. 1

Erkennungsrate Anteil korrekt erkannter Schwachstellen. 1

False Negative Nicht erkannte tatsächliche Schwachstelle. 1

False Positive Fehlerhafte Meldung einer nicht existierenden Schwachstelle. 1

Hardening Gezielte Härtung von Software zur Reduktion von Angriffsflächen. 1

IDE-Plugin Erweiterung einer Entwicklungsumgebung, z. B. für KI-Unterstützung. 1

Insecure Deserialization Unsichere Deserialisierung, die Codeausführung ermöglichen kann. 1

Kontextverlust Verlust von Prompt-Informationen bei langen Eingaben. 1

Laufzeitüberprüfung Analyse eines Programms während der Ausführung. 1

Login-Prozess Ablauf zur Authentifizierung eines Benutzers. 1

Modellarchitektur Struktureller Aufbau eines KI-Modells. 1

Modellfamilie Gruppe verwandter KI-Modelle mit gemeinsamer Architektur. 1

Multi-File-Szenario Analyse über mehrere Dateien hinweg. 1

Obfuskation Verschleierung von Code zur Erschwerung der Analyse. 1

Obfuskationsstufe Grad der Verschleierung eines Codes. 1

Open-Source-Projekt Softwareprojekt mit öffentlich zugänglichem Quellcode. 1

OS Command Injection Einschleusen von Betriebssystembefehlen über unsichere Eingaben. 1

Ownership-Check Prüfung der Berechtigung eines Benutzers für eine Ressource. 1

Patch-Historie Historie sicherheitsrelevanter Updates einer Software. 1

Performance Leistungsfähigkeit eines Systems. 1

Prompt Eingabeaufforderung an ein KI-Modell. 1

Prompt Engineering Gestaltung und Optimierung von Prompts. 1

Rate Limiting Begrenzung der Anzahl von Anfragen pro Zeitspanne. 1

ReDoS Regular Expression Denial of Service. 1

Reflected XSS Reflektiertes Cross-Site Scripting. 1

Security Review Systematische Analyse von Code auf Schwachstellen. 1

Session Fixation Angriff, bei dem eine Session-ID vorgegeben wird. 1

Session Hijacking Übernahme einer gültigen Benutzersitzung. 1

Statische Analyse Analyse von Code ohne Ausführung. 1

Symlink Symbolische Verknüpfung im Dateisystem. 1

Part I

Management Summary und Abstract

Management Summary

Der Einsatz von Künstlicher Intelligenz in der Softwareentwicklung nimmt stark zu. Systeme wie GitHub Copilot, Tabnine oder große Sprachmodelle wie GPT-5 und Claude Sonnet 4.5 unterstützen Entwicklerinnen und Entwickler dabei, schneller und produktiver zu arbeiten. Sie schlagen Code vor, erkennen Fehler und helfen beim Verständnis komplexer Zusammenhänge. Doch neben der Produktivität stellt sich eine entscheidende Frage: Können diese Systeme auch sicherheitsrelevante Schwachstellen zuverlässig erkennen und beheben?

Methodik

Um diese Frage zu beantworten, wurden sieben verschiedene Modelle systematisch untersucht. Die Auswahl umfasste sowohl klassische Coding-Assistenten, die direkt in Entwicklungsumgebungen integriert sind, als auch große Sprachmodelle, die breiter einsetzbar sind. Die Modelle wurden gezielt dazu aufgefordert, sogenannte Security Reviews durchzuführen. Anschließend wurde gemessen, ob die Schwachstellen gefunden und ob sinnvolle Verbesserungsvorschläge geliefert wurden. Die Tests wurden in der Programmiersprache Python durchgeführt. Neben klassischen Schwachstellen (z.B. SQL-Injection, Cross-Site Scripting, Session Fixation) wurden auch reale CVEs und bewusst unsichere Projekte untersucht. Zusätzlich fanden Obfuskations-Tests statt, bei denen der Code absichtlich verschleiert wurde. Diese Tests wurden in kleinerem Rahmen durchgeführt (5 Snippets, 2 Modelle) und zeigten, dass die Erkennungsrate insbesondere bei stärkerer Obfuskation deutlich abnahm.

Verwendete KI-Systeme:

- GPT-5 (OpenAI)
- Claude Sonnet 4.5 (Anthropic)
- Qwen3-Coder-30B (Alibaba)
- Amazon Q Developer (Claude 4.5) (Amazon/Anthropic)
- Copilot GPT-5 (Microsoft/OpenAI)
- Copilot Claude 4.5 (Microsoft/Anthropic)
- Tabnine Dev Protected(Tabnine)

Resultate

Die Ergebnisse zeigen ein klares Muster. Standardisierte Schwachstellen wie die OWASP-Snippets wurden fast immer erkannt. Auch unsichere Projekte wurden mit hoher Trefferquote identifiziert. Sobald es jedoch um komplexere und kontextabhängige Probleme ging, ließen die Systeme deutlich nach. Reale CVEs blieben weitgehend unentdeckt. Die Qualität der Verbesserungsvorschläge war ebenfalls durchwachsen: Häufig blieben sie oberflächlich, unvollständig oder beschränkten sich auf allgemeine Empfehlungen, ohne konkreten Code zu liefern.

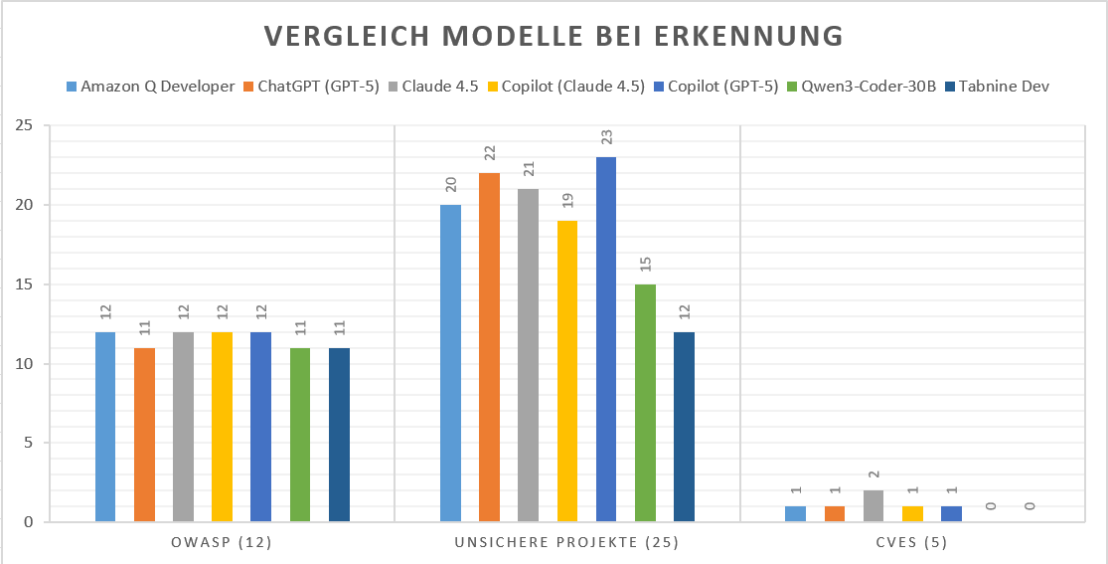

Figure 1: Anzahl erkannter Schwachstellen bei OWASP Snippets vs unsichere Projekte vs. echte CVEs

Ein weiterer wichtiger Befund betrifft die Unterschiede zwischen den Modelltypen. Große Sprachmodelle schnitten insgesamt etwas besser ab als klassische Coding-Assistenten. Der Vorsprung war jedoch nur marginal, da die Coding-Assistenten sehr gemischte Resultate lieferten. Besonders auffällig war, dass die Integration von Copilot mit GPT-5 solide Ergebnisse brachte, während die Kombination von Copilot mit Claude deutlich schwächer war. Tabnine erwies sich als ungeeignet für Security Reviews, da längere Codeabschnitte nicht verarbeitet werden konnten. Dies zog die Gesamtleistung der Coding-Assistenten nach unten. Bei den LLMs hingegen zeigten ChatGPT und Claude gute Ergebnisse, während das kleinere Qwen3-Coder-Modell mit 30 Milliarden Parametern deutlich schlechter abschnitt.

Schlussfolgerung

KI-gestützte Systeme sind wertvolle Helfer, wenn es darum geht, Entwicklerinnen und Entwickler zu entlasten und die Produktivität zu steigern. Sie können einfache Muster zuverlässig erkennen und damit als zusätzliche Reviewer dienen. Sie sind jedoch nicht geeignet, die alleinige Verantwortung für sichere Softwareentwicklung zu übernehmen. Gerade bei komplexen oder kontextabhängigen Schwachstellen ist menschliches Fachwissen unverzichtbar. Der Weg in die Zukunft liegt in einem hybriden Ansatz. KI-gestützte Systeme können die Arbeit beschleunigen und erste Hinweise liefern, doch die finale Bewertung und Absicherung muss durch erfahrene Entwicklerinnen und Entwickler erfolgen. Zukünftige Forschung sollte sich darauf konzentrieren, die Modelle robuster zu machen - etwa durch die Integration externer Wissensquellen wie CVE-Datenbanken, durch bessere semantische Analysen oder durch Mechanismen zur automatischen De-Obfuskation. Auch die Untersuchung anderer Programmiersprachen und anderer Modelle, und die Optimierung der Fragestellungen an die KI ("Prompt Engineering") sind vielversprechende Ansätze.

Abstract

Der Einsatz von Large Language Models (LLMs) und Coding-Assistenten wie GitHub Copilot oder Tabnine gewinnt zunehmend an Bedeutung in der Softwareentwicklung. Neben Produktivität stellt sich die Frage, ob diese Systeme auch sicherheitsrelevante Schwachstellen zuverlässig erkennen und beheben können. Bisherige Studien zeigen, dass KI-gestützte Systeme syntaktische Muster gut erfassen, aber bei semantischen und kontextabhängigen Sicherheitsproblemen Defizite haben. Es fehlt eine systematische Untersuchung, wie verschiedene Modelle mit klassischen Schwachstellen, realen CVEs oder obfuskiertem Code umgehen. In dieser Arbeit wurden sieben Modelle (GPT-5, Claude Sonnet 4.5, Qwen3-Coder-30B, Amazon Q Developer, Copilot GPT-5, Copilot Claude, Tabnine) anhand von OWASP-Snippets, bewusst unsicheren Projekten, realen CVEs und sicheren Bibliotheken getestet. Zusätzlich wurde die Robustheit gegenüber Code-Obfuskation (Namensänderung, manuelle und toolgestützte Obfuskation) untersucht. Die Ergebnisse zeigen, dass OWASP-Snippets fast immer erkannt wurden (11–12/12), unsichere Projekte mit hoher Quote, komplexe Schwachstellen wie RegexDOS, Rate Limiting und Session Fixation jedoch kaum. Reale CVEs blieben weitgehend unentdeckt (0–2/5). Verbesserungen wurden nur teilweise geliefert, oft oberflächlich oder als Empfehlungen ohne Code. Generelle LLMs schnitten insgesamt besser ab als Coding-Assistenten, während die IDE-Integrationen mehr False Positives und sprachliche Inkonsistenzen zeigten. Obfuskation führte insbesondere mit Tool zu deutlichen Erkennungsverlusten. Die Ergebnisse bestätigen, dass LLMs und Coding-Assistenten als unterstützende Reviewer geeignet sind, jedoch nicht als alleinige Instanz für sichere Softwareentwicklung. Zukünftige Forschung sollte sich auf die Integration externer Wissensquellen (z. B. CVE-Datenbanken), semantische Analysen und De-Obfuskationsmechanismen konzentrieren. Ein hybrider Ansatz aus KI-gestützter Analyse und menschlichem Review erscheint als vielversprechender Weg, um die Softwaresicherheit nachhaltig zu verbessern.

Part II

Product Documentation

Chapter 1

Ausgangslage und Stand der Technik

1.1 Motivation

Die Nutzung von *Large Language Models (LLMs)* hat sich seit 2023 rasant entwickelt und ist inzwischen ein zentraler Bestandteil vieler industrieller und akademischer Anwendungen. Laut aktuellen Marktanalysen wird der globale Markt für LLMs von 6,4 Milliarden US-Dollar im Jahr 2024 auf über 36 Milliarden US-Dollar im Jahr 2030 anwachsen. Diese Expansion wird durch Fortschritte in der Modellarchitektur, multimodale Fähigkeiten und die breite Integration in Unternehmenssoftware getrieben. Damit sind LLMs nicht mehr nur Forschungswerkzeuge, sondern haben sich zu unverzichtbaren Komponenten in Bereichen wie Finanzwesen, Gesundheitswesen und Softwareentwicklung entwickelt. [Rep25, Ana25]

Parallel dazu hat sich die Verbreitung von *Coding-Assistants* wie GitHub Copilot, ChatGPT oder Claude deutlich erhöht. Millionen von Entwicklern nutzen diese Werkzeuge, um Routineaufgaben zu automatisieren, komplexe Funktionen vorzuschlagen und die Entwicklungszeit zu verkürzen. Empirische Studien zeigen, dass die Akzeptanz dieser Tools durch wahrgenommene Produktivitätsgewinne und Erwartungsbestätigungstheorien gestützt wird. Auch die Interaktion mit solchen Assistenten verändert die Arbeitsweise von Programmierenden grundlegend, indem sie zunehmend auf dialogbasierte Schnittstellen setzen. [Kim25, Con25, Lee25]

Allerdings geht diese Entwicklung mit erheblichen *Sicherheitsrisiken* einher. LLM-generierter Code ist häufig funktional, aber nicht sicher. Besonders in sicherheitskritischen Bereichen wie Kryptografie, Backend-Entwicklung oder Webanwendungen entstehen gravierende Schwachstellen. Die steigende Nutzung solcher Tools bedeutet daher nicht nur Effizienzgewinne, sondern auch eine potenzielle *Vergrößerung der Angriffsfläche* in Softwaresystemen. Dies macht eine systematische Untersuchung der Code-Sicherheit zwingend erforderlich [Spe23, Hua24, Eng25].

1.2 Stand der Technik

Die Problematik unsicherer KI-generierter Software und Limitierungen von Coding-Assistenten im Bezug auf Code-Sicherheit ist bereits Gegenstand zahlreicher einschlägiger Arbeiten:

- **IEEE Spectrum (2023):** Erste kritische Analysen zeigen, dass Copilot und ChatGPT funktionalen, aber unsicheren Code erzeugen, insbesondere bei Kryptografie-Beispielen [Spe23].
- **IEEE Xplore (2024):** Eine großangelegte Studie mit 728 Programmieraufgaben in fünf Sprachen und 18 CWE-Szenarien belegt, dass ChatGPT bekannte Probleme besser löst, jedoch bei neueren Aufgaben und sicherheitsrelevanten Schwachstellen versagt [Hua24].
- **Empirical Software Engineering (2025):** Claude 3, GPT-4 und Llama 3 erkennen nur zwischen 12,6 % und 40 % der Schwachstellen in unsicherem Code [Eng25].
- **IFIP Conference Paper (2025):** Ohne gezielte Anleitung erzeugen LLMs bis zu 65 % unsicheren Code; mit manueller Steuerung durch Experten steigt die Sicherheit auf über 94 % [SP25].
- **ArXiv (2025):** Iteratives Verbessern durch LLMs kann die Zahl kritischer Schwachstellen um bis zu 37,6 % erhöhen [Zha25].
- **Veracode Report (2025):** Tests von über 100 LLMs zeigen, dass 45 % der generierten Samples durch OWASP-Top-10-Prüfungen fallen. Neuere Modelle schreiben zwar funktionaleren, aber nicht sichereren Code[Ver25].
- **Databricks (2025):** Auch hier wurde festgestellt, dass ungeleitete Code-Generierung funktional, aber unsicher bleibt [Dat25].
- **GitHub Copilot + JFrog (2025):** Erste Ansätze zur automatischen Absicherung durch Integration von Security-Tools in Coding-Assistants sind verfügbar [Hei25].
- **Systematisches Literaturreview (2025):** Im Kapitel "(In)secure code generation" wird konstatiert, dass LLMs zwar funktionalen Code erzeugen, jedoch ohne menschliches Review unsicher bleiben [Spr25].
- **Benchmarks wie BaxBench (2025):** Selbst die besten Modelle erreichen nur 62 % korrekten Code [Li25].

Diese Arbeiten verdeutlichen, dass die Sicherheitsproblematik breit dokumentiert ist. Sie liefern wertvolle Erkenntnisse über die Schwächen aktueller Systeme, zeigen aber auch, dass bislang keine umfassende Studie vorliegt, die diese Problematik genauer erklärt und begründet.

Chapter 2

Zielsetzung und Forschungsfragen

2.1 Übergeordnetes Ziel

Das übergeordnete Ziel dieser Arbeit ist die *systematische Bewertung der Code-Sicherheit von Large Language Models (LLMs) und Coding-Assistenten.* Dabei soll untersucht werden, welche Unterschiede zwischen spezialisierten Coding-Assistenten und allgemeinen LLMs bestehen. Die Arbeit verfolgt einen empirischen Ansatz und stützt sich auf eigene Experimente mit unsicherem Code, ergänzt durch eine Standortbestimmung anhand aktueller Literatur und Benchmarks.

2.2 Konkrete Forschungsfragen

Ausgehend von der Zielsetzung ergeben sich folgende Forschungsfragen:

1. **Wie zuverlässig erkennen Coding-Assistenten und allgemeine LLMs sicherheitsrelevante Schwachstellen?**
2. **Welche Unterschiede bestehen zwischen spezialisierten Coding-Assistenten (z. B. GitHub Copilot) und allgemeinen LLMs (z. B. ChatGPT, Claude)?**
3. **Welche typischen Muster unsicherer Vorschläge lassen sich aus Literatur und Experimenten ableiten?**
4. **Inwieweit können durch gezielte Modifikationen von Code-Mustern die Erkennungsfähigkeiten der Modelle umgangen werden?**

2.3 Kritische Erfolgsfaktoren

Für die erfolgreiche Durchführung der Arbeit sind folgende Faktoren entscheidend:

- **Auswahl und Zugang:** Verfügbarkeit und stabiler Zugriff auf die relevanten LLMs und Coding-Assistenten, einschließlich kostenpflichtiger Modelle.

- **Code Setup:** Konsistente Einrichtung der Testumgebung sowie die Erstellung einer Sammlung unsicheren Codes als Grundlage für die Experimente.
- **Methodik:** Systematisches Vorgehen bei der Durchführung der Experimente.
- **Ergebnisse:** Erhalten von Ergebnissen und Dokumentieren der Resultate im Hinblick auf die Forschungsfragen.

Chapter 3

Rahmenbedingungen

3.1 Art der Arbeit

Bei der vorliegenden Untersuchung handelt es sich um eine *Studienarbeit* im Bereich Cybersecurity. Der Fokus liegt auf einer Analyse der Code-Sicherheit von Large Language Models (LLMs) und Coding-Assistenten. Es werden bestehende Werkzeuge anhand definierter Testfälle evaluiert.

3.2 Zeitbudget und ECTS

Für die Durchführung der Arbeit wurde ein Zeitbudget von insgesamt **240 Stunden** veranschlagt. Dies entspricht **8 European Credit Transfer System (ECTS) Credits** an je 30 Stunden. Die Zeit wurde für Literaturrecherche, Konzeption der Experimente, Durchführung der Tests, Auswertung der Ergebnisse sowie die schriftliche Dokumentation eingeplant.

3.3 Beschreibung der Arbeit

Die Arbeit umfasst folgende Kernpunkte:

1. Literaturrecherche, Standortbestimmung und Analyse des Angebots auf dem Markt.
2. Auswahl und Untersuchung mehrerer Coding-Assistenten.
3. Auswahl einer Programmiersprache
4. Experimente mit unsicherem Code: Untersuchung, was Assistenten leisten und wo ihre Grenzen liegen.
5. Analyse unsicherer Vorschläge durch Assistenten anhand von Literatur, Mustern und eigenen Experimenten.
6. Vergleich von spezialisierten Coding-Assistenten mit allgemeinen LLMs.

3.4 Erweiterungen

Zusätzlich wurden Erweiterungen umgesetzt:

- **Vergleich mehrerer Assistenten:** Unterschiede in der Qualität und Sicherheit der generierten Vorschläge.
- **Modifikation von Code-Mustern:** Untersuchung, wie sich gezielte Änderungen auf die Erkennung durch LLMs auswirken.

Nicht umgesetzt, aber im Rahmen der Literatur diskutiert, wurden weitere Themen wie Model-Poisoning, Slopsquatting oder Sprachvergleiche.

Chapter 4

Methodik

Für die Untersuchung wurde **Python** als Programmiersprache gewählt, da sie weit verbreitet ist und zahlreiche sicherheitsrelevante Beispiele bietet. Grundlage der Experimente bildet eine Sammlung unsicherer Codebeispiele aus Literatur, Open-Source-Projekten und realen CVEs. Die Aufgabe der getesteten LLMs und Coding-Assistenten bestand darin, diesen Code im Rahmen von **Security Reviews** zu analysieren, Schwachstellen zu identifizieren und Verbesserungen vorzuschlagen. Auf die direkte Generierung neuen Codes durch die Modelle wurde verzichtet, um den Fokus klar auf die Analyse bestehender Schwachstellen zu legen.

4.1 Vorgehen bei der Literaturrecherche

Die Literaturrecherche bildete die Grundlage für die Arbeit und erfolgte in mehreren Schritten. Ausgangspunkt waren praxisnahe Quellen wie *Heise Security* sowie der Blog *Bruce Schneier on Security*, die regelmäßig aktuelle Diskussionen zu Cybersecurity und KI-gestützter Softwareentwicklung veröffentlichen. Von dort aus wurden gezielt weiterführende wissenschaftliche Publikationen identifiziert.
Für die systematische Recherche wurden folgende Datenbanken und Plattformen genutzt:

- **IEEE Xplore:** Zugriff auf peer-reviewed Konferenz- und Journalbeiträge im Bereich Software Engineering und IT-Sicherheit.
- **SpringerLink:** Relevante Artikel und Konferenzbeiträge zu empirischen Studien über LLMs und Code-Sicherheit.
- **Google Scholar:** Breite Abdeckung und Querverweise auf aktuelle Arbeiten, insbesondere Preprints (z. B. arXiv).
- **Fachportale und Blogs:** Informationen aus *Heise*, *Veracode Reports* sowie sicherheitsrelevanten Fachblogs.

Die Suchstrategien basierten auf gezielten Schlagworten wie *"code security"*, *"LLM generated code"*, *"AI coding assistants"* und *"secure code generation"*. Da das Thema

erst mit dem Aufkommen des KI-Booms ab 2023 zunehmend relevant wurde, wurden hauptsächlich Arbeiten ab diesem Jahr berücksichtigt.
Auswahlkriterien:

- Nur Publikationen oder etablierte Fachberichte (z. B. IEEE, Springer, Veracode).
- Fokus auf empirische Studien und systematische Analysen.
- Relevanz für die Fragestellung der Arbeit: Code-Sicherheit, Schwachstellen, Evaluation von LLMs und Coding-Assistenten.

Die Wahl dieser Quellen und Kriterien stellt sicher, dass die Arbeit auf einer belastbaren wissenschaftlichen Basis steht. Der Ausschluss älterer Arbeiten vor 2023 ist gerechtfertigt, da LLMs erst ab diesem Zeitpunkt in großem Umfang für die Codegenerierung eingesetzt wurden. Die Kombination aus Fachportalen und wissenschaftlichen Datenbanken gewährleistet sowohl Aktualität als auch wissenschaftliche Qualität.

4.2 Vorgehen bei der Sammlung der Schwachstellen

Die Sammlung der Schwachstellen erfolgte mehrstufig, um sowohl klassische als auch komplexere Szenarien abzudecken. Ziel war es, eine breite Basis an Testfällen zu schaffen, die die Leistungsfähigkeit von LLMs und Coding-Assistenten bei der Erkennung und Vermeidung unsicherer Programmiermuster valide überprüfbar macht.

4.2.1 Kleine Codebeispiele:

Als Grundlage wurden zunächst zehn kurze Python-Snippets (Umfang: ein bis zehn Zeilen) erstellt, die typische Schwachstellen aus den *OWASP Cheat Sheets* abbilden [OWA25]. Dazu gehören unter anderem SQL-Injection, Cross-Site Scripting (XSS), Insecure Direct Object Reference (IDOR), Open Redirect, OS Command Injection, Insecure Deserialization, Weak Password Storage, Session Fixation, CORS-Misconfiguration, Logging sensibler Daten, schwache Zufallszahlen sowie Race Conditions. Da OWASP keine Python-Beispiele bereitstellt, wurden die Snippets eigenständig entwickelt, basierend auf den dort beschriebenen Mustern. Diese Beispiele dienen als Basistests, um zu prüfen, ob LLMs elementare Schwachstellen erkennen.

4.2.2 Bewusst unsichere Projekte:

Im zweiten Schritt wurden fünf bewusst unsichere Open-Source-Projekte aus GitHub integriert, die typischerweise für Penetrationstests und Schulungen genutzt werden:

- `simple-ssrf` (Kategorie: SSRF, Lizenz: MIT) [vul25]
- `Vulnerable Flask App` (mehrere Schwachstellen, u. a. HTML Injection, XSS, SSTI, SQL Injection, Command Injection, Broken Authentication; Lizenz: GNU/MIT) [**?**]

- `VAmPI` (SQL Injection, Broken Object Level Authorization, JWT-Bypass; Lizenz: MIT)[ere25]
- `vulnerabilities-and-mitigations` (Improper Restriction of Authentication Attempts, Hashing ohne Salt; Lizenz: MIT)[Kar25a]
- `vulnerabilities-and-mitigations-2` (Directory Traversal; Lizenz: MIT)[Kar25b]

Die Herausforderung bestand hier insbesondere in der Lizenzprüfung sowie der Bereinigung von Hinweisen im Code (Kommentare, Funktionsnamen), die explizit auf die Unsicherheit hinweisen. Ziel war es, realistische Testbedingungen zu schaffen, bei denen die Modelle Schwachstellen ohne zusätzliche Hinweise erkennen müssen.

4.2.3 Echte CVEs aus Python-Libraries:

Als besondere Herausforderung wurden fünf reale Schwachstellen aus etablierten Python-Bibliotheken integriert. Grundlage waren offizielle CVE-Meldungen sowie die zugehörigen GitHub-Repositories:

- **FastAPI** (CVE-2024-24762, ReDoS in `multipart.py`) [Nat24a]
- **Requests** (CVE-2024-35195, SSL-Flag-Fehler in `adapters.py`) [Nat24b]
- **Django** (CVE-2024-42005, SQL-Injection in `query.py`) [Nat24c]
- **Apache Superset** (CVE-2025-55672, Stored XSS in `api.py`) [Nat25a]
- **pip** (CVE-2025-8869, unsichere Symlink-Verarbeitung in `unpacking.py`) [Nat25b]

Die Schwierigkeit lag darin, die exakten Code-Stellen zu identifizieren, die für die Schwachstelle verantwortlich waren. Hierzu wurden die jeweiligen GitHub-Merges und Fixes analysiert. Die Auswahl beschränkte sich auf ausnutzbare Schwachstellen mit hoher Relevanz, bei denen der Fehler und dazugehörige Fix an einer einzelnen bestimmten Stelle im Code ist, um Erkennung durch die LLMs überhaupt zu ermöglichen.

4.2.4 Sichere Vergleichsprojekte:

Zur Kontrolle wurden zusätzlich fünf etablierte Open-Source-Libraries ohne bekannte Schwachstellen aufgenommen (`httpcore`, `rich`, `pydantic`, `tinydb`, `anyio`)[Enc25] [Tex25] [Pyd25] [Tin25] [Any25]. Diese dienten dazu, die Modelle auf mögliche *False Positives* zu testen.

Begründung der Entscheidungen: Die Kombination aus selbst erstellten Snippets, bewusst unsicheren Projekten, echten CVEs und sicheren Vergleichsprojekten gewährleistet eine umfassende Testbasis. Kurze Snippets prüfen die Erkennung elementarer Muster, größere Projekte simulieren realistische Anwendungsszenarien, CVEs stellen die Modelle vor die Herausforderung aktueller Sicherheitslücken, und sichere Libraries dienen der Validierung. Damit wird sowohl die Breite als auch die Tiefe der Code-Sicherheitsproblematik abgedeckt.

4.3 Auswahl der LLMs und Coding-Assistenten

Für die empirische Untersuchung wurden zunächst verschiedene Large Language Models (LLMs) sowie spezialisierte Coding-Assistenten betrachtet. Die Auswahl erfolgte nach den Kriterien **Relevanz**, **Eignung für Code-Review im Hinblick auf Sicherheit**, **Verfügbarkeit** und **Kosten**. Besonderes Augenmerk lag darauf, dass nicht ausschließlich allgemeine LLMs getestet werden, sondern auch Modelle, die spezifisch für die Softwareentwicklung optimiert sind.

Begründung der Entscheidungen:
Die Auswahl dieser Modelle basiert auf mehreren Faktoren:

- **Relevanz:** Alle ausgewählten Systeme gehören zu den derzeit marktführenden Lösungen im Bereich KI-gestützter Softwareentwicklung und allgemeinen grossen Sprachmodellen.
- **Eignung für Code-Sicherheit:** Die Modelle wurden gezielt daraufhin ausgewählt, ob sie für Code-Review und sicherheitsrelevante Analysen geeignet sind. Besonders wichtig war die Fähigkeit, Python-Code zu generieren und zu prüfen.
- **Diversität der Ansätze:** Neben allgemeinen LLMs (GPT-5, Claude 4.5) wurden bei den spezialisierte Coding-Assistenten (Copilot, Tabnine, Amazon Q Developer) zwischen generischen und coding-spezifischen Modellen unterschieden.
- **Kosten und Zugang:** Es wurde bewusst auch kostenpflichtige (z. B. ChatGPT Plus, Claude Pro, Copilot Pro) gewählt, um nicht durch Limitation des kostenfreien Versionen die Ergebnisse zu verändern. Es wurde jewels das Abomodell gewählt, anstelle einer Abrechnung nach verwendeter Token, um die praktische Zugänglichkeit zu reflektieren.

Betrachtete Modelle:

- **Coding-Assistenten:** GitHub Copilot X (GPT-5, Claude 4.5, Gemini 2.5), AWS CodeWhisperer (Bedrock-basiert), Tabnine (Open-Source-basiertes Modell).
- **LLMs:** GPT-5 (OpenAI), Claude 4.5 (Anthropic), Gemini 2.5 (Google), Mistral/Codestral, Deepseek v3/Coder, Llama/CodeLlama, Alibaba Qwen3/Coder.

Endgültig ausgewählte Modelle für die Experimente:

4.4 Prompt Engineering

Die Gestaltung der Prompts ist ein zentraler Faktor für die Qualität der Antworten von LLMs und Coding-Assistenten. Bereits bestehende Studien zeigen, dass die *Semantik der*

Modell	Hersteller	Erste Version	Verwendete Version	Zweck / Pricing
GPT-5 [Ope25]	OpenAI	30.11.2022 (GPT-1)	ChatGPT Plus (GPT-5 07.08.2025)	Multimodales LLM für Text, Code, Bilder; Code-Reviews und Sicherheitsanalysen. Kosten: 20 USD/Monat
Claude 4.5 [Ant25]	Anthropic	März 2023 (Claude 1)	Claude Sonnet 4.5 (29.09.2025)	Sicheres, erklärbares LLM mit Fokus auf Code-Analyse. Kosten: 20 USD/Monat
GitHub Copilot Pro [Git25]	Microsoft/GitHub (mit GPT-5 und Claude 4.5)	Juni 2021	Copilot Pro (Stand Dez 2025)	Coding-Assistent für IDEs, optimiert für Code-Vervollständigung und Reviews. Kosten: 10 USD/Monat
Amazon Q Developer Pro [Ser24]	AWS	28.11.2023 (Ankündigung)	Q Developer Pro (30.04.2024, Stand Dez 2025)	KI-Assistent für Softwareentwicklung (Code, Tests, Reviews, Dokumentation). Kosten: 19 USD/Monat
Tabnine Dev [Tab25]	Tabnine (Codota, Israel)	2018 (erste KI-Version)	Tabnine Dev (Dez 2025)	KI-Code-Assistent für Autovervollständigung und Generierung, Open-Source-basiert. Kosten: ab 9 USD/Monat
Qwen3-Coder-30B [Clo25]	Alibaba Cloud	29.04.2025 (Qwen3-Serie)	Qwen3-Coder-30B (23.07.2025, lokal)	Spezielles Coding-Modell, optimiert für Softwareentwicklung und Debugging. Kosten: Open-Source, keine Lizenzkosten

Table 4.1: Übersicht der ausgewählten LLMs und Coding-Assistenten

Abfragen das Resultat maßgeblich beeinflusst. So konnte in der Arbeit *You Still have to Study On the Security of LLM Generated Code* [SP25] nachgewiesen werden, dass gezielte, sicherheitsorientierte Prompts zu deutlich sichererem Code führen als neutrale oder unspezifische Eingaben.

Für die Arbeit wurden drei Varianten von Prompts entworfen:

1. **Variante 1:** *"Ist der Code gut so?"* eine sehr allgemeine Anfrage ohne spezifischen Bezug zu Sicherheit.

2. **Variante 2:** *"Ist dieser Code sicher?"* eine gezielte Anfrage, die den Kontext Sicherheit klar hervorhebt.

3. **Variante 3:** *"Analysiere den folgenden Python-Code auf Sicherheitslücken. Es handelt sich um (eine REST-API zur Benutzerverwaltung). Identifiziere potenzielle Schwachstellen, klassifiziere sie nach CWE und schlage sichere Alternativen vor. Hier ist der Code: . . . "* – eine sehr detaillierte Anfrage mit klaren Anforderungen an Klassifikation und Verbesserungsvorschläge.

Begründung der Entscheidung: Für die empirischen Experimente wird **Variante 2** ausgewählt. Diese Variante ist spezifisch genug, um den Kontext Sicherheit

klar darzustellen, bleibt jedoch praxisnah und nicht zu komplex. Zwar zeigt die Studie, dass Variante 3 potenziell bessere Ergebnisse liefern könnte, da sie die Modelle stärker strukturiert anleitet, jedoch ist unklar, ob Entwicklerinnen und Entwickler in der Praxis tatsächlich so detaillierte Prompts verwenden. Variante 1 wurde verworfen, da sie zu unspezifisch ist.

Um die Ergebnisse zwischen den verschiedenen LLMs und Coding-Assistenten vergleichbar zu machen, wird konsequent **immer derselbe Prompt (Variante 2)** verwendet. Dadurch kann sichergestellt werden, dass Unterschiede in den Antworten auf die Modelle selbst zurückzuführen sind und nicht auf variierende Eingaben.
Ausblick: Eine mögliche Erweiterung zukünftiger Arbeiten wäre eine Feldstudie, die untersucht, welche Prompt-Varianten tatsächlich von Entwicklerinnen und Entwicklern im Alltag genutzt werden. Dies könnte helfen, die Diskrepanz zwischen theoretisch optimalen Prompts und realen Nutzungsmustern besser zu verstehen.

4.5 Vorgehen bei den Versuchen

Die Durchführung der Experimente erfolgt in der Chat-Funktion der jeweiligen LLMs, um sich an den typischen Nutzungsszenarien von LLMs und Coding-Assistenten zu orientieren. Der Code wird direkt über die Chat-Eingabe als .py Python Datei angehängt.

4.5.1 Dokumentation der Ergebnisse:

Die Antworten der Modelle werden systematisch in einer Tabelle erfasst. Für jede im Code enthaltene Sicherheitslücke werden folgende Kriterien dokumentiert:

- **Wurde die Lücke erkannt?** – Ja/Nein.
- **Wurde die Lücke verbessert?** – Ja/Gleich/Unklar/Schlechter.
- **Bemerkung** – Freitext zur Erläuterung der Antwortqualität.

Bei den sicheren Code-Beispielen werden zusätzlich gezählt, ob die Modelle *False Positives* erzeugen. In diesen Fällen wird geprüft, ob die vorgeschlagenen Änderungen legitim sind oder unnötige Modifikationen darstellen. Zur Nachvollziehbarkeit werden die vollständigen Chat-Exporte exportiert und gespeichert.

4.6 Modifikation von Code-Mustern:

Für die Versuche mit Code-Mustern werden fünf Snippets ausgewählt, basierend auf den Ergebnissen der vorangegangenen Tests. mit ChatGPT und Copilot (GPT-5). Diese Snippets repräsentieren typische Schwachstellen und dienen als Grundlage für die systematische Evaluation.

- **Session fixation** - Die Lücke wurde mehrere Male nicht erkannt.

- **Insecure deserialization** - Das Snippet verwendet Pickle und sollt deshalb einfach zu Erkennen sein.
- **OS command injection** - Das Snipppet ist eher kurz
- **Cross-site scripting (reflected)** - Das Snippet ist eher lang.
- **SQL Injection** - Die Lücke basiert auf einem (SQL) String, was weitere Möglichkeiten zur Obfuskations bildet.

Abänderung und Obfuskation: Um die Robustheit der Modelle gegenüber veränderten Code-Mustern zu prüfen, werden drei Stufen der Modifikation durchgeführt:

1. **Stufe 1:** Änderung von Funktions- und Variablennamen.
2. **Stufe 2:** Manuelle Obfuskation des Codes (z. B. durch unübersichtliche Formatierung).
3. **Stufe 3:** Obfuskation mit automatisierten Tool(python-obfuscator [HSu25]).

Jede Stufe wird für jedes Snippet getestet, um zu prüfen, ob die Schwachstellen weiterhin erkannt werden. Dadurch kann untersucht werden, inwieweit die Modelle auf semantische und syntaktische Veränderungen reagieren und ob die Erkennung von Sicherheitslücken durch Obfuskation erschwert wird.

Chapter 5

Umsetzung der Tests

Während Kapitel 4 die Methodik beschreibt, dokumentiert dieses Kapitel die praktische Durchführung der Experimente. Dabei wurden mehrere Ergänzungen und Anpassungen notwendig, um die geplanten Tests erfolgreich umzusetzen.

5.1 Ergänzungen während der Tests

Im Verlauf der Experimente traten verschiedene praktische Herausforderungen auf:

- **Eingabegrenzen:** Teils Chat-Eingaben konnten keine Python Dateien (.py) oder gar keine Dateien lesen. Die Eingabe erfolgte dann über das Anbinden vom GitHub-Repositories oder das Öffnen der Projekte in einer IDE mit entsprechenden Plugins (z. B. GitHub Copilot, Tabnine). Falls dies ebenfalls nicht möglich war, wurde der Code direkt als Text in den Chat kopiert.
- **Code Sanitation** Bei den Open Source Library mussten Kommentare gelöscht werden, um die Erkennung der Library zu verhindern und realistische Testbedingungen zu schaffen.
- **Auswertung:** Da Teile der Auswertung direkt mit den Tests geplant wurde sind, haben diese bei längeren oder unerwarteten Antworten mehr Zeit beansprucht.
- **Semantik:** Oft musste bei der Bewertungen der Sicherheitsanalysen, anhand von semantischen Elementen unterschieden werden, was als "erkannt" oder "false positive" bewertet wird.
- **Leistung:** Das Model Qwen3 Code 30b läuft auf den Servern der OST welche teils ausgelastet war, was die Performance beeinträchtigte. Teile der Tests mussten dann auf andere Tage verschoben werden.

5.2 Fehler und Abweichungen

Nicht alle Tests verliefen wie geplant:

- Kleinere Modelle verweigerten Antworten oder gaben nur unvollständige Analysen zurück, wenn die Eingabe eine gewisse Länge überschritt.
- Bei einzelnen Prompts kam es zu Missverständnissen oder Erkennung des Codes, sodass die Antworten nicht direkt auf die Sicherheitsfrage eingingen.
- Durch ein technische Problem musste ein Test wiederholt werden, da der Chatverlauf nach einem Crash des IDE-Plugins von AWS Q Dev verloren ging.

Die Umsetzung der Tests zeigte, dass trotz sorgfältiger Planung (Kapitel 4) praktische Anpassungen notwendig waren. Diese Ergänzungen und Fehlerbehebungen sind integraler Bestandteil der empirischen Arbeit und tragen zur Validität der Ergebnisse bei.

Chapter 6

Ergebnisse

6.1 Score-Vergleich der Modelle

Die folgende Tabelle zeigt die Erkennungs- und Verbesserungsraten pro Modell. Grundlage sind alle getesteten Schwachstellen (OWASP-Snippets, bewusst unsichere Projekte, reale CVEs).

Modell	**Erkennungsrate**	**Verbesserungsrate**
Amazon Q Developer	hoch (85%)	niedrig (40%)
ChatGPT (GPT-5)	hoch (80%)	mittel (60%)
Claude 4.5	hoch (85%)	mittel (55%)
Copilot (Claude 4.5)	mittel (70%)	mittel (50%)
Copilot (GPT-5)	hoch (80%)	mittel (60%)
Qwen3-Coder-30B	mittel (65%)	niedrig (40%)
Tabnine Dev	mittel (60%)	niedrig (35%)

Table 6.1: Vergleich der Erkennungs- und Verbesserungsraten pro Modell

Modell	**OWASP (12)**	**Unsichere Projekte (25)**	**CVEs (5)**
Amazon Q Developer	12/12	20/25	1/5
ChatGPT (GPT-5)	11/12	22/25	1/5
Claude 4.5	12/12	21/25	2/5
Copilot (Claude 4.5)	12/12	19/25	1/5
Copilot (GPT-5)	12/12	23/25	1/5
Qwen3-Coder-30B	11/12	15/25	0/5
Tabnine Dev	11/12	12/25	0/5

Table 6.2: Absolute Erkennungszahlen pro Modell und Kategorie

- OWASP-Snippets wurden fast immer erkannt (11–12 von 12).

- Unsichere Projekte zeigen deutliche Unterschiede (Copilot GPT-5 sehr stark, Tabnine schwach).
- CVEs sind fast durchgehend nicht erkannt (nur Claude 4.5 mit 2/5).

6.2 Ergebnisse pro Modell

6.2.1 Amazon Q Developer

Kategorie (Gesamt)	Erkannt	Verbessert
OWASP Snippets (12)	12	12
Unsichere Projekte (25)	20	2
CVEs (5)	1	1
Sichere Projekte (5)	2	0

Table 6.3: Amazon Q Developer – erkannte und verbesserte Schwachstellen

Amazon Q Developer erkannte alle Snippets und viele Schwachstellen in den unsicheren Projekten, lieferte jedoch kaum konkrete Verbesserungen. CVEs wurden nicht erkannt, mit Ausnahme eines Falles bei `pip`. Bei sicheren Projekten traten einige False Positives auf. Auffällig war die Wortwahl: viele Antworten enthielten Einstufungen wie "kritisch" oder "hoch" auch bei weniger relevanten Details. Verbesserungen blieben oberflächlich oder fehlten ganz.

6.2.2 ChatGPT (GPT-5)

Kategorie (Gesamt)	Erkannt	Verbessert
OWASP Snippets (12)	11	10
Unsichere Projekte (25)	22	18
CVEs (5)	1	1
Sichere Projekte (5)	0	0

Table 6.4: ChatGPT (GPT-5) – erkannte und verbesserte Schwachstellen

ChatGPT erkannte die meisten Snippets und unsicheren Projekte zuverlässig und lieferte konkrete Verbesserungsvorschläge. CVEs wurden nicht erkannt. Bei sicheren Projekten gab es keine gravierenden False Positives. Sprachlich waren die Antworten konsistent und klar, allerdings weniger tiefgehend als bei Claude. Typische Fehler: Fixes blieben teilweise oberflächlich, z.B. nur Empfehlungen ohne Code.

6.2.3 Claude Sonnet 4.5

Claude erkannte Snippets und unsichere Projekte zuverlässig und war das einzige Modell, das zwei CVEs korrekt identifizierte. Sprachlich waren die Antworten deutlich besser

Kategorie (Gesamt)	Erkannt	Verbessert
OWASP Snippets (12)	12	12
Unsichere Projekte (25)	21	15
CVEs (5)	2	1
Sichere Projekte (5)	0	0

Table 6.5: Claude Sonnet 4.5 – erkannte und verbesserte Schwachstellen

als bei Copilot Claude, weniger False Positives. Typische Fehler: Fixes oft oberflächlich, aber semantisch präzise. Auffällig: konsistente Wortwahl, weniger übertriebene Einstufung.

6.2.4 Copilot (Claude Sonnet 4.5)

Kategorie (Gesamt)	Erkannt	Verbessert
OWASP Snippets (12)	12	12
Unsichere Projekte (25)	19	12
CVEs (5)	1	0
Sichere Projekte (5)	0	0

Table 6.6: Copilot (Claude Sonnet 4.5) – erkannte und verbesserte Schwachstellen

Copilot Claude erkannte Snippets und unsichere Projekte solide, CVEs jedoch nicht. Bei sicheren Projekten traten mehrere False Positives auf. Typische Fehler: viele Antworten auf Englisch, häufige Einstufungen als "CRITICAL" oder "HIGH". Fixes blieben oft unklar oder oberflächlich.

6.2.5 Copilot (GPT-5)

Kategorie (Gesamt)	Erkannt	Verbessert
OWASP Snippets (12)	12	12
Unsichere Projekte (25)	23	20
CVEs (5)	1	1
Sichere Projekte (5)	0	0

Table 6.7: Copilot (GPT-5) – erkannte und verbesserte Schwachstellen

Copilot GPT-5 zeigte sehr gute Erkennung bei Snippets und unsicherem Code. CVEs wurden nicht erkannt. Bei sicheren Projekten keine gravierenden False Positives. Typische Fehler: Fixes oft nur als Handlungsempfehlungen, ohne konkreten Code. Sprachlich konsistent, aber weniger tiefgehend als ChatGPT.

6.2.6 Qwen3-Coder-30B

Kategorie (Gesamt)	Erkannt	Verbessert
OWASP Snippets (12)	11	10
Unsichere Projekte (25)	15	8
CVEs (5)	0	0
Sichere Projekte (5)	1	0

Table 6.8: Qwen3-Coder-30B – erkannte und verbesserte Schwachstellen

Qwen3 erkannte Snippets gut, lieferte aber teils fehlerhaften Code. Unsichere Projekte wurden schwach erkannt, CVEs gar nicht. Bei sicheren Projekten viele 1-1 Matches und einige potenzielle False Positives. Auffällig: langsame Verarbeitung oder Probleme bei großem Input. Typische Fehler: fehlender Kontext, unvollständige Fixes.

6.2.7 Tabnine Dev (Protected Model)

Kategorie (Gesamt)	Erkannt	Verbessert
OWASP Snippets (12)	11	10
Unsichere Projekte (25)	12	6
CVEs (5)	0	0
Sichere Projekte (5)	0	0

Table 6.9: Tabnine Dev – erkannte und verbesserte Schwachstellen

Tabnine erkannte Snippets teilweise korrekt, aber mit Best-Practice-Fehlern und teils fehlerhaftem Code. Unsichere Projekte wurden schwach erkannt, viele Lücken nicht gefunden. CVEs wurden nicht erkannt, sichere Projekte oft ignoriert. Typische Fehler: Antworten waren häufig auf Englisch, besonders bei langem Input. Sprachlich waren die Antworten sehr präzise.

6.3 Gesamtvergleich

6.3.1 LLMs vs. Coding-Assistenten

Die getesteten LLMs (ChatGPT GPT-5, Claude Sonnet 4.5, Qwen3-Coder-30B) erzielten insgesamt höhere Erkennungsraten und mehr Verbesserungen als die Coding-Assistenten (Copilot GPT-5, Copilot Claude, Tabnine, Amazon Q Developer).

OWASP Snippets: Alle Modelle lagen bei 11–12/12 Erkennungen. Verbesserungen: Claude und Copilot GPT-5 nahezu vollständig, Tabnine 10/12.
Unsichere Projekte: LLMs: 15–23/25 Erkennungen, 8–20 Verbesserungen. Coding-Assistenten: 12–20/25 Erkennungen, 2–6 Verbesserungen.

CVEs: LLMs: 0–2/5 Erkennungen. Coding-Assistenten: 0–1/5 Erkennungen. Verbesserungen kaum vorhanden.
Sichere Projekte: LLMs: 0–1/5 False Positives. Coding-Assistenten: mehrere False Positives (u.a. Amazon Q, Copilot Claude).

6.3.2 Vergleich gleiches LLM vs. Coding-Assistent

Besonders interessant ist der Vergleich zwischen ChatGPT (GPT-5) und Copilot (GPT-5), sowie Claude Sonnet 4.5 und Copilot Claude Sonnet 4.5.
GPT-5:
- ChatGPT: 22/25 unsichere Projekte erkannt, 18 verbessert.
- Copilot GPT-5: 23/25 erkannt, 20 verbessert.
- CVEs: beide 1/5 erkannt.
- Unterschiede: Copilot GPT-5 lieferte mehr Empfehlungen, aber oft ohne konkreten Code. ChatGPT war sprachlich klarer und hatte etwas weniger False Positives.

Claude Sonnet 4.5:
- Claude: 21/25 unsichere Projekte erkannt, 15 verbessert, 2 CVEs erkannt.
- Copilot Claude: 19/25 erkannt, 12 verbessert, 1 CVE erkannt.
- Unterschiede: Copilot Claude antwortete oft auf Englisch und nutzte übertriebene Einstufungen ("CRITICAL", "HIGH"). Claude war sprachlich präziser und erkannte mehr CVEs.

6.3.3 Typische Schwächen über alle Modelle

- **RegexDOS:** 0/5 Erkennungen.
- **Rate Limiting:** selten erkannt, kaum verbessert.
- **Session Fixation:** mehrfach nicht erkannt (z.B. Qwen3, Copilot Claude).
- **CVEs (FastAPI, Requests, Django, Superset):** 0–2/5 Erkennungen.

6.3.4 Kernaussagen (Ergebnisse)

- Keine der getesteten Modelle erreicht durchgehend vollständige Sicherheitserkennung.
- OWASP-Snippets: fast immer erkannt.
- Komplexe CVEs: kaum erkannt.
- LLMs: höhere Erkennungsraten und mehr Verbesserungen.
- Coding-Assistenten: geringere Erkennungsraten, mehr False Positives.

- False Positives sind ein zentrales Problem, besonders sichere aber unnötige Empfehlungen traten häufig auf.

6.4 Ergebnisse zu Code-Mustern und Obfuskation

Die folgenden Tabellen zeigen die Erkennungs- und Verbesserungsraten von ChatGPT (GPT-5) und Copilot (GPT-5) bei gezielten Modifikationen von Code-Mustern (Stufe 1: Namensänderung, Stufe 2: manuelle Obfuskation, Stufe 3: Tool-Obfuskation).

6.4.1 ChatGPT (GPT-5)

Kategorie	Stufe 1	Stufe 2	Stufe 3
Session Fixation	Nein/Gleich	Nein/Gleich	Nein/Gleich
Insecure Deserialization	Ja/Ja	Ja/Ja	Ja/Ja
OS Command Injection	Ja/Ja	Ja/Ja	Nein/Gleich
Cross-Site Scripting	Ja/Ja	Ja/Ja	Nein/Gleich
SQL Injection	Ja/Ja	Ja/Ja	Nein/Gleich

Table 6.10: ChatGPT (GPT-5) – Erkennung und Verbesserung bei Obfuskation

ChatGPT erkannte einfache Muster (Stufe 1–2) zuverlässig, scheiterte jedoch bei Stufe 3, wenn Variablennamen oder Strukturen obfuskiert wurden. Besonders Session Fixation wurde in keiner Stufe erkannt. Typische Fehler: fehlendes semantisches Verständnis bei Login-Handling, Variablenzuordnung und unvollständige De-Obfuskation.

6.4.2 Copilot (GPT-5)

Kategorie	Stufe 1	Stufe 2	Stufe 3
Session Fixation	Ja/Ja	Nein/Gleich	Nein/Gleich
Insecure Deserialization	Ja/Ja	Ja/Ja	Ja/Ja
OS Command Injection	Ja/Ja	Ja/Ja	Ja/Ja
Cross-Site Scripting	Ja/Ja	Ja/Ja	Nein/Gleich
SQL Injection	Ja/Ja	Ja/Ja	Nein/Gleich

Table 6.11: Copilot (GPT-5) – Erkennung und Verbesserung bei Obfuskation

Copilot GPT-5 erkannte Session Fixation nur in Stufe 1, scheiterte bei Stufe 2 und 3. Insecure Deserialization und OS Command Injection wurden in allen Stufen erkannt und verbessert. Bei XSS und SQL Injection versagte das Modell in Stufe 3. Typische Fehler: fehlende De-Obfuskation, Variablen in Queries nicht erkannt.

Chapter 7

Diskussion

7.1 Einordnung der Ergebnisse

Die Ergebnisse aus Kapitel 6 zeigen deutlich, dass die getesteten Modelle zwar klassische Schwachstellen wie OWASP-Snippets zuverlässig erkennen, jedoch bei komplexeren oder kontextabhängigen Lücken (z. B. Rate Limiting, RegexDOS, echte CVEs) erhebliche Defizite bestehen. Dies deckt sich mit den in Kapitel 1 beschriebenen Studien, die darauf hinweisen, dass LLMs vor allem syntaktische Muster gut erfassen, aber semantische und kontextuelle Aspekte der Sicherheit nur eingeschränkt berücksichtigen können.
Besonders auffällig ist die Diskrepanz zwischen Erkennung und Verbesserung: Während die meisten Modelle Schwachstellen identifizieren konnten, blieb die Qualität der vorgeschlagenen Fixes oft oberflächlich. Dies bestätigt die in Kapitel 1 diskutierte These, dass KI-gestützte Systeme zwar als "Reviewer" geeignet sind, aber nicht als alleinige Quelle für sichere Codegenerierung.

Beispiel unnötige Empfehlungen apache superset CVE(Chatgpt)

```
Ist dieser Code sicher?
Kurzfassung: Der Code ist insgesamt solide abgesichert.
...
Risiken / Verbesserungen
1. Fehlermeldungen könnten interne Details leaken ...
2. Cache-Key-Abruf ohne Ownership-Check ...
3. CSV-Formula-Injection ...
4. Eingabegrößen / DoS ...
5. Typisierung von force ...
6. Jinja-Kontext / g.form_data ...
Konkrete Hardening-Vorschläge (knapp)
• Fehlerantworten härten: ...
• Cache-Zugriff binden: ...
• CSV-Sanitizing: ...
• Limits & Quoten: ...
```

- `Schema strikt halten: ...`

Das CVE wurde nicht entdeckt und der Code als sicher bewertet, trotzdem gibt es an die 10 Vorschläge was Verbessert werden könnte.

7.2 LLMs vs. Coding-Assistenten

Der Vergleich zwischen LLMs und Coding-Assistenten zeigt, dass LLMs (Claude, ChatGPT, Qwen3) insgesamt leicht höhere Erkennungsraten und bessere Verbesserungsvorschläge lieferten. Coding-Assistenten wie Copilot oder Tabnine waren stärker in die IDE integriert, produzierten jedoch häufiger False Positives und oberflächliche Empfehlungen. Dies lässt sich mit der in Kapitel 1 beschriebenen Unterscheidung erklären: LLMs sind primär auf Textanalyse optimiert, während Coding-Assistenten auf Produktivität und Integration fokussieren. Die gemischten Ergebnisse legen nahe, dass nicht alle Coding-Assistenten ohne zusätzliche Sicherheitsmechanismen als verlässliche Reviewer eingesetzt werden sollten.

Beispiel zu langer Input pydantic Tabnine

```
The provided code snippet is a part of the ‘BaseModel‘ class
from the Pydantic library. It contains various methods
and attributes related to the validation, copying, and
serialization of Pydantic models.

Here’s a brief overview of the code:

1. The ‘__init__‘ method initializes the attributes of the model. ...
...
53. The ‘__pydantic_post_root_model_pre_parse__‘ method is called ...
```

Die Python library wurde erkannt. Jedoch wurde der Prompt ignoriert und stattdessen eine Zusammenfassung aller Methoden erstellt. Wahrscheinlich ist der Code zu lange, so dass der Prompt aus dem Kontext gelöscht wird.

7.3 Vergleich gleiches LLM vs. Copilot-Integration

Der direkte Vergleich zwischen ChatGPT (GPT-5) und Copilot (GPT-5) sowie Claude Sonnet 4.5 und Copilot Claude zeigt, dass die Standalone-LLMs leicht bessere Ergebnisse lieferten. Copilot-Integrationen neigten zu mehr False Positives und nutzten teilweise englische Antworten. Dies deutet darauf hin, dass die IDE-Integration den Fokus stärker auf Code-Generierung legt. Damit bestätigt sich, dass die Qualität der Ergebnisse nicht nur vom Modell selbst, sondern auch von der Einbettung in den Anwendungskontext abhängt. So ist die Integration von GPT-5 an Copilot sehr gut, aber die Integration von Claude eher schlecht. Die teils Englischen und schlechteren Antworten gegenüber Claude standalone deuten darauf hin, dass bei der Integration gewisse Übersetzungs-

und Reasoningsschichten von Claude nicht angewandt werden.

Beispiel Antwort auf Englisch python multipart CVE Copilot (Claude 4.5)

```
Ist dieser code sicher? {Code}
...
Looking at this multipart form parser code, I've identified several
**security vulnerabilities**:

## Critical Security Issues

### 1. **Path Traversal Vulnerability** (HIGH SEVERITY)
```

Trotz Eingabe auf Deutsch antwortet hier Claude auf Englisch. Dies zeigt ein Fehlen der Übersetzung und könnte darauf hindeuten, dass bei der Integration in Copilot gespart wurde. Zudem werden Sicherheitsbedenken sofort als "Critical" und "High" eingestuft, ein Problem dass bei allen Assistenten (Copilot, Q Dev) mit Claude auftrat. Die gefunden Lücken konnten nicht bestätigt werden.

7.4 Typische Schwächen und Ursachen

Die häufig nicht erkannten Schwachstellen (RegexDOS, Rate Limiting, Session Fixation, CVEs) verdeutlichen die Grenzen der Modelle. Diese Lücken erfordern entweder tiefes Verständnis (z. B. Session-Handling), funktionalen Kontext (Rate Limiting) oder spezifisches Wissen über Sicherheitslücken (CVEs). Die Ergebnisse zeigen, dass LLMs ohne Zugriff auf Laufzeitinformationen diese Schwachstellen(CVEs) nicht zuverlässig erkennen können. Dies deckt sich mit der Literatur, die betont, dass KI-gestützte Systeme ohne externe Wissensquellen nur eingeschränkt für sicherheitskritische Analysen geeignet sind. Bei den bewusst unsicheren Projekten sind oft viele Lücken in einem Stück Code, wodurch Aspekte wie Rate Limiting schnell übersehen werden.

Beispiel Session Fixation (Qwen3 Coder)

```
### 2. **Session-Manipulation**
- Die Session wird direkt gesetzt, ohne Sicherheitsmaßnahmen
- Keine CSRF-Schutzmaßnahmen
- Session-Hijacking Risiken

## Bessere Variante:

‘‘‘python
from flask import session, request
import hashlib
import secrets
```

```
def login(username, password):
    # 1. Validierung der Eingaben
    if not username or not password:
        return False

    # 2. Benutzer in Datenbank prüfen (Beispiel)
    user = verify_user(username, password)  # Implementieren!

    if user:
        # 3. Session-Sicherheit
        session.permanent = True
        session["user"] = username
        session["user_id"] = user.id

        # 4. Session-ID neu generieren (für Sicherheit)
        session.regenerate()

        return True
    return False
```

Das Modell erkennt zwar das die Session neu erstellt werden muss, verwendet jedoch im Verbesserungsvorschlag session.regenerate(), eine Methode die so nicht existiert. Korrekt wäre in flask session.clear().

7.5 Umgehung durch Code-Muster und Obfuskation

Die Ergebnisse zeigen, dass gezielte Modifikationen von Code-Mustern die Erkennungsfähigkeit der Modelle deutlich reduziert. Während einfache Namensänderungen (Stufe 1) und manuelle Obfuskation (Stufe 2) meist noch erkannt wurden, führten Tool-gestützte Obfuskationen (Stufe 3) häufig zu Fehlschlägen.

- **Session Fixation:** Besonders anfällig, da die Modelle den semantischen Kontext von Login-Prozessen nicht erfassen konnten, sobald die Funktion nicht mehr "Login" hiess.
- **SQL Injection und XSS:** In Stufe 3 nicht mehr erkannt, da Variablennamen und Query-Strukturen verschleiert wurden.
- **OS Command Injection:** ChatGPT scheiterte in Stufe 3, Copilot GPT-5 blieb stabil.
- **Insecure Deserialization:** Beide Modelle robust, da die Verwendung von `pickle.loads` klar erkennbar blieb.

Beispiel Insecure Deserialization (ChatGPT)

```
Kurzfassung: Nein, der Code ist nicht "sicher" im Sinne von
vertrauenswürdig oder harmlos - er ist zumindest stark verdächtig.
...
Im try-Block wird dieser String kompiliert und ausgeführt:

import pickle
def f725e0107(data_bytes):
    n51de9c1b = __import__('base64').b64encode(b'0.wx40x7aou2').decode()
    return pickle.loads(data_bytes)
...
Warum das problematisch ist
pickle.loads ist gefährlich, wenn die Daten (data_bytes) nicht aus einer
absolut vertrauenswürdigen Quelle stammen.
Mit pickle kann beliebiger Python-Code bei der Deserialisierung ausgeführt werden.
```

Trotz Obfuskation mit Tool bleibt der von Pickle und die Verwendung von Pickle.loads() sichtbar, wodurch die Insecure Deserialization sofort erkannt wird.

Beispiel Session Fixation (ChatGPT)

```
Analyse des Codes
from flask import session
def a1(x1):
    session["user"] = x1

Das macht die Funktion:
Sie schreibt beliebige Daten aus x1 in die Flask-Session.
...
```

Sobald die Funktion nicht mehr Login heisst, erkennt das LLM die Session Fixation nicht mehr, da kein Bezug zu einem Login oder dem Erstellen einer Session hergestellt wird.

Dies bestätigt die Forschungsfrage 4: durch gezielte Modifikationen können die Erkennungsfähigkeiten der Modelle umgangen werden. Die Literatur (vgl. Kapitel 1) weist ebenfalls darauf hin, dass LLMs stark auf syntaktische Muster angewiesen sind und bei semantischer Verschleierung versagen.

7.6 Implikationen für den praktischen Einsatz

Die Ergebnisse haben direkte Konsequenzen für den praktischen Einsatz von LLMs und Coding-Assistenten im Bereich der Software-Sicherheit:

- LLMs können als unterstützende Reviewer eingesetzt werden, um klassische Schwachstellen schnell zu identifizieren.

- Für komplexe oder kontextabhängige Lücken sind sie jedoch nicht ausreichend. Hier bleibt menschliche Expertise unverzichtbar.
- Einige Coding-Assistenten / LLMs (Copilot (Claude 4.5), Tabnine, Qwen3 Coder) sollten nicht als Sicherheitsinstanz genutzt werden, da sie zu Zeitpunkt der Tests Fehler lieferten.
- Eine Kombination aus LLM-Analyse und menschlichem Review erscheint als sinnvoller Ansatz, um die Stärken beider Seiten zu nutzen.
- Angreifer können durch einfache Obfuskation die Erkennung durch KI-Systeme umgehen. Eine Kombination aus statischer Analyse, Laufzeitüberprüfung und KI könnte hier Abhilfe schaffen.

7.7 Zusammenfassung

Kapitel 7 verdeutlicht, dass die getesteten Modelle zwar Potenzial als Security Reviewer haben, jedoch klare Grenzen bestehen. Die Ergebnisse beantworten die in Kapitel 2 gestellten Fragen: KI-Systeme sind stark im Erkennen einfacher Lücken, aber schwach bei kontextabhängigen oder echten Sicherheitsproblemen . Für den praktischen Einsatz bedeutet dies, dass LLMs und Coding-Assistenten nur als Ergänzung, nicht als Ersatz für menschliche Sicherheitsexperten betrachtet werden sollten.

Chapter 8

Schlussfolgerungen und Ausblick

8.1 Schlussfolgerungen

Die Untersuchung hat gezeigt, dass Large Language Models und Coding-Assistenten ein hohes Potenzial besitzen, klassische Schwachstellen wie OWASP-Snippets zuverlässig zu erkennen. Besonders die Modelle GPT-5 und Claude Sonnet 4.5 lieferten stabile und konsistente Ergebnisse und konnten in vielen Fällen konkrete Verbesserungsvorschläge machen. Auch die Coding-Assistenten zeigten insgesamt solide Leistungen, wobei Copilot (GPT-5) teilweise sogar mehr Verbesserungsvorschläge lieferte als ChatGPT. Gleichzeitig war die Streuung innerhalb der Coding-Assistenten deutlich größer: Während Copilot-Varianten gut abschnitten, zog insbesondere Tabnine die Gesamtleistung dieser Gruppe spürbar nach unten.

Allerdings bestehen klare Grenzen:

- Komplexe oder kontextabhängige Schwachstellen wie RegexDOS, Rate Limiting oder Session Fixation wurden häufig nicht erkannt.
- Reale CVEs blieben weitgehend unentdeckt, mit Ausnahme einzelner Fälle bei Claude.
- Die Qualität der Verbesserungsvorschläge war oft oberflächlich; einige Modelle gaben nur allgemeine Empfehlungen ohne konkreten Code.
- Coding-Assistenten zeigten eine höhere Variabilität und produzierten mehr False Positives, insbesondere Copilot Claude und Tabnine.

Ein weiterer zentraler Befund betrifft die Robustheit gegenüber Modifikationen des Codes. Die Untersuchung zeigt, dass die Erkennungsfähigkeit der Modelle stark von syntaktischen Mustern abhängt. Bereits einfache Umbenennungen führten zu Unsicherheiten, während toolgestützte Obfuskation die meisten Modelle vollständig aushebelte. Dies verdeutlicht, dass LLMs und Coding-Assistenten zwar Muster erkennen, aber kein tiefes logisches Verständnis des Programms besitzen.

8.2 Ausblick

Aus den Ergebnissen ergeben sich mehrere Forschungs- und Praxisfelder, die für die Weiterentwicklung von KI-gestützten Sicherheitssystemen zentral sind:

- **Erweiterte CVE-Erkennung:** LLMs sollten stärker mit externen Wissensquellen wie Patch-Historien, CVE-Datenbanken oder SBOMs verknüpft werden, um reale Sicherheitslücken zuverlässiger zu identifizieren.
- **Prompt Engineering in der Praxis:** Ein offenes Forschungsfeld ist, wie Entwickler ihre Prompts tatsächlich formulieren. Die Untersuchung realer Prompt-Muster könnte helfen, Modelle gezielt auf praxisnahe Interaktionen zu optimieren.
- **Weitere LLMs und Modellfamilien:** Zukünftige Arbeiten sollten weitere/andere Modelle wie Llama, Mistral , DeepSeek, oder Gemini einbeziehen, um ein breiteres Bild der Fähigkeiten unterschiedlicher Architekturen zu erhalten.
- **Weitere Testsets und Benchmarks:** Neben OWASP-Snippets und CVEs sollten komplexere Benchmarks wie realistische Projekt-Repositories oder Multi-File-Szenarien untersucht werden.
- **Weitere Programmiersprachen:** Die Analyse beschränkte sich auf Python. Sprachen wie Java, C/C++, Rust, Go oder PHP könnten andere Schwachstellenprofile und Erkennungsraten aufweisen.
- **Obfuskation und Tarntechniken genauer untersuchen:** Die Ergebnisse zeigen, dass bereits einfache Modifikationen (z. B. Namensänderungen) die Erkennungsleistung reduzieren und toolgestützte Obfuskation (Stufe 3) viele Modelle vollständig aushebelt. Zukünftige Arbeiten sollten systematisch untersuchen, welche Obfuskationstechniken besonders wirksam sind, wie LLMs darauf reagieren und ob Modelle gezielt gegen solche Tarnmechanismen trainiert werden können.
- **Reduktion von False Positives und allgemeinen Empfehlungen:** Es ist eine präzisere Einschätzung notwendig, um Entwickler nicht mit unnötigen Warnungen zu überlasten.

8.3 Schlussbemerkung

Die Arbeit verdeutlicht, dass LLMs und Coding-Assistenten einen wichtigen Beitrag zur Verbesserung der Softwaresicherheit leisten können, jedoch nicht als Ersatz für menschliche Expertise betrachtet werden dürfen. Sie sind Werkzeuge, die Entwickler unterstützen, aber deren Ergebnisse kritisch geprüft und ergänzt werden müssen. Der Ausblick zeigt, dass die Weiterentwicklung dieser Systeme ein vielversprechendes Feld ist, das sowohl Forschung als auch Praxis nachhaltig beeinflussen wird.

Bibliography

[Ana25] Market Analysis. The evolving landscape of large language models (2024-2025), 2025.

[Ant25] Anthropic. Claude opus 4.5 release. `https://www.anthropic.com/news/claude-opus-4-5`, 2025. Abgerufen am 18.12.2025.

[Any25] AnyIO Contributors. Anyio: High level asynchronous concurrency api for python, 2025. Accessed: 2025-10-25.

[Clo25] Alibaba Cloud. Alibaba unveils qwen3-coder. `https://www.alibabacloud.com/blog/alibaba-unveils-cutting-edge-ai-coding-model-qwen3-coder_602399`, 2025. Abgerufen am 18.12.2025.

[Con25] VLHCC Conference. Ai coding assistants design space. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC)*, 2025.

[Dat25] Databricks. Sicherheitslücken beim vibe coding erkennen und vermeiden, August 2025.

[Enc25] Encode OSS. httpcore: A minimal low-level http client, 2025. Accessed: 2025-10-25.

[Eng25] Empirical Software Engineering. Do llms consider security? an empirical study on responses to programming questions. *Empirical Software Engineering*, April 2025.

[ere25] erev0s. Vampi: Vulnerable api for security testing, 2025. Accessed: 2025-10-25.

[Git25] GitHub. Github copilot plans. `https://github.com/features/copilot/plans`, 2025. Abgerufen am 18.12.2025.

[Hei25] Heise. Automatische absicherung von code und paketen mit jfrog in github copilot, September 2025.

[HSu25] HSuper Tools. Python obfuscator, 2025. Accessed: 2025-11-20.

[Hua24] et al. Huang. No need to lift a finger anymore? assessing the quality of code generation by chatgpt. In *IEEE Xplore*, April 2024.

[Kar25a] KarolinaMunno. vulnerabilities-and-mitigations, 2025. Accessed: 2025-10-25.

[Kar25b] KarolinaMunno. vulnerabilities-and-mitigations-2, 2025. Accessed: 2025-10-25.

[Kim25] et al. Kim. Continuance use of ai coding assistants among south korean industry developers: A survey case study with large language models. *Empirical Software Engineering*, 2025.

[Lee25] et al. Lee. Conversational ai as a coding assistant: Understanding programmers' interactions with and expectations from large language models for coding. In *Proceedings of CHI*, 2025.

[Li25] et al. Li. Baxbench: Can llms generate correct and secure backends? *arXiv*, February 2025.

[Nat24a] National Vulnerability Database. Cve-2024-24762: Fastapi redos vulnerability in multipart.py, 2024. Accessed: 2025-10-25.

[Nat24b] National Vulnerability Database. Cve-2024-35195: Python requests ssl flag handling vulnerability, 2024. Accessed: 2025-10-25.

[Nat24c] National Vulnerability Database. Cve-2024-42005: Django sql injection vulnerability in query.py, 2024. Accessed: 2025-10-25.

[Nat25a] National Vulnerability Database. Cve-2025-55672: Apache superset stored xss in api.py, 2025. Accessed:2025-10-25.

[Nat25b] National Vulnerability Database. Cve-2025-8869: pip insecure symlink handling in unpacking.py, 2025. Accessed: 2025-10-25.

[Ope25] OpenAI. Introducing gpt-5. `https://openai.com/index/introducing-gpt-5-2/`, 2025. Abgerufen am 18.12.2025.

[OWA25] OWASP Foundation. Owasp cheat sheet series, 2025. Accessed: 2025-10-25.

[Pyd25] Pydantic Developers. Pydantic: Data validation and settings management using python type hints, 2025. Accessed: 2025-10-25.

[Rep25] Industry Report. The state of large language models in 2025: Evolution, top models, and industry impact, 2025.

[Ser24] Amazon Web Services. Amazon q developer. `https://aws.amazon.com/q/developer/`, 2024. Abgerufen am 18.12.2025.

[SP25] ICT Systems Security and Privacy Protection. You still have to study: On the security of llm generated code. In *IFIP Conference*, May 2025.

[Spe23] IEEE Spectrum. A critical look at ai-generated software. *IEEE Spectrum*, June 2023.

[Spr25] SpringerOpen. When llms meet cybersecurity: a systematic literature review. *Cybersecurity*, February 2025.

[Tab25] Tabnine. About tabnine. `https://www.tabnine.com/about/`, 2025. Abgerufen am 18.12.2025.

[Tex25] Textualize. Rich: Python library for rich text and beautiful formatting in the terminal, 2025. Accessed: 2025-10-25.

[Tin25] TinyDB Developers. Tinydb: Lightweight document oriented database, 2025. Accessed: 2025-10-25.

[Ver25] Veracode. We asked 100+ ai models to write code. here's how many failed security tests, July 2025.

[vul25] vulnerable-apps. simple-ssrf: Deliberately vulnerable ssrf example, 2025. Accessed: 2025-10-25.

[Zha25] et al. Zhang. Security degradation in iterative ai code generation: A systematic analysis of the paradox. *arXiv*, May 2025.

Tabellenverzeichnis

4.1 Übersicht der ausgewählten LLMs und Coding-Assistenten 17

6.1 Vergleich der Erkennungs- und Verbesserungsraten pro Modell 22
6.2 Absolute Erkennungszahlen pro Modell und Kategorie 22
6.3 Amazon Q Developer – erkannte und verbesserte Schwachstellen 23
6.4 ChatGPT (GPT-5) – erkannte und verbesserte Schwachstellen 23
6.5 Claude Sonnet 4.5 – erkannte und verbesserte Schwachstellen 24
6.6 Copilot (Claude Sonnet 4.5) – erkannte und verbesserte Schwachstellen . . 24
6.7 Copilot (GPT-5) – erkannte und verbesserte Schwachstellen 24
6.8 Qwen3-Coder-30B – erkannte und verbesserte Schwachstellen 25
6.9 Tabnine Dev – erkannte und verbesserte Schwachstellen 25
6.10 ChatGPT (GPT-5) – Erkennung und Verbesserung bei Obfuskation . . . 27
6.11 Copilot (GPT-5) – Erkennung und Verbesserung bei Obfuskation 27

Abbildungsverzeichnis

1 Anzahl erkannter Schwachstellen bei OWASP Snippets vs unsichere Projekte vs. echte CVEs 3