

Analyse der Fähigkeiten von KI Assistenten zur Erkennung und Behebung von Speicherverwaltungsfehlern in systemnahen Programmiersprachen (C, C++)

Bachelorarbeit im Studiengang Software Engineering und vernetzte Systeme

Author:
Supervisor:
Date:

Pensuni Gino
DI Knipp, Franz
22. Mai 2025, Eisenstadt

Danksagung

An dieser Stelle möchte ich all jenen meinen herzlichen Dank aussprechen, die mich auf meinem Weg zum Bachelorabschluss unterstützt und begleitet haben.

Mein besonderer Dank gilt meinem Betreuer *DI Franz Knipp* für seine wertvolle Unterstützung sowie die Möglichkeit, meine Bachelorarbeit unter seiner Anleitung verfassen zu dürfen. Die zahlreichen Gespräche mit ihm haben wesentlich zur Themenfindung und zur inhaltlichen Ausarbeitung beigetragen. Ich schätzte es sehr, dass ich mich jederzeit mit Fragen an ihn wenden konnte und er stets bereit war, mir mit Geduld und Fachkenntnis zur Seite zu stehen.

Ein ebenso großer Dank gebührt meiner Verlobten *Klaudia Vass*, die mich während der gesamten Zeit mit viel Verständnis und Rückhalt begleitet hat. Sie war mir in herausfordernden Momenten eine unverzichtbare Stütze.

Auch meiner Familie, insbesondere meinem Bruder, möchte ich danken. Durch seine aufmunternden Worte hat er mir immer wieder neuen Mut gegeben und mich darin bestärkt, nicht aufzugeben.

Nicht zuletzt danke ich meinen Arbeitskollegen *Stefan Steindl*, *Paul Schandl*, *Kevin Perry*, *Robert Trausmuth* und *Alexander Nöhner*, die mir während der Ausarbeitung als engagierte Sparringpartner zur Seite standen. Ihre fachliche Unterstützung und ihre Erfahrung haben mir sehr geholfen, meine Arbeit weiterzuentwickeln.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Problembeschreibung	1
1.2	Zielsetzung der Arbeit	2
2	Grundlagen	3
2.1	Allgemeine Definitionen	3
2.1.1	Speicherverwaltung in Systemnahen Programmiersprachen	3
2.1.2	Sicherheitsrelevante Speicherfehler	4
2.1.2.1	Buffer Overflow	4
2.1.2.2	Dangling Pointer und Use-after-Free	5
2.1.2.3	Memory Leak	6
2.1.3	Common Weakness Enumeration	7
2.1.4	Traditionelle Codeanalyse Methoden	8
2.1.4.1	Statische Codeanalyse	8
2.1.4.2	Dynamische Codeanalyse	10
2.1.5	Large Language Models	13
2.1.6	Software Assurance Reference Dataset	14
2.1.6.1	Über Software Assurance Reference Dataset (SARD)	14
2.1.6.2	Aufbau der Testfälle	14
2.2	Stand des Wissens	16
2.2.1	AI Chatbots als Ersatz zu statischen Codeanalyse-Tools	16
2.2.2	Intelligent Code Analysis Agents (ICAA) als neue Form der statischen Codeanalyse	17
2.2.3	Wiederverwendbare Prompt-Patterns	18
2.2.4	Verständnis von Code-Syntax und -Semantik durch Large Language Models	19
3	Vorgangsweise und Methoden	21
3.1	Vorgehensweise	21
3.2	Methoden	21
4	Empirisches Vorgehen	23
4.1	Übersicht über Tools und Plattformen	23
4.2	Vorgangsweise des Experiments	24
4.3	Auswahl der Testfälle	26
4.4	Definition des Prompts für KI	29
4.5	Bewertungsschema für die Erkennung und Behebung von Speicherfehlern	31
4.6	Implementierung der Testfälle	32
4.6.1	Aufbau des eigenen Testframeworks	32
4.6.1.1	Struktur	32
4.6.1.2	Ablaufsteuerung und Ausführung der Testfälle	33
4.6.1.3	Testausgabe	37
4.6.2	Umsetzung der Testfälle in C++	37
4.6.2.1	Code Repository	37
4.6.2.2	Buffer Overflow	38
4.6.2.3	Use-after-free / Dangling Pointer	40

4.6.2.4	Memory Leak	41
4.7	Durchführung des Experiments	43
4.7.1	Prompten der KI-Tools	43
4.7.2	Durchführung der statischen Code Analyse	43
4.7.3	Durchführung der dynamischen Code Analyse	45
4.7.4	Probleme während der Durchführung	48
4.7.4.1	Probleme beim Prompten der KI-Modelle und der Auswertung	48
4.7.4.2	Probleme bei der statischen Code Analyse und Dynamischen Code Analyse	49
4.7.4.3	Probleme mit dem Bewertungsschema	49
5	Ergebnisse und Schlussfolgerungen	51
5.1	Darstellung der Ergebnisse	51
5.1.1	Allgemein	51
5.1.1.1	Testcase TC1_B	51
5.1.1.2	Testcase TC2_B	52
5.1.1.3	Testcase TC3_B	53
5.1.1.4	Testcase TC4_B	53
5.1.1.5	Testcase TC5_B	54
5.1.1.6	Testcase TC6_B	54
5.1.1.7	Testcase TC7_B	55
5.1.1.8	Testcase TC8_B	56
5.1.1.9	Testcase TC9_B	56
5.1.1.10	Testcase TC10_B	57
5.1.1.11	Testcase TC11_B	57
5.1.1.12	Testcase TC12_B	58
5.1.1.13	Testcase TC13_B	58
5.1.1.14	Testcase TC14_B	59
5.1.1.15	Testcase TC15_B	59
5.2	Interpretation und Diskussion	60
5.2.1	Vergleich der Werkzeuge anhand der Ergebnisse	60
5.2.1.1	KI-gestützte Werkzeuge	60
5.2.1.2	Statische und dynamische Analysetools	60
5.2.1.3	Benutzerfreundlichkeit und Aufwand	61
5.2.1.4	Zwischenfazit	61
5.2.2	Stärken und Schwächen der Werkzeuge	61
5.2.3	Reflexion des Bewertungsschemas	63
5.3	Beantwortung der Forschungsfrage	65
5.4	Schlussfolgerungen	66
6	Zusammenfassung	67
	Literaturverzeichnis	69
	Abbildungsverzeichnis	72
	Tabellenverzeichnis	73

A	KI-Prompts und Antworten	75
A.1	Vorwort zum Appendix	75
A.2	Testfall <i>TC1_B</i>	76
A.2.1	Prompt	76
A.2.2	GPT-4o Antwort TC1_B	78
A.2.3	Sonnet 3.7 Antwort TC1_B	80
A.2.4	Gemini 2.5 Pro Antwort TC1_B	82
A.3	Testfall <i>TC2_B</i>	83
A.3.1	Prompt	83
A.3.2	GPT-4o Antwort TC2_B	85
A.3.2.1	Reprompt	85
A.3.2.2	GPT-4o Antwort TC2_B Reprompt	85
A.3.3	Sonnet 3.7 Antwort TC2_B	87
A.3.4	Gemini 2.5 Pro Antwort TC2_B	89
A.4	Testfall <i>TC3_B</i>	91
A.4.1	Prompt	91
A.4.2	GPT-4o Antwort TC3_B	93
A.4.3	Sonnet 3.7 Antwort TC3_B	93
A.4.4	Gemini 2.5 Pro Antwort TC3_B	94
A.5	Testfall <i>TC4_B</i>	96
A.5.1	Prompt	96
A.5.2	GPT-4o Antwort TC4_B	98
A.5.3	Sonnet 3.7 Antwort TC4_B	98
A.5.4	Gemini 2.5 Pro Antwort TC4_B	100
A.6	Testfall <i>TC5_B</i>	103
A.6.1	Prompt	103
A.6.2	GPT-4o Antwort TC5_B	105
A.6.3	Sonnet 3.7 Antwort TC5_B	105
A.6.4	Gemini 2.5 Pro Antwort TC5_B	108
A.7	Testfall <i>TC6_B</i>	111
A.7.1	Prompt	111
A.7.2	GPT-4o Antwort TC6_B	113
A.7.3	Sonnet 3.7 Antwort TC6_B	113
A.7.4	Gemini 2.5 Pro Antwort TC6_B	115
A.8	Testfall <i>TC7_B</i>	116
A.8.1	Prompt	116
A.8.2	GPT-4o Antwort TC7_B	118
A.8.3	Sonnet 3.7 Antwort TC7_B	118
A.8.4	Gemini 2.5 Pro Antwort TC7_B	120
A.9	Testfall <i>TC8_B</i>	123
A.9.1	Prompt	123
A.9.2	GPT-4o Antwort TC8_B	125
A.9.3	Sonnet 3.7 Antwort TC8_B	125
A.9.4	Gemini 2.5 Pro Antwort TC8_B	128
A.10	Testfall <i>TC9_B</i>	131
A.10.1	Prompt	131
A.10.2	GPT-4o Antwort TC9_B	133
A.10.3	Sonnet 3.7 Antwort TC9_B	134

A.10.4 Gemini 3.7 Antwort TC9_B	136
A.11 Testfall <i>TC10_B</i>	139
A.11.1 Prompt	139
A.11.2 GPT-4o Antwort TC10_B	141
A.11.3 Sonnet 3.7 Antwort TC10_B	142
A.11.4 Gemini 3.7 Antwort TC10_B	144
A.12 Testfall <i>TC11_B</i>	147
A.12.1 Prompt	147
A.12.2 GPT-4o Antwort TC11_B	149
A.12.3 Sonnet 3.7 Antwort TC11_B	149
A.12.4 Gemini 3.7 Antwort TC11_B	152
A.13 Testfall <i>TC12_B</i>	156
A.13.1 Prompt	156
A.13.2 GPT-4o Antwort TC12_B	158
A.13.3 Sonnet 3.7 Antwort TC12_B	159
A.13.4 Gemini 3.7 Antwort TC12_B	162
A.14 Testfall <i>TC13_B</i>	164
A.14.1 Prompt	164
A.14.2 GPT-4o Antwort TC13_B	166
A.14.3 Sonnet 3.7 Antwort TC13_B	166
A.14.4 Gemini 3.7 Antwort TC13_B	169
A.15 Testfall <i>TC14_B</i>	172
A.15.1 Prompt	172
A.15.2 GPT-4o Antwort TC14_B	174
A.15.3 Sonnet 3.7 Antwort TC14_B	175
A.15.4 Gemini 3.7 Antwort TC14_B	177
A.16 Testfall <i>TC15_B</i>	181
A.16.1 Prompt	181
A.16.2 GPT-4o Antwort TC15_B	184
A.16.3 Sonnet 3.7 Antwort TC15_B	186
A.16.4 Gemini 3.7 Antwort TC15_B	190
B Clang Output	194
B.1 Testcase TC1_B	194
B.2 Testcase TC2_B	194
B.3 Testcase TC3_B	195
B.4 Testcase TC4_B	196
B.5 Testcase TC5_B	196
B.6 Testcase TC6_B	198
B.7 Testcase TC7_B	199
B.8 Testcase TC8_B	199
B.9 Testcase TC9_B	202
B.10 Testcase TC10_B	202
B.11 Testcase TC11_B	204
B.12 Testcase TC12_B	205
B.13 Testcase TC13_B	208
B.14 Testcase TC14_B	212
B.15 Testcase TC15_B	213

1 Einleitung

1.1 Problembeschreibung

Die Programmiersprachen C und C++ eignen sich gut für die Entwicklung systemnaher Software und bilden die Grundlage zahlreicher kritischer Systeme unserer digitalen Infrastruktur. Diese Sprachen sind nahezu allgegenwärtig und finden sich in Computern, Smartphones, Fahrzeugen und sogar Kameras wieder. Millionen Entwickler weltweit setzen C und C++ in praktischen Anwendungsbereichen ein. Dabei kommt ihnen insbesondere die Fähigkeit dieser Sprachen zugute, direkt auf Hardwareebene zu operieren und Ressourcen wie Speicher manuell zu verwalten. Gerade deshalb eignen sich C und C++ besonders für Systemprogrammierung, Betriebssysteme, sowie performancekritische Anwendungen. Viele bekannte Betriebssysteme wie Windows, Linux oder Apple OS enthalten entscheidende Komponenten, die in C++ entwickelt wurden (Stroustrup, 2013, S. 30).

Diese Programmiersprachen bieten Entwicklern hohe Kontrolle über die Speicherverwaltung und verzichten auf automatische *Garbage Collection*. Daher obliegt es den Entwicklern, zum Beispiel dynamisch allokierte Objekte manuell zu deallokieren. Wird der Speicher nicht deallokiert, sind Speicherlecks die Folge, welche häufig zu Bugs führen (Murali et al., 2024, S. 1).

Neben Speicherlecks zählen auch *Buffer Overflows* oder *Use After Free* zu den schwerwiegenden Problemen solcher Programmiersprachen (Serebryany et al., 2024, S. 1).

Solche Fehler können nicht nur zu Instabilitäten führen, sondern auch gezielt für Angriffe ausgenutzt werden. Räumliche Fehler (z.B. Buffer Overflows durch Zugriff außerhalb der Speichergrenzen) als auch zeitliche Fehler (z.B. Use-after-Free durch Zugriffe auf bereits freigegebenen Speicher) ermöglichen Angreifern, den Programmfluss umzuleiten (Control-Flow Hijacking), sensible Daten zu manipulieren (Data-only Attacks) oder Informationen aus dem Speicher auszulesen (Information Leaks). Da Speicher- und Kontrollsicherheitsmechanismen in C/C++ fehlen, sind solche Schwachstellen weit verbreitet und weiterhin ein zentrales Sicherheitsproblem. Trotz jahrzehntelanger Forschung und zahlreicher Gegenmaßnahmen stellen Speicherfehler bis heute eine ernsthafte Bedrohung dar, da viele bestehende Schutzmechanismen in der Praxis umgangen werden können (Szekeres et al., 2013, S. 48–52).

Neuere Programmiersprachen wie *Rust* hingegen bieten eine moderne Alternative zu C und C++ und versuchen, die Probleme der Speicherverwaltung durch die Fähigkeit Speicherfehler zu vermeiden oder zu erkennen, zu lösen. Rust verwendet ein auf Ownership basiertes System, um Speicherzuweisungen und -freigaben bereits zur Compile-Zeit zu bestimmen (Bugden & Alahmar, 2022, S. 1–2). Um Entwicklern die Möglichkeit zu geben, auf low level Operationen zuzugreifen, erlaubt Rust jedoch auch die Verwendung von *unsafe Code* (Qin et al., 2020, S. 1).

Statische Codeanalyse untersucht den Quellcode eines Programms ohne dessen Ausführung, um potenzielle Fehler und Sicherheitslücken frühzeitig zu identifizieren (Fatima et al., 2018, S. 1). Ein wesentliches Problem dieser Methode ist die hohe Rate an falsch-positiven Ergebnissen, sowie die Art wie die angezeigten Warnung den Entwicklern präsentiert werden, was eine Hürde für die Entwickler darstellt (Johnson et al., 2013, S. 1).

Dynamische Codeanalyse Tools hingegen überprüfen das Verhalten eines Programms während seiner Ausführung auf der Ebene des Maschinencodes und werden häufig verwendet, um die Qualität von Software durch Fehlerprüfungen und Performance-Profiling zu verbessern. Schwergewichtige Tools wie solche, die mit Shadow Values arbeiten, ver-

ursachen typischerweise hohe Laufzeitverlangsamungen von dem Faktor 10 bis 100 oder mehr (Nethercote & Seward, 2007, S. 1–2).

Speicherfehler zu beheben ist somit eine anspruchsvolle Aufgabe, die viel Zeit, Erfahrung und Fachwissen erfordert (Lee et al., 2018, S. 1). An diesem Punkt kommen moderne KI-gestützte Entwicklungswerkzeuge ins Spiel, die versprechen, Entwickler nicht nur bei der Fehlersuche, sondern auch aktiv bei der Fehlerbehebung zu unterstützen. Insbesondere generative KI, also KI-Modelle, die in der Lage sind, selbstständig Code zu analysieren und zu erzeugen, wird zunehmend als zentrales Automatisierungswerkzeug im Software Engineering betrachtet. Sie ermöglicht eine deutlich schnellere und effizientere Entwicklung durch automatisierte Code-Optimierung, Fehlererkennung und die Generierung von Lösungsvorschlägen (Krishna et al., 2024, S. 1).

1.2 Zielsetzung der Arbeit

Ziel dieser Arbeit ist es, das Potenzial moderner KI-gestützter Entwicklungswerkzeuge zur Erkennung und Behebung von Speicherverwaltungsfehlern in systemnahen Programmiersprachen zu untersuchen. Im Fokus stehen dabei die Sprachen *C*, *C++* und *Rust*, die Entwicklern eine manuelle beziehungsweise kontrollierte Speicherverwaltung ermöglichen und dadurch anfällig für schwerwiegende Fehler wie *Buffer Overflows*, *Use-after-Free* oder *Memory Leaks* sind.

Im Zentrum stehen dabei folgende Forschungsfragen:

**Wie gut können KI-Assistenten mit Sprachen umgehen, bei denen der Speicher explizit durch den Quellcode verwaltet wird (C, C++)?
Können diese Assistenten Speicherfehler wie Buffer Overflows zuverlässig erkennen und beheben?**

Die Ergebnisse dieser Arbeit richten sich insbesondere an Entwicklerinnen und Entwickler sicherheitskritischer Software, an Forschende im Bereich Softwareanalyse sowie an Unternehmen, die KI-basierte Werkzeuge zur Verbesserung der Softwarequalität und -sicherheit evaluieren oder einsetzen möchten.

2 Grundlagen

2.1 Allgemeine Definitionen

2.1.1 Speicherverwaltung in Systemnahen Programmiersprachen

Wie ein Programmierer Speicher verwaltet, hängt stark von der verwendeten Programmiersprache ab. Viele modernen Programmiersprachen bieten eine automatische Speicherverwaltung an. Dabei übernimmt ein sogenannter *Garbage Collector* (GC) die Verantwortung für die Verwaltung des Speichers, indem dieser zur Laufzeit automatisch Objekte erkennt, die nicht länger benötigt werden, und deren Speicher wieder freigibt. Diese automatisierte Speicherverwaltung reduziert den Arbeitsaufwand für Programmierer und erhöht oft die Zuverlässigkeit der Software. Jedoch kann das automatische Speichermanagement durch Garbage Collection aufgrund zusätzlicher Laufzeitanalysen zu erhöhter Programmausführungszeit führen, was in zeitkritischen oder systemnahen Anwendungen, wie beispielsweise Betriebssystemen oder Echtzeitsystemen, problematisch sein kann (Davis, 2015, S. 4–6).

Im Gegensatz dazu verlangen Programmiersprachen wie C oder C++ von den Programmierern explizit die manuelle Anforderung und Freigabe des Speichers. Während dies einerseits die Flexibilität und Leistung bietet, erhöht es andererseits deutlich die Komplexität und Fehleranfälligkeit. Häufige Fehler sind hierbei *Memory Leaks*, bei denen Speicher, der nicht mehr benötigt wird, nicht freigegeben wird, was langfristig zu Speichererschöpfung führen kann. Einer der Gründe für die Schwierigkeit in der manuellen Speicherverwaltung liegt in der Herausforderung, die Lebensdauer eines allozierten Speicherblocks korrekt zu bestimmen. Oftmals wird Speicher in einer Funktion alloziert und in einer anderen Funktion wieder freigegeben, was zu einer Diskrepanz zwischen Allokations- und Freigabestellen führt. Diese Situation erfordert *global reasoning*, da Programmierer stets den gesamten Kontext des Programms berücksichtigen müssen. Im Gegensatz dazu benötigt lokale Speicherverwaltung *local reasoning*, welche meist auf dem Stack implementiert ist, keine solche globale Betrachtungsweise, da die Lebensdauer solcher Speicherbereiche streng auf den Funktionsumfang begrenzt ist (Davis, 2015, S. 4–6).

Im Kontext der Programmiersprachen C und C++ lassen sich Stack und Heap klar unterscheiden: Der *Stack* verwaltet Speicher nach dem LIFO-Prinzip und speichert lokale Variablen sowie Parameter von Funktionen, deren Lebensdauer auf den Funktionsumfang begrenzt ist. Der Compiler kümmert sich hierbei automatisch um die Zuweisung und Freigabe des Speichers. Der *Heap* hingegen erlaubt dynamische Speicherallokationen, die durch die Programmierer explizit mit Funktionen wie `malloc()`, `calloc()` oder `realloc()` in C oder durch die Operatoren `new` und `delete` in C++ verwaltet werden. Die manuelle Freigabe dieser dynamischen Speicherbereiche erfolgt ebenfalls explizit mittels der Funktion `free()` in C bzw. den Operatoren `delete` oder `delete[]` in C++ (Will, 2022, S. 489–492) (Wolf, 2014, Kapitel 14).

Listing 1 zeigt ein einfaches Beispiel in C, das die manuelle Speicherverwaltung demonstriert. Hierbei wird ein Integer-Wert auf dem Heap alloziert, der anschließend mit dem Wert 42 initialisiert wird. Nach der Verwendung des Wertes wird der Speicher wieder freigegeben.

```
1  int main() {  
2      int *ptr = (int *)malloc(sizeof(int)); // Allocate memory  
3      if (ptr == NULL) {  
4          // Handle error  
5          return 1;  
6      }  
7      *ptr = 42; // Assign value  
8      printf("%d\n", *ptr); // Print value  
9      free(ptr); // Release memory  
10     return 0;  
11 }
```

Listing 1: Speicherallokation in C, basierend auf (Bryant und O'Hallaron, 2019, S. 7)

2.1.2 Sicherheitsrelevante Speicherfehler

2.1.2.1 Buffer Overflow

Ein Buffer Overflow ist (unter anderen) ein sicherheitsrelevanter Fehler, der in systemnahen Programmiersprachen wie *C* und *C++* auftreten. Diese Sprachen bieten eine sehr direkte und effiziente Speicherverwaltung, überprüfen aber keine Speichergrenzen automatisch. Das bedeutet, dass beim Schreiben von Daten in ein Array oder einen Buffer keine Warnung erfolgt, wenn mehr Daten geschrieben werden, als vorgesehen ist (Tanenbaum & Bos, 2015, S. 640–642).

Ein typisches Beispiel ist die Funktion `gets()` aus der C-Standardbibliothek, die eine Eingabezeile vom Benutzer in einen statisch reservierten Speicherbereich liest, ohne die Länge der Eingabe zu überprüfen. Wenn ein Benutzer mehr Zeichen eingibt als der Buffer fassen kann, werden benachbarte Speicherbereiche überschrieben - unter anderem möglicherweise auch die Rücksprungadresse einer Funktion, die auf dem Stack gespeichert ist (Tanenbaum & Bos, 2015, S. 640–642).

Ein Angreifer kann diese Schwachstelle gezielt ausnutzen, indem er eine Eingabe konstruiert, die nicht nur den Buffer überfüllt, sondern auch die Rücksprungadresse manipuliert. Wird diese Adresse so verändert, dass sie auf den überlaufenen Buffer zeigt, kann der darin enthaltene Schadcode (sogenannter Shellcode) ausgeführt werden - und zwar mit den Rechten des ursprünglichen Programms. Dadurch kann der Angreifer die Kontrolle über das System übernehmen (Tanenbaum & Bos, 2015, S. 640–642).

Um die Erfolgchancen solcher Angriffe zu erhöhen, verwenden Angreifer oft Techniken wie NOP-Sleds (eine Folge von harmlosen Befehlen, die zur eigentlichen Schadcode-Anweisung führen) oder Heap Spraying, bei dem Speicherbereiche gezielt mit ausführbarem Code gefüllt werden, um die Trefferwahrscheinlichkeit beim Umleiten des Programmflusses zu steigern (Tanenbaum & Bos, 2015, S. 640–642).

Folgender selbstentwickelter Beispielcode zeigt, wie ein Buffer Overflow in *C* auftreten kann. Hierbei wird ein statisch reservierter Buffer verwendet, der nicht ausreichend groß ist, um die Eingabe zu speichern. Dies führt zu einem Überlauf und möglicherweise zur Ausführung von Schadcode.

```
1  #include <stdio.h>
2
3  void secret() {
4      printf("successfully executed secret function.\n");
5  }
6
7  void vulnerable() {
8      char buffer[64]; // small buffer allocated on the stack
9
10     printf("Enter your text: ");
11     gets(buffer);      // unsafe function, no length check
12
13     printf("You entered: %s\n", buffer);
14 }
15
16 int main() {
17     vulnerable();
18     return 0;
19 }
```

Listing 2: Beispiel für Buffer Overflow in C (Tanenbaum und Bos, 2015, S. 640)

Die Funktion `gets()`, welche zur Demonstration in diesem Code Beispiel verwendet wird und zur Veranschaulichung genüge tut, ist in modernen C-Standards als unsicher deklariert und sollte nicht mehr verwendet werden. Stattdessen sollten sicherere Alternativen wie `fgets()` oder `gets_s()` verwendet werden (cppreference.com, 2012).

Um ein *buffer overflow* auszunutzen und den Programmablauf zu modifizieren, bedarf es an Kenntnis über das Speicherlayout des Programms sowie des verwendeten Algorithmus (Lhee & Chapin, 2003, S. 425).

Um die Funktion `secret()` aus Listing 2 auszuführen, müsste ein Angreifer die Rücksprungadresse auf dem Stack gezielt mit der Adresse dieser Funktion überschreiben, sodass das Programm beim Verlassen der verwundbaren Funktion nicht wie vorgesehen weiterläuft, sondern stattdessen die Schadfunktion aufruft.

Trotz langer Forschung und zahlreicher Schutzmechanismen (e.g. *Stack Canaries*) spielen *buffer overflows* eine Gefahr im Bereich der Speicherkorruption dar. Ein Grund dafür ist die weite Verbreitung unsicherer C Programme, die keine Grenzprüfungen vornehmen. Während auf der einen Seite kontinuierlich neue Verteidigungsmaßnahmen - sowohl auf Binärebene als auch Compilererweiterungen - entwickelt werden, verbessern Angreifer gleichzeitig ihre Techniken, um diese zu umgehen (Tanenbaum & Bos, 2015, S. 649).

2.1.2.2 Dangling Pointer und Use-after-Free

Ein *Dangling Pointer* entsteht, wenn ein Zeiger auf ein Objekt verweist, das bereits freigegeben wurde. Der Zeiger zeigt dann auf einen Speicherbereich, der entweder bereits neu alloziert oder überschrieben worden sein kann. Solche Situationen beeinträchtigen die Korrektheit und Sicherheit eines Programms erheblich, da sie die Grundlage für schwerwiegende Sicherheitslücken wie *Use-after-Free* oder *Double-Free* bilden können (Caballero et al., 2012, S. 133).

Use-after-Free bezeichnet dabei den Fall, dass ein Programm versucht, über einen Dangling Pointer auf bereits freigegebenen Speicher zuzugreifen. Dies kann zu unvorher-

sehbarem Verhalten, Speicherbeschädigungen oder Sicherheitslücken führen. Besonders problematisch ist, dass die Entstehung und Nutzung des Dangling Pointers zeitlich weit auseinanderliegen können, was die Identifikation und Diagnose solcher Fehler erschwert. Die Analyse solcher Fehler verlangt oft ein Verständnis über mehrere Speicherobjekte hinweg, insbesondere wenn Zeiger gemeinsam auf denselben Speicherbereich zeigen (Cabbalero et al., 2012, S. 133).

Folgendes Code Beispiel soll *Dangling Pointer* und *Use-after-Free* verdeutlichen. Hierbei wird ein Zeiger auf ein Objekt erstellt, das anschließend gelöscht wird. Der Zeiger verweist dann auf einen Speicherbereich, der nicht mehr gültig ist. Der Zugriff auf diesen Speicherbereich führt zu undefiniertem Verhalten.

```
1  #include <iostream>
2  #include <string>
3
4  class User {
5  public:
6      std::string name;
7
8      User(const std::string& n) {
9          name = n;
10     }
11
12     void printName() {
13         std::cout << "Name: " << name << std::endl;
14     }
15 };
16
17 int main() {
18     User* user1 = new User("Alice");
19     User* user2 = user1; // user2 points to the same object as
20                         ↪ user1
21
22     user2->printName(); // Output: Name: Alice
23     delete user1; // memory is freed and user1 and user2 are
24                 ↪ dangling pointers
25
26     user2->printName(); // Use-after-Free: accessing memory we do
27                     ↪ not own anymore -> undefined behavior
28 }
```

Listing 3: Dangling Pointer / Use after Free Beispiel

2.1.2.3 Memory Leak

Memory Leaks treten in Sprachen mit expliziter Speicherverwaltung auf. Hier muss der Entwickler selbst unbenutzten Speicher freigeben, was leicht übersehen wird. Insbesondere bei lang laufenden Anwendungen führt dieses schleichende Vollaufen des Speichers auf Dauer zu Leistungsproblemen oder sogar zu Abstürzen. Zusätzlich sind Speicherlecks nur schwer aufzuspüren, da sie sich vorrangig durch einen allmählichen Anstieg der Speicherbelegung bemerkbar machen und sonst kaum Symptome zeigen. Gerade in zeitlich lang laufenden Applikationen stellen Memory Leaks deshalb weiterhin eine große Herausforderung bei der Analyse dar (Xie & Aiken, 2005, S. 1).

Das Beispielcode zeigt einen *Memory Leak* in C++. Hierbei wird ein Objekt auf dem Heap alloziert, aber nie freigegeben. Dies führt dazu, dass der Speicher, der für das Objekt reserviert wurde, nicht mehr zugänglich ist und somit nicht wiederverwendet werden kann.

```
1  #include <iostream>
2
3  class User {
4  public:
5      std::string name;
6
7      User(const std::string& n) {
8          name = n;
9      }
10 };
11
12 static void printUserAlice() {
13     User* user = new User("Alice"); // Allocates memory on the heap
14     std::cout << "User: " << user->name << std::endl;
15     // Memory is not freed, leading to a memory leak
16 }
17
18 static void doSomething() {
19     /* do something */
20 }
21
22 static void doSomethingElse() {
23     /* do something else */
24 }
25
26 int main() {
27     doSomething();
28     printUserAlice();
29     doSomethingElse();
30 }
```

Listing 4: Beispiel für Memory Leak in C++

2.1.3 Common Weakness Enumeration

Die *Common Weakness Enumeration* (CWE™) stellt eine von der Community entwickelte Liste gängiger Schwachstellentypen in Software und Hardware dar, die potenziell sicherheitsrelevant sind. Ein „Weakness“-Eintrag beschreibt eine Bedingung in Software-, Firmware-, Hardware- oder Dienstkomponenten, die unter bestimmten Umständen zur Entstehung von Sicherheitslücken beitragen kann und häufig bereits während der Entwicklungsphase eingeführt wird (MITRE Corporation, 2025a).

Jeder CWE-Eintrag ist durch eine eindeutige Nummer (CWE-ID) gekennzeichnet und umfasst neben einem prägnanten Namen und einer Kurzbeschreibung weitere Pflichtinformationen wie eine ausführliche Beschreibung, Einführungsmodi, potenzielle Gegenmaßnahmen, typische Folgen, betroffene Plattformen, Demonstrationsbeispiele, in der Praxis beobachtete Beispiele, Beziehungen zu anderen Schwachstellen sowie Referenzen (MITRE Corporation, 2025a).

Die Nutzung von CWE ist kostenfrei für jedermann – sei es für Forschungs-, Entwicklungs- oder kommerzielle Zwecke – gemäß den Lizenzbedingungen der MITRE Corporation (MITRE Corporation, 2025a).

CWE wird von Softwareentwicklern und Sicherheitsexperten als gemeinsame Sprache verwendet, um Schwachstellen in Architektur, Design, Code und Implementierung zu identifizieren, zu klassifizieren und zu beheben. Darüber hinaus dient CWE als Standardbasis für die Evaluierung von Sicherheitstools und als Grundlage für Maßnahmen zur Schwachstellenprävention in Organisationen (MITRE Corporation, 2025a).

Zur *Community* gehören Organisationen aus Regierung, Industrie und Wissenschaft. Es betreibt verschieden Arbeits- und Interessensgruppen zur Förderung der Nutzung und Erweiterung der *CWE*-Liste. Community-Mitglieder können sich beteiligen, indem sie an Diskussionen teilnehmen, Inhaltsvorschläge einreichen, *CWE*-kompatible Produkte verwenden, *CWE* aktiv über soziale Medien fördern und die Nutzung von *CWE*-Ressourcen vorantreiben (MITRE Corporation, 2025b).

2.1.4 Traditionelle Codeanalyse Methoden

2.1.4.1 Statische Codeanalyse

In der Softwareentwicklung bezeichnet statische Codeanalyse das Prüfen von Programmcode auf potenzielle Fehler, ohne den Code auszuführen. Bereits in den 1970er-Jahren entwickelte Stephen Johnson bei den Bell Laboratories ein frühes Werkzeug namens Lint, das C-Programme nach versteckten Fehlern durchsuchte, die dem Compiler entgangen waren. Im Gegensatz zur dynamischen Analyse - also dem Testen durch Ausführen des Programms - erfolgt die statische Analyse rein auf Grundlage des Quelltextes. Lint durchkämmt den Code nach bekannten Fehlermustern, anstatt das Programm laufen zu lassen und dessen Verhalten zu prüfen. Solche statischen Code-Prüfer kommen nach dem erfolgreichen Kompilieren, aber noch vor dem Testen zum Einsatz, um typische Fallstricke aufzudecken, solange ihre Behebung noch kostengünstig und einfach ist (Louridas, 2006, S. 1).

Statische Code Analyse Tools kommen somit bereits in der Implementierungsphase von Software zum Einsatz um den Code robuster zu machen und *wiederkehrende Fallen* in die Entwickler Tappen zu reduzieren. Es ist die Aufgabe der statischen Codeanalyse, den Quellcode auf potenzielle Fehler zu überprüfen und Schwächen aufzuspüren und dem Entwickler die Codestelle anzuzeigen, die den Fehler verursacht. Schwächen werden dabei in Schweregrade unterteilt die mit einer Datenbank, sogenannten *Common Weakness Enumeration* (CWE, 2.1.3) verknüpft sind. Die folgende Grafik von Alexandru G. Bardas verdeutlicht hierbei den zeitlichen Einsatz von statischen Codeanalyse Tools im Entwicklungsprozess und zeigt, dass diese Tools in der Regel vor dem Testen des Codes bereits eingesetzt werden (Bardas, o.J. S. 1–2).

Es gibt Tools für die statische Code Analyse, die nicht nur den Quellcode überprüfen, sondern auch den Bytecode analysieren. Die Analyse des Quellcodes ist genauer, die Bytecodeanalyse jedoch schneller. Sie bauen eine abstrakte Programmdarstellung auf und nutzen Datenflussanalysen, um Schwachstellen wie Buffer Overflows oder SQL-Injections zu erkennen. Kein Checker ist vollständig oder fehlerfrei – sie finden nicht alle Fehler und produzieren auch false-positives, also falsch-positive Ergebnisse. Ihre Eignung sollte deshalb immer im Kontext des jeweiligen Projekts bewertet werden. Viele Tools erlauben benutzerdefinierte Regeln um gezielt Fehler zu vermeiden. Fehlerfreiheit garantiert jedoch nicht hohe Codequalität, dafür sind zusätzliche Metriken notwendig (Bardas, o.J.

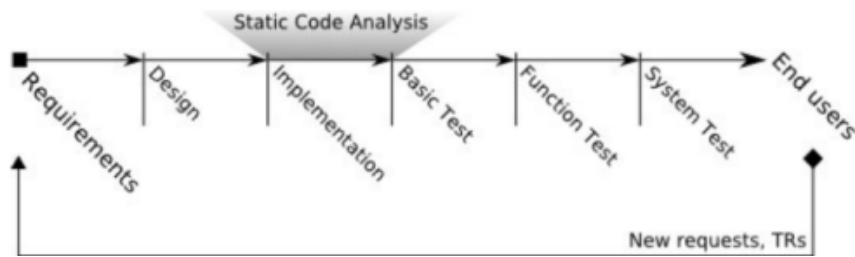


Abbildung 1: Einsatz von statischen Codeanalyse Tools im Entwicklungsprozess (Bardas, o.J. S. 1).

S. 2–3).

Folgendes Codebeispiel in Listing 5 zeigt eine einfache statische Codeanalyse in C++ mit dem Tool *clang-tidy*. Das Listing ist stark vereinfacht und soll lediglich die Funktionsweise verdeutlichen. Aus der Dokumentation von *clang-tidy*:

Clang-tidy ist ein auf Clang basierendes Tool zur statischen Analyse von C++-Code. Es dient als erweiterbares Framework zur Erkennung und Behebung typischer Programmierfehler wie Stilverletzungen, falscher Schnittstellennutzung oder analysierbarer Bugs“ (LLVM Project, o.J.).

```

1  #include <iostream>
2
3  int main() {
4      int* p = new int[10]; // Allocate an array of 10 integers
5      delete p;
6      return 0;
7  }
```

Listing 5: Beispiel für statische Codeanalyse mit clang-tidy

Folgender Screenshot 2 aus dem demo Projekt zeigt die Ausgabe von *clang-tidy* integriert in Visual Studio Code.

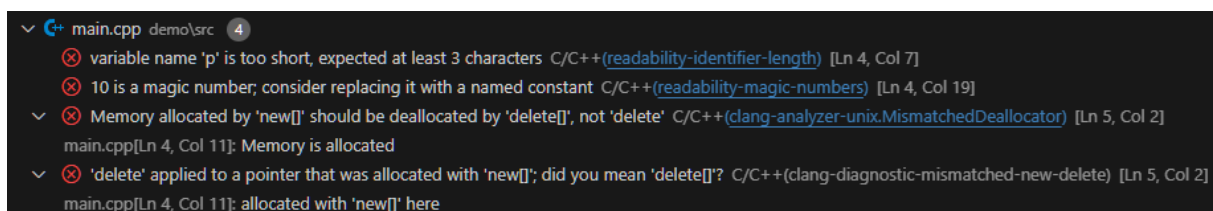


Abbildung 2: Screenshot von clang-tidy in Visual Studio Code.

Zum einen werden Lesbarkeitsprobleme wie zu kurze Variablennamen (“p” ist zu kurz) und die Verwendung von sogenannten Magic Numbers (hier die Zahl 10) angemahnt, die durch benannte Konstanten ersetzt werden sollten. Diese Hinweise dienen der Verbesserung der Codequalität und Wartbarkeit.

Zum anderen zeigt das Tool auch sicherheitsrelevante Probleme bei der Speicherverwaltung auf. Konkret warnt *clang-tidy* vor einem Mismatched Deallocator – also der fehlerhaften Verwendung von `delete` statt `delete[]` bei einem mit `new[]` angelegten Array. Solche Fehler können zu undefiniertem Verhalten oder Speicherfehlern führen. *clang-tidy* erkennt diesen Sachverhalt automatisch und schlägt die korrekte Lösung direkt vor.

2.1.4.2 Dynamische Codeanalyse

Die dynamische Analyse bezeichnet die Untersuchung von Eigenschaften eines laufenden Programms. Sie ermittelt Eigenschaften, die für eine oder mehrere konkrete Programmausführungen gelten. Dies geschieht in der Regel durch Ausführung und Benutzung der Software zur Laufzeit. Im Gegensatz zur statischen Analyse, die den Programmtext analysiert, um Eigenschaften zu bestimmen, die für alle Ausführungen gelten, leitet die dynamische Analyse Eigenschaften ab, die speziell für eine oder mehrere konkrete Ausführungen gültig sind (Bell, 1999, S. 217). Zur Durchführung dynamischer Analysen wird unter anderen Varianten dem untersuchten Programm sogenannter Instrumentierungscode hinzugefügt – kleine Codeabschnitte, die Programmereignisse aufzeichnen oder Daten sammeln. Solche Instrumentierung kann in jeder Phase der Kompilierung erfolgen (Larus & Ball, 1994, S. 1).

Der Vollständigkeit halber sei erwähnt, dass auch andere Techniken zur dynamischen Codeanalyse existieren, wie zum Beispiel der *profiling*-basierte Ansatz. Hierbei werden *profiling* und *debugging* Mechanismen von virtuellen Maschinen wie in Microsofts *.NET Framework* oder *JPDA* für die *Java SDK* verwendet. Die Analyse ist dabei als Plugin umgesetzt und wird als sogenannter *profiling agent* aufgerufen. Diese Plugins greifen dann auf den *profiling* Service der virtuellen Maschine über Interfaces zu (Gosain & Sharma, 2015, S. 116). Ein spezifischer Ansatz profilbasierter dynamischer Analyse ist die sogenannte Frequency Spectrum Analysis. Dabei wird untersucht, wie häufig bestimmte Programmelemente während einer Programmausführung aufgerufen werden. Dies hilft Entwicklern, zusammengehörige Programmteile zu identifizieren und Rückschlüsse auf die Struktur und Abhängigkeiten im Programm zu ziehen (Bell, 1999, S. 218).

Die Nützlichkeit der dynamischen Analyse ergibt sich aus zwei zentralen Eigenschaften: Erstens ermöglicht sie eine hohe Genauigkeit der gesammelten Informationen. Durch Instrumentierung des Programms kann gezielt genau die Laufzeitinformation erfasst werden, die zur Lösung eines spezifischen Problems benötigt wird. So kann man beispielsweise zur Analyse der Struktur dynamisch erzeugter Daten (wie Listen, Bäume oder gerichtete Graphen) ein Instrumentierungstool entwickeln, das gezielt die Verknüpfungen zwischen Speicherzellen im Heap aufzeichnet. Zweitens basiert die dynamische Analyse auf konkreten Programmeingaben, was zwar ihre Vollständigkeit einschränkt, ihr aber gleichzeitig eine große Stärke verleiht: Sie erlaubt eine direkte Beobachtung und Verknüpfung von Eingaben, internem Verhalten und Ausgaben eines Programms. Dadurch lassen sich Veränderungen im Verhalten und in den Ergebnissen des Programms klar auf Veränderungen in den Eingabedaten zurückführen (Bell, 1999, S. 217). In diesem Sinne könnte man sagen, dass dynamische Analyse eher eingabenzentriert und statische Analyse eher programmzentriert ist (Bell, 1999, S. 217).

Dynamische Analyse leidet im Gegensatz zur statischen Analyse nicht unter dem Problem der nicht ausführbaren Pfade (infeasible paths), da sie tatsächliche Programmausführungen betrachtet. Sie erzeugt dynamische Invarianten, die zwar präzise und konkret, aber auf die tatsächlich getesteten Programmpfade beschränkt sind. Daher bietet sie keine Garantie, dass diese Eigenschaften für alle möglichen Ausführungen gelten, während statische Analyse umfassender, jedoch abstrakter und weniger präzise sein kann (Bell, 1999, S. 217).

Besonders leistungsfähig sind sogenannte Shadow-Value-Tools, die zusätzliche Informationen zu Speicherinhalten erfassen und so komplexe Fehler wie undefinierte Zugriffe oder Datenlecks aufdecken können. Ihr Nachteil liegt im hohen Laufzeit-Overhead und der aufwendigen Implementierung. Man unterscheidet zwischen leichtgewichtigen Tools mit

geringem Einfluss auf die Performance und schwergewichtigen Tools mit tiefer Analyse, aber höherem Ressourcenverbrauch (Nethercote & Seward, 2007, S. 1–2).

Zur Veranschaulichung der Möglichkeiten dynamischer Analyse lassen sich verschiedene praxisrelevante Tools nennen. Memcheck (Teil von Valgrind) erkennt uninitialisierte Speicherzugriffe in C/C++-Programmen. TaintCheck verfolgt den Fluss unsicherer Eingabedaten und hilft so bei der Erkennung potenzieller Sicherheitslücken. Redux ermöglicht eine Visualisierung des gesamten Datenflusses zur besseren Nachvollziehbarkeit komplexer Programmabläufe. Weitere spezialisierte Tools wie Annelid (zur Erkennung von Array-Grenzverletzungen) oder Hobbess (zur dynamischen Typanalyse) zeigen die Vielfalt und Anwendbarkeit dynamischer Binäranalyse in sicherheitskritischen Kontexten (Nethercote & Seward, 2007, S. 1).

Abbildung 8 zeigt beispielhaft eine dynamische Analyse mittels des Intel Inspectors. Das Tool erkennt hierbei mehrere typische Speicherprobleme wie etwa inkorrekte Speicherfreigaben ("Mismatched allocation/deallocation"), Zugriffe außerhalb der Array-Grenzen ("Invalid memory access") sowie Zugriffe auf nicht initialisierte Speicherbereiche ("Uninitialized memory access"). Diese visuelle Darstellung hilft Entwicklern, die genaue Stelle im Quellcode sowie die Ursache des Problems direkt zu erkennen und anschließend gezielt beheben zu können.

The screenshot displays the Intel Inspector interface. The 'Problems' table lists the following issues:

ID	Type	Sources	Modules	Object Size	State
P1	Mismatched allocation/deallocation	main.cpp	bachelorthesisdemo.exe		New
P2	Kernel resource leak	iostream	bachelorthesisdemo.exe		New
P3	Kernel resource leak	[Unknown]	pghook.dll		New
P4	Invalid memory access	main.cpp	bachelorthesisdemo.exe		New
P5	Invalid partial memory access	[Unknown]	pghook.dll		New
P6	Uninitialized memory access	main.cpp	bachelorthesisdemo.exe		New
P7	Uninitialized partial memory access	[Unknown]	pghook.dll		New
P8	Uninitialized partial memory access	[Unknown]	pghook.dll		New

The 'Code Locations' pane shows the source code for 'Mismatched deallocation site' and 'Allocation site'.

```

Mismatched deallocation site
Source: main.cpp:11
Function: main
Module: bachelorthesisdemo.exe
Object Size:
Offset:
Variable: block allocated at main.cpp:4
9 p[5] = 30; // access out of bounds (undefined b
10
11 delete p; // wrong delete, should be delete[]
12
13 return 0;
Allocation site
Source: main.cpp:4
Function: main
Module: bachelorthesisdemo.exe
Object Size:
Offset:
Variable: block allocated at main.cpp:4
2
3 int main() {
4 int* p = new int[5]; // Allocate an array reserving space for
5
6 p[0] = 10;

```

The 'Timeline' pane shows the execution flow, including the function `RtlActivateActivationContextUnsafeFast`.

Abbildung 3: Beispiel einer dynamischen Speicheranalyse mit Intel Inspector

Listing 6 zeigt den verwendeten *Quellcode*, der kompiliert wurde und dessen Kompilat anschließend mit dem Intel Inspector analysiert wurde.

```
1  #include <iostream>
2
3  int main() {
4      int* p = new int[5];          // Allocate an array reserving space
4      ↪ for 5 integers
5
6      p[0] = 10;
7      p[1] = 20;
8      std::cout << p[4] << std::endl; // access uninitialized
8      ↪ memory (undefined behavior)
9      p[5] = 30;                    // access out of bounds (
9      ↪ undefined behavior)
10
11     delete p;                      // wrong delete, should be
11     ↪ delete[] p;
12
13     return 0;
14 }
```

Listing 6: Demonprogramm source code in C++

2.1.5 Large Language Models

Large Language Models (LLMs) sind eine spezielle Klasse von Sprachmodellen, die auf der Transformer-Architektur basieren und durch umfangreiches Training auf großen Textkorpora bemerkenswerte Fähigkeiten in der Verarbeitung natürlicher Sprache entwickelt haben. Sie ermöglichen es Maschinen, menschenähnlichen Text zu generieren und komplexe sprachliche Aufgaben zu bewältigen (Moradi et al., 2024, S. 3–6).

LLMs sind tief neuronale Netzwerke mit Milliarden von Parametern, die durch selbstüberwachtes Lernen auf riesigen Mengen von Textdaten trainiert werden. Die zugrunde liegende Transformer-Architektur, eingeführt von Vaswani et al. (2017), nutzt Mechanismen wie Selbstaufmerksamkeit und Positionskodierung, um den Kontext und die Bedeutung von Wörtern in einem Satz effektiv zu erfassen. Dies ermöglicht es dem Modell, die Wahrscheinlichkeit des Auftretens eines bestimmten Wortes basierend auf seinem Kontext vorherzusagen (Vaswani et al., 2023, S. 2–5).

Das Training von LLMs erfordert enorme Rechenressourcen und große Mengen an Textdaten. Durch das Training auf umfangreichen und vielfältigen Datensätzen entwickeln LLMs die Fähigkeit, Muster in der Sprache zu erkennen und zu generalisieren. Dies führt zu beeindruckenden Leistungen in Aufgaben wie Textklassifikation, maschineller Übersetzung und Textgenerierung (Moradi et al., 2024, S. 3–6).

Das Training großer Sprachmodelle erfolgt typischerweise in zwei Phasen: einem allgemeinen Pre-Training und einem darauf aufbauenden Fine-Tuning. Während des Pre-Trainings werden neuronale Netze auf umfangreichen und vielfältigen Textkorpora trainiert. Ziel ist es, statistische Muster, grammatikalische Strukturen und semantische Zusammenhänge in der Sprache zu erlernen, ohne dass explizit gelabelte Daten benötigt werden - ein Verfahren, das als self-supervised learning bezeichnet wird. Je nach Modellarchitektur werden dabei unterschiedliche Trainingsziele verfolgt, etwa das Vorhersagen maskierter Tokens (BERT) oder das sequentielle Generieren von Text (GPT). Die trainierten Parameter speichern das sprachliche Wissen des Modells und bilden die Grundlage für nachgelagerte Anwendungen. In der Fine-Tuning-Phase wird das Modell dann auf spezifische Aufgaben oder Domänen angepasst, um seine Leistungsfähigkeit in gezielten Anwendungsszenarien weiter zu verbessern (Moradi et al., 2024, S. 6–8).

LLMs finden Anwendung in verschiedenen Bereichen, darunter Bildung, Medizin und Wirtschaft. Sie unterstützen bei der Analyse großer Textmengen, der Beantwortung von Fragen und der Generierung von Zusammenfassungen. In der Medizin können LLMs beispielsweise bei der Interpretation von klinischen Daten helfen und die Entscheidungsfindung unterstützen (L. Fan et al., 2023, pp. 3-5, 20).

Trotz ihrer beeindruckenden Fähigkeiten stehen Large Language Models (LLMs) vor spezifischen Herausforderungen. Eine zentrale Problematik ist das sogenannte *Halluzinieren*, bei dem die Modelle sachlich falsche, aber sprachlich überzeugende Inhalte generieren. Dieses Phänomen entsteht dadurch, dass LLMs auf Grundlage statistischer Wahrscheinlichkeiten das jeweils nächste Wort in einer Sequenz vorhersagen - basierend auf Mustern aus ihren Trainingsdaten, nicht auf geprüftem Wissen. Da sie keinen direkten Zugriff auf verifizierte Informationsquellen besitzen, kann dies zur Erzeugung von Fehlinformationen führen, insbesondere wenn die Eingabe mehrdeutig ist oder das benötigte Wissen nicht ausreichend im Trainingsmaterial repräsentiert war (Moradi et al., 2024, S. 16).

Neben der inhaltlichen Korrektheit spielen auch ethische Aspekte eine wichtige Rolle im Umgang mit LLMs. Dazu gehören unter anderem Fragen zu Datenschutz, Verzerrungen durch Trainingsdaten, sowie die potenzielle Verbreitung von manipulativen oder diskriminierenden Inhalten. Die Weiterentwicklung von Methoden zur Verbesserung der Genauig-

keit, Vertrauenswürdigkeit und ethischen Vertretbarkeit von LLMs ist daher ein zentrales Anliegen aktueller Forschung. Ansätze wie Retrieval-Augmented Generation (RAG), die externe Wissensquellen einbinden, oder Reinforcement Learning from Human Feedback (RLHF), bei dem menschliche Rückmeldungen genutzt werden, um die Modellantworten besser an gesellschaftliche Werte und Faktenlage anzupassen, stellen vielversprechende Strategien dar (Moradi et al., 2024, S. 13–17).

2.1.6 Software Assurance Reference Dataset

2.1.6.1 Über Software Assurance Reference Dataset (SARD)

Das *Software Assurance Reference Dataset* ist eine umfangreiche und stetig wachsende Sammlung von Testprogrammen mit dokumentierten Schwachstellen. Die Testfälle reichen von kleinen, synthetischen Codebeispielen bis hin zu komplexer produktionsnaher Software. Aktuell umfasst die Sammlung über 450.000 Testfälle, die in verschiedenen Programmiersprachen wie *C*, *C++*, *Java*, *PHP* und *C#* vorliegen.

Die Testfälle decken über 150 Klassen von Schwachstellen gemäß der Common Weakness Enumeration (CWE) ab, darunter vielfältige sicherheitsrelevante Fehlerarten. Ziel von SARD ist es, die Software Assurance zu fördern - also die Gewährleistung von Sicherheit und Qualität im gesamten Softwareentwicklungsprozess.

Die Plattform unterstützt sowohl die Forschung als auch die Weiterentwicklung von Analysetools zur Schwachstellenerkennung. Entwickeln von Sicherheitswerkzeugen bietet SARD eine breite Basis zum Testen und Validieren ihrer Lösungen. Gleichzeitig hilft sie Endanwendern, geeignete Werkzeuge für spezifische Projektanforderungen zu finden.

Das Projekt wird vom *Software Assurance Metrics and Tool Evaluation* (SAMATE) Team betrieben, das daran arbeitet, standardisierte Methoden für die Evaluierung von Software Assurance Tools zu etablieren. Angesichts der enormen wirtschaftlichen Schäden, die durch mangelnde Softwaresicherheit entstehen, versteht sich SARD als wichtiger Beitrag zur Erhöhung der Softwarequalität und -sicherheit.

2.1.6.2 Aufbau der Testfälle

Die Testfälle aus dem *Software Assurance Reference Dataset* enthalten jeweils einen *Status*, der ihre Validität und Relevanz beschreibt. Für diese Arbeit werden ausschließlich Testfälle mit dem Status **Accepted** verwendet. Dieser Status wird von der *Software Assurance Reference Dataset* des NIST vergeben, wenn ein Testfall bestimmten Qualitäts- und Reproduzierbarkeitskriterien genügt (National Institute of Standards and Technology (NIST), 2023).

Accepted-Testfälle erfüllen laut Definition des NIST unter anderem folgende Eigenschaften:

- Sie demonstrieren klar eine spezifische Schwachstelle entsprechend einer CWE-Kategorie (z. B. CWE-121 für Buffer Overflow).
- Sie sind kompiliert- und ausführbar.
- Die fehlerhafte Stelle ist klar dokumentiert und nachvollziehbar.
- Es existieren sowohl eine fehlerhafte (bad) als auch eine korrekte (good) Version.

Der Einsatz ausschließlich akzeptierter Testfälle erhöht die methodische Zuverlässigkeit des Experiments. Er stellt sicher, dass die zugrundeliegenden Schwachstellen eindeutig definiert und dokumentiert sind, was die Bewertung der Erkennung und Behebung durch KI-Systeme und Analysetools objektiv nachvollziehbar macht. Darüber hinaus fördern standardisierte und geprüfte Testfälle die Vergleichbarkeit der Ergebnisse mit anderen Studien, die ebenfalls auf der Juliet Test Suite basieren (National Institute of Standards and Technology (NIST), 2023).

2.2 Stand des Wissens

2.2.1 AI Chatbots als Ersatz zu statischen Codeanalyse-Tools

(Ozturk et al., 2023) untersuchten die Effektivität von KI-gestützten Tools im Vergleich zu traditionellen statischen Codeanalyse-Tools hinsichtlich der Erkennung von Sicherheitslücken. Der Fokus ihrer Untersuchung lag auf *PHP*, die besonders anfällig für Sicherheitsprobleme ist, da sie nur wenige eingebaute Sicherheitsmechanismen anbietet. Das primäre Ziel der Studie war es herauszufinden, ob KI-basierte Lösungen, konkret das Modell *ChatGPT* von OpenAI, in der Lage sind, etablierte statische Codeanalyse-Tools effektiv zu ergänzen wenn nicht gar zu ersetzen.

Für ihre Untersuchung nutzten die Autoren eine eigens entwickelte “code-repository” mit insgesamt 92 PHP-Codebeispielen, die bewusst Sicherheitslücken enthielten. Diese Beispiele basierten auf den *OWASP* Top 10 Webanwendungsrisiken von 2021 und deckten ein breites Spektrum gängiger Schwachstellen ab. Jeder dieser Codebeispiele wurde mit entsprechenden *CWE-Nummern* eindeutig kategorisiert und zur Validierung manuell getestet (Ozturk et al., 2023).

Zur Evaluation der Ergebnisse unterteilten die Autoren den Output der analysierten Tools in Kategorien. Die untersuchten Codes wurden zunächst generell in die Kategorien *verwundbar* und *sicher* eingeteilt. Bei den als verwundbar eingestuften Codes wurde weiter in die Subkategorien *gefunden*, *teilweise gefunden* und *nicht gefunden* unterschieden. Die Kriterien für *gefunden* waren hierbei, dass das Tool die Sicherheitslücke korrekt klassifizierte und eindeutig die betroffenen Codebestandteile, wie Funktionen oder Variablen, hervorhob. Wurde eine Sicherheitslücke zwar erkannt, aber nicht korrekt klassifiziert oder nur teilweise richtig zugeordnet, wurde dies als *teilweise gefunden* bewertet. Alle anderen Fälle wurden als *nicht gefunden* klassifiziert. Bei den sicheren Codes erfolgte eine Einteilung in *korrekt* und *falsch positiv*, wobei *falsch positiv* bedeutet, dass ein Tool eine Sicherheitslücke meldete, obwohl der Code sicher war (Ozturk et al., 2023).

Im experimentellen Vergleich zeigte sich, dass *ChatGPT* zwischen 62% und 68% der Sicherheitslücken korrekt identifizieren konnte. Im Gegensatz dazu erreichte das beste getestete statische Analyse-Tool lediglich eine Erkennungsrate von 32%. Selbst die Kombination der Stärken mehrerer traditioneller Tools konnte nur eine maximale Erfolgsquote von 53% erzielen, was weiterhin deutlich unterhalb der von *ChatGPT* erzielten Erkennungsrate liegt. Allerdings wies *ChatGPT* auch eine außergewöhnliche hohe Rate an falsch-positiven Ergebnissen (91%) auf, im Vergleich zum schlechtesten traditionellen Tool, welches 82% erreichte (Ozturk et al., 2023).

Ozturk et al., 2023 betonen in ihrer Analyse das beträchtliche Potenzial von *ChatGPT*, den Prozess der statischen Codeanalyse erheblich zu verbessern. Jedoch unterstreichen sie gleichzeitig die Herausforderungen, insbesondere durch die hohe Anzahl falsch-positiver Ergebnisse, die in der Praxis eine sorgfältige Nachkontrolle erforderlich machen würden. Darüber hinaus äußerten sie Datenschutzbedenken, da bei der Nutzung solcher KI-gestützten Werkzeuge oft der gesamte Code online eingereicht werden muss, was potenzielle Risiken im Umgang mit sensiblen Daten birgt (Ozturk et al., 2023).

Aufgrund dieser gemischten Ergebnisse schlugen die Autoren vor, dass zukünftige Forschungen sich stärker auf hybride Ansätze konzentrieren sollten. Diese könnten die Stärken von KI-basierten Systemen wie *ChatGPT* mit den etablierten Verfahren traditioneller statischer Codeanalyse-Tools kombinieren, um so die Erkennung von Sicherheitslücken sowohl in Bezug auf Genauigkeit als auch auf Zuverlässigkeit optimal zu gestalten (Ozturk et al., 2023).

2.2.2 Intelligent Code Analysis Agents (ICAA) als neue Form der statischen Codeanalyse

Das Paper von G. Fan et al., 2023 untersucht die Grenzen traditioneller Methoden zur statischen Codeanalyse und stellt das Konzept sogenannter *Intelligent Code Analysis Agents (ICAA)* vor. Aufgrund zunehmender Komplexität von Software und immer kürzerer Entwicklungszyklen reichen klassische Techniken wie *Code Reviews*, *Unit Tests* und *Integrationstests* nicht mehr aus, um komplexe Logikfehler und Sicherheitslücken zuverlässig aufzuspüren, insbesondere stoßen diese traditionellen Ansätze bei der Analyse semantischer Fehler und der Identifikation komplexer funktionaler Inkonsistenzen an ihre Grenzen, was oft zu einer hohen Anzahl false positives führt und somit den manuellen Aufwand für Entwickler stark erhöht.

Vor diesem Hintergrund schlagen Fan et. al. eine Kombination aus klassischen statischen Codeanalyse-Methoden und KI-gestützten Methoden vor, insbesondere durch den Einsatz von großen Sprachmodellen (*LLMs*) wie *GPT-3* oder *GPT-4*. Diese Modelle bieten aufgrund ihrer Fähigkeit, natürliche Sprache und Programmcode zu verstehen, erhebliches Potenzial zur Erkennung subtiler semantischer Inkonsistenzen und logischer Fehler. Die Autoren präsentieren den "*ReAct Bug Detection Agent*", der mittels einer Hierarchie von Sub-Agenten arbeitet und neben statischen Analysetools auch Kontextinformationen und Code-Dokumentationen verwendet, sowie den *Code-Intention Consistency Checking Agent*, der die Intention hinter Codeabschnitten analysiert und mit deren Implementierung abgleicht. Grafik 4 zeigt den Workflow des *React Bug Detection Agent*:

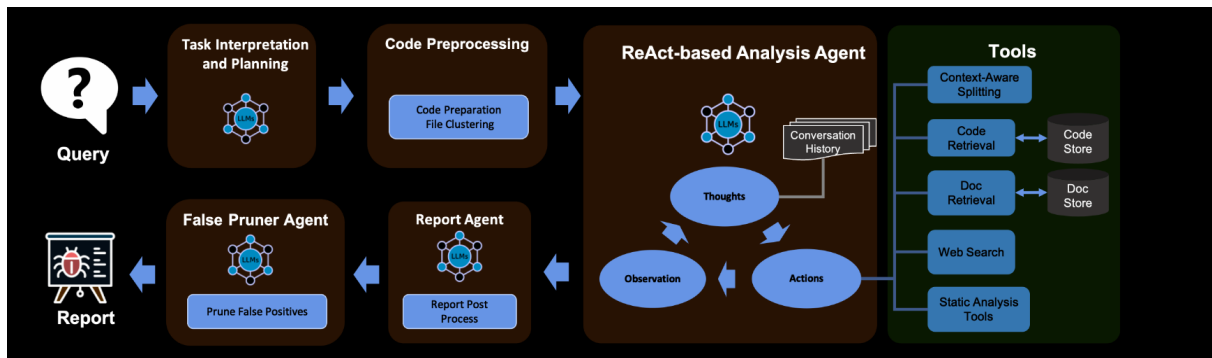


Abbildung 4: ReAct Bug Detection Agent (G. Fan et al., 2023, S. 7)

Der ReAct Bug Detection Agent arbeitet mithilfe eines sogenannten ReAct-Frameworks, bei dem der Agent durch ein iteratives Vorgehen bestehend aus Gedanken, Aktionen und Beobachtungen mit einem Sprachmodell interagiert. Dabei verwendet er Werkzeuge wie Kontext-sensitive Codesegmentierung, Retrieval-Tools für relevante Codebeispiele und Dokumentationen sowie statische Analysetools, um die Codeanalyse effektiv durchzuführen. Der Code-Intention Consistency Checking Agent nutzt Techniken des Prompt Engineering, um mögliche Inkonsistenzen zwischen der Implementierung und den zugehörigen Kommentaren, Dokumentationen und Funktionsnamen aufzudecken. Beide Agenten arbeiten dabei nicht streng linear, sondern verwenden dynamische, kontextabhängige Entscheidungsprozesse.

Die Evaluation dieser Agenten anhand realistischer Benchmarks und realer Projekte zeigt vielversprechende Ergebnisse. So konnten G. Fan et al., 2023 mit ihrem Ansatz die false-positive Rate von 85% auf 66% senken und eine *Recall-Rate* (Treffergenauigkeit) von etwa 60,8% erreichen. Trotz der hohen Effizienz bei der Fehlererkennung, weisen die

Autoren darauf hin, dass der Einsatz von *LLMs* mit einem nicht unerheblichen Kosten-
aufwand für Token-Verbrauch verbunden ist, was eine breite Anwendung derzeit noch
einschränkt. Hierbei schätzen die Autoren die Kosten für die Analyse einer Codezeile
zwischen \$0.000812 und \$0.001032. Unter der Annahme, dass ein Softwareprojekt aus
1.000.000 lines of code besteht, belaufen sich die Kosten für einen einzelnen Analyselauf
auf rund \$1000.

2.2.3 Wiederverwendbare Prompt-Patterns

Die Autoren White et al., 2023 untersuchen in ihrem Paper “A Prompt Pattern Ca-
talog to Enhance Prompt Engineering for ChatGPT”, wie sich bewährte Konzepte aus
der Software-Patterns-Literatur auf das Prompt Engineering für große, konversationelle
Sprachmodelle übertragen lassen. Sie argumentieren, dass Prompts als Form von Program-
mierung verstanden werden können, die wiederverwendbare, modulare “Prompt Patterns”
erfordert, um Interaktionen mit LLMs systematisch zu gestalten und zu optimieren.

Zur Erarbeitung ihres Frameworks adaptieren die Forschenden die klassische Pattern-
Form (Name, Intent, Motivation, Struktur/Kernaussagen, Beispiel, Konsequenzen) und
passen sie an die Besonderheiten des Prompt-Kontexts an. Als Datengrundlage dienen
Community-Quellen (“community-posted prompts”) sowie systematische Experimente mit
ChatGPT+: Die Patterns werden iterativ in realen Chat-Sessions getestet, dokumentiert
und hinsichtlich ihrer Wirksamkeit geprüft. Dabei legen sie besonderen Wert auf so ge-
nannte “Fundamental Contextual Statements”, mit denen die zentralen Ideen eines Pat-
terns auf einer höheren Abstraktionsebene erfasst werden (White et al., 2023).

Als Ergebnis präsentieren die Autor:innen einen umfassenden Katalog von 16 Prompt-
Patterns, der in fünf Hauptkategorien strukturiert ist:

Pattern Category	Prompt Pattern
Input Semantics	Meta Language Creation
Output Customization	Output Automater Persona Visualization Generator Recipe Template
Error Identification	Fact Check List Reflection
Prompt Improvement	Question Refinement Alternative Approaches Cognitive Verifier Refusal Breaker
Interaction	Flipped Interaction Game Play Infinite Generation
Context Control	Context Manager

Tabelle 1: Kategorien und Beispiele der Prompt-Patterns White et al., 2023

Jedes Pattern wird anhand von Intent, Motivation, Kernaussagen, Beispielimplemen-
tierungen und möglichen Konsequenzen beschrieben. Darüber hinaus zeigen die Autor:innen,

wie sich mehrere Patterns kombinieren lassen – etwa Game Play plus Visualization Generator, um interaktive Sicherheits-Demos mit Diagrammausgaben zu erstellen. Abschließend reflektieren sie, dass Prompt-Patterns die Gestaltungsfreiheit bei LLM-Interaktionen deutlich erweitern, generalisierbar auf unterschiedliche Domänen sind und laufend an die sich wandelnden Modellfähigkeiten angepasst werden müssen White et al., 2023.

2.2.4 Verständnis von Code-Syntax und -Semantik durch Large Language Models

Die Autoren Ma et al., 2024 untersuchen in ihrer Studie die Fähigkeiten großer Sprachmodelle in Bezug auf das Verständnis von Code-Syntax und -Semantik für Software Engineering-Aufgaben, speziell für die Codeanalyse. Zu den zentralen Anforderungen für LLMs im Bereich der Codeanalyse zählt das Verständnis der Syntax sowie das statische sowie dynamische Verhalten von Programmen. Zur Evaluation dieser Fähigkeiten, verwendeten die Autoren die LLMs von *OpenAI* (GPT-3.5 und GPT-4) und StarCoder, ein Open-Source-Sprachmodell, das von *Hugging Face* in Zusammenarbeit mit *ServiceNow Research* entwickelt wurde, sowie CodeLlama-13b-instruct, dem OpenSource Sprachmodell von *Meta*. Die untersuchten Programmiersprachen umfassten *C*, *Java*, *Python* und *Solidity*.

Die Autoren entwarfen insgesamt neun Aufgaben, um die Fähigkeiten der Modelle zur Analyse von Quellcode zu testen. Diese Aufgaben umfassten beispielsweise die Erzeugung von abstrakten Syntaxbäumen (AST), Kontrollfluss- und Aufrufgraphen sowie Datenflussanalysen, Taint- und Zeigeranalysen. Darüber hinaus wurden auch komplexere Analysen wie die Erkennung äquivalenter Mutationen und die Begründung für instabile Testergebnisse (Flaky Tests) durchgeführt. Die Auswertung erfolgte anhand eines eigens erstellten Datensatzes mit über 2.500 Codebeispielen aus den vier oben erwähnten Programmiersprachen. Die Resultate wurden sowohl qualitativ als auch quantitativ miteinander verglichen.

Die Ergebnisse der Studie zeigen, dass LLMs grundsätzlich ein gutes Verständnis der Code-Syntax besitzen, was sich in der erfolgreichen Generierung von abstrakten Syntaxbäumen widerspiegelt. Jedoch offenbarten sie Schwächen beim Verständnis der Semantik, besonders der dynamischen Semantik. Dies führte dazu, dass LLMs zwar statische Strukturen wie Kontrollflussgraphen und Aufrufgraphen teilweise korrekt erzeugen konnten, bei komplexeren Aufgaben wie Datenabhängigkeitsanalyse, Taint-Analyse und Zeigeranalyse jedoch Schwächen aufwiesen.

Die Autoren der Studie kommen zu dem Ergebnis, dass LLMs Schwierigkeiten beim Verständnis des dynamischen Programmverhalten haben. Tests zur Erkennung äquivalenter Mutationen und das Analysieren von nicht deterministischen Testfällen wurden mangelhaft erfüllt. Dies legen die Autoren dem Mangel an entsprechenden dynamischen Kontextinformationen im Trainingsdatensatz der Modelle zugrunde.

Eine weitere Herausforderung für die untersuchten Modelle ist die *Halluzination* (siehe 2.1.5). Dieses Verhalten reduziert die Zuverlässigkeit der Modelle für kritische Softwareentwicklungsaufgaben erheblich.

Ma et al., 2024 schlussfolgern, dass insbesondere das Modell *GPT-4* unter den getesteten Modellen das beste Verständnis von Code-Syntax und statischen Strukturen zeigte. Dennoch weisen sie auf die Notwendigkeit hin, Mechanismen zur Validierung der von LLMs erzeugten Ergebnisse zu entwickeln, um deren Zuverlässigkeit und Nutzbarkeit in sicherheitskritischen Softwareentwicklungsumgebungen zu gewährleisten. Ihre Studie liefert somit wichtige Erkenntnisse zur praktischen Anwendung von LLMs im Bereich der

Softwareentwicklung und zeigt auf, welche Fähigkeiten verbessert werden müssen, um die Zuverlässigkeit solcher Modelle weiter zu steigern.

3 Vorgangsweise und Methoden

3.1 Vorgehensweise

Die vorliegende Arbeit verfolgt einen empirischen Ansatz zur Beantwortung der Forschungsfrage, inwiefern moderne KI-gestützte Code-Assistenten in der Lage sind, sicherheitsbezogene Sicherheitslücken in C/C++-Programmen zu erkennen und zu beheben.

Dazu wird ein vergleichendes Studiendesign gewählt, das sowohl KI-Systeme als auch etablierte Analysewerkzeuge gegenüberstellt. Die Untersuchung erfolgt anhand synthetischer und realitätsnaher Beispielprogramme mit gezielten Schwachstellen, die systematisch ausgewählt, analysiert und bewertet werden.

Die Untersuchung folgt dem experimentellen Paradigma der Softwaretechnikforschung, wie es etwa in Wohlin et al. (2012) beschrieben ist: Durch kontrollierte Eingaben (fehlerhafte Quelltexte) und reproduzierbare Analysen (standardisierte Prompts, feste Testumgebung) werden die Reaktionen der Systeme auf konsistente Weise erhoben. Die Ergebnisse werden anschließend qualitativ und quantitativ ausgewertet.

Zur Sicherstellung von Nachvollziehbarkeit und Replizierbarkeit wird die gesamte Untersuchung dokumentiert. Alle Testfälle, Prompts und Ergebnisse werden archiviert. Die Bewertung folgt einem transparenten Schema, das sowohl Erkennung als auch Lösungsvorschläge klassifiziert (siehe Abschnitt 4.5).

3.2 Methoden

Zur Beantwortung der Forschungsfrage werden qualitative und quantitative Analysemethoden kombiniert, um sowohl die Erkennungsleistung als auch die Qualität der Lösungsvorschläge der untersuchten Systeme zu bewerten.

Die empirische Erhebung erfolgt auf Basis definierter Testfälle, die gezielt Schwachstellen nach dem *Common Weakness Enumeration (CWE)*-Standard abbilden. Diese stammen aus dem *Software Assurance Reference Dataset (SARD)* des *National Institute of Standards and Technology (NIST)*, insbesondere aus der *Juliet Test Suite* sowie ergänzenden C/C++-Testkollektionen. Für jede Fehlerklasse (z. B. Buffer Overflow, Use-after-Free, Memory Leak) werden jeweils eine fehlerhafte (*bad*) und eine korrekte (*good*) Implementierung analysiert.

Um die Testfälle selbst testen und analysieren zu können, wird ein eigenes leichtgewichtiges Testframework in C++ entwickelt. Dieses ermöglicht die strukturierte Ausführung aller Testfälle in isolierten Prozessen und bietet damit eine stabile Umgebung zur dynamischen Analyse, insbesondere für die Nutzung mit Tools wie dem *Intel Inspector*. Die genaue technische Umsetzung des Frameworks wird in Kapitel 4.6.2 erläutert.

Drei KI-gestützte Code-Assistenten (*ChatGPT*, *Claude*, *Gemini*) werden über ein standardisiertes Prompt-Template mit fehlerhaften Codefragmenten konfrontiert. Das Template definiert eine einheitliche Rollenbeschreibung und Aufgabenstellung, wobei lediglich der eingebettete Quellcode für jeden Testfall angepasst wird. Dadurch wird sichergestellt, dass alle Systeme unter vergleichbaren Bedingungen evaluiert werden. Die Antworten der KI-Assistenten werden im Anschluss manuell klassifiziert.

Zur Einordnung der KI-Ergebnisse werden ergänzend zwei etablierte Codeanalysetools herangezogen: der *Clang Static Analyzer* zur statischen Analyse sowie der *Intel Inspector* zur dynamischen Laufzeitanalyse. Diese dienen als Referenz zur Einordnung der Leistungsfähigkeit der KI-Systeme.

Die Bewertung der Erkennung und Behebung erfolgt anhand eines standardisierten Kategorienschemas, das die Ausgaben sowohl der KI-Assistenten als auch der klassischen Analysewerkzeuge vergleichbar macht. Dieses Bewertungsschema unterscheidet u. a. zwischen vollständiger, teilweiser oder fehlender Erkennung sowie effektiver oder unzureichender Behebung der Schwachstellen.

4 Empirisches Vorgehen

4.1 Übersicht über Tools und Plattformen

Die in Kapitel 3 beschriebenen Werkzeuge und Plattformen kommen in der praktischen Durchführung zur Anwendung.

Die KI-Assistenten über ihre jeweiligen Webschnittstellen bedient. Die klassischen Analysetools, namentlich der *Clang Static Analyzer* und der *Intel Inspector*, werden lokal auf einem Windows Rechner mit Windows 11 installiert. Tabelle 2 gibt einen Überblick über die zu verwendenden KI-Assistenten, deren zugrunde liegende Modelle und die Plattformen, auf denen sie bereitgestellt werden.

KI-Assistent	Anbieter	Modell	Zugriffsplattform
ChatGPT	OpenAI	GPT-4o	chat.openai.com
Claude	Anthropic	Claude 3.7 Sonnet	perplexity.ai
Gemini	Google	Gemini 2.5 Pro	perplexity.ai

Tabelle 2: Übersicht der verwendeten KI-Assistenten, Modelle und Plattformen

Um die Vergleichbarkeit der Ergebnisse zu gewährleisten, wird sichergestellt, dass sämtliche Tools – sowohl KI-gestützte als auch klassische – auf dieselben vorbereiteten Testfälle angewendet werden.

Tabelle 3 zeigt die Spezifikationen der Testumgebung, auf der die Analyse durchgeführt wird. Diese Umgebung wurde speziell für die Durchführung der Tests konzipiert und stellt sicher, dass alle relevanten Faktoren kontrolliert und reproduzierbar sind.

Komponente	Spezifikation / Version
Testgerät	HP ZBook Fury G8
Arbeitsspeicher	32 GB RAM
Betriebssystem	Windows 11 Pro (64-bit)
Entwicklungsumgebung	Visual Studio Code 1.99.3
Build-System	CMake 3.29.2
Compiler	MSVC 19.43.34810.0 (Visual Studio 2022 Professional)
KI-Plattformen	ChatGPT, Perplexity
Statisches Analysetool	Clang Static Analyzer (Version 17)
Dynamisches Analysetool	Intel Inspector 2023

Tabelle 3: Spezifikationen der Testumgebung

4.2 Vorgangsweise des Experiments

Grafik 5 zeigt den gesamten Ablauf der Untersuchung, beginnend mit der Auswahl der Testfälle aus dem *SARD*-Repository bis hin zur Auswertung, Dokumentation und Zusammenfassung der Ergebnisse. Die Grafik soll einen Überblick über den Forschungsprozess geben und das Vorgehen illustrieren. Die Folgekapitel geben detaillierte Einblicke in die einzelnen Schritte des Experiments, einschließlich der Auswahl der Testfälle, der Implementierung des Testframeworks, der Analyse der Ergebnisse und der Bewertung der Leistung der untersuchten Systeme. Die Untersuchung wird in mehreren Phasen durchgeführt, die aufeinander aufbauen. Zunächst werden die Testfälle ausgewählt und implementiert, gefolgt von der Analyse der Ergebnisse der KI-Assistenten und der klassischen Analysetools. Abschließend erfolgt eine umfassende Auswertung und Dokumentation der Ergebnisse.

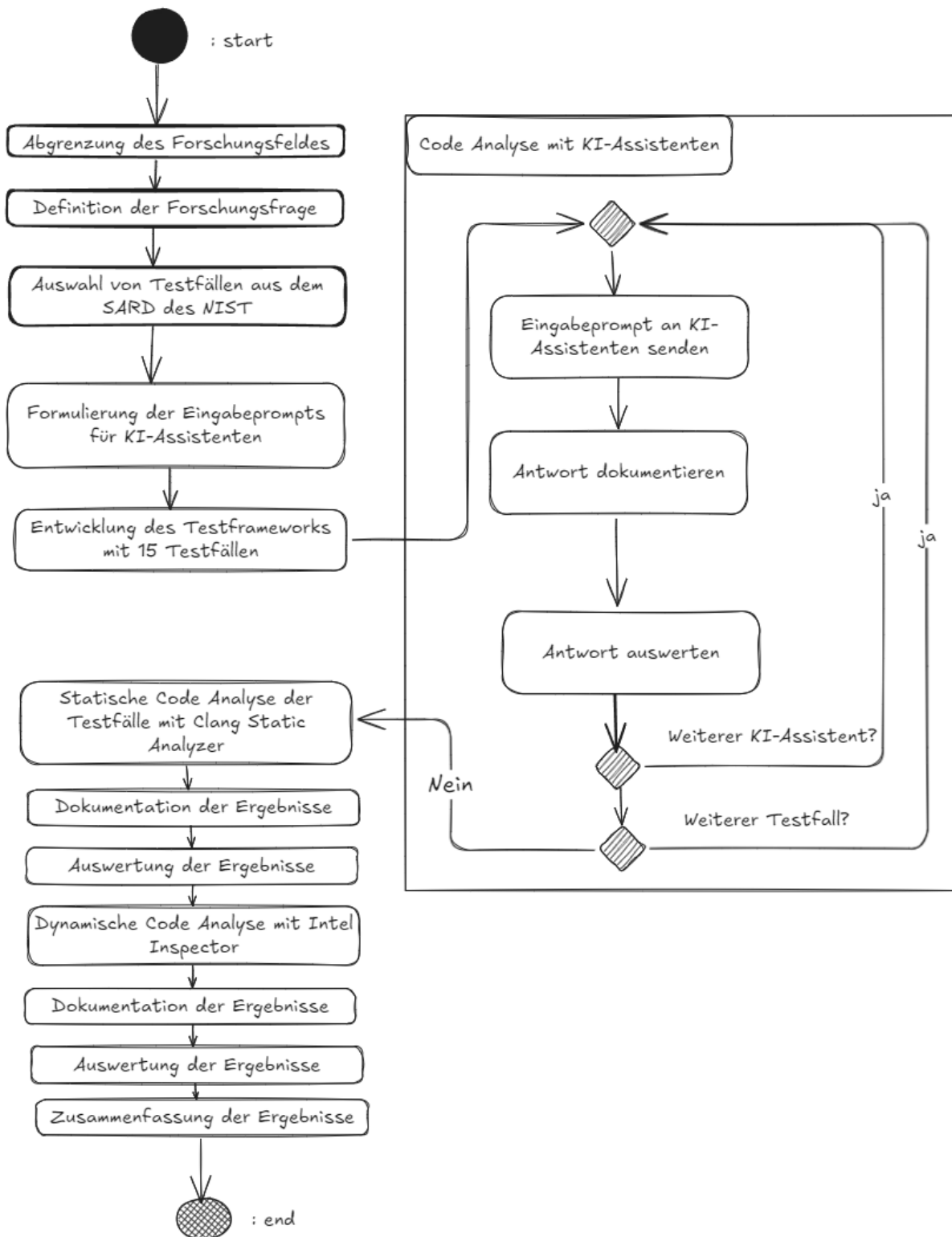


Abbildung 5: Ablauf der Untersuchung

4.3 Auswahl der Testfälle

Zur gezielten Analyse speicherbezogener Sicherheitslücken wurden in dieser Arbeit konkrete Testfälle aus dem *Software Assurance Reference Dataset (SARD)* des *National Institute of Standards and Technology (NIST)* ausgewählt.

Die ausgewählten Testfälle wurden so gewählt, dass sie realistische Schwachstellen reproduzieren, dabei aber isoliert und kontrolliert ausführbar bleiben. Tabelle 4 zeigt die in dieser Arbeit aktuell eingesetzten Testfälle. Jeder Testfall ist eindeutig einer CWE-Kategorie zugeordnet, enthält eine dokumentierte Quelle und verweist, wenn möglich, auf die originale Test-ID im SARD-Repository.

Der Name der Testfälle folgt dem Schema **TC<X>_<B|G>**, wobei **X** die Testfallnummer und **B|G** den Typ des Testfalls angibt: **B** für *bad* (fehlerhaft) und **G** für *good* (korrekt).

Die Einbeziehung sowohl fehlerhafter (*bad*) als auch korrekter (*good*) Varianten einzelner Testfälle ermöglicht eine differenzierte Bewertung der Analyseergebnisse. Während die *bad*-Fälle gezielt Sicherheitslücken wie Buffer Overflows reproduzieren, dienen die *good*-Fälle als fehlerfreie Referenzimplementierungen.

Diese Referenzen sind insbesondere für den Vergleich von Toolausgaben von Bedeutung: In der dynamischen Analyse etwa sollte ein Analysewerkzeug bei *good*-Fällen keine Speicherfehler detektieren, während *bad*-Fälle entsprechende Warnungen oder Abstürze erzeugen können. Auch im Kontext der Bewertung von KI-Assistenten lassen sich durch den Vergleich der von der KI vorgeschlagenen "Fixes" mit den vorhandenen *good*-Varianten qualitative Aussagen über die Qualität und Angemessenheit der automatisierten Korrekturansätze treffen.

Die *good*-Testfälle sind daher nicht nur Kontrollinstanz, sondern liefern zusätzlich eine Orientierung hinsichtlich guter sicherheitsrelevanter Programmierpraxis.

Tabelle 4: Definition der Testfälle und erwartetes Verhalten

ID	CWE	Fehlerart	Beschreibung	SARD-Referenz
TC1_B	121	Stack Buffer Overflow	Ungeprüfte Pufferkopie auf dem Stack	231444
TC1_G	-	Stack Buffer Overflow	Sichere Längenprüfung vor Kopie	231445
TC2_B	122	Heap Buffer Overflow	Überschreiben dynamisch allokierten Speichers	149083
TC2_G	-	Heap Buffer Overflow	Allokation mit korrekter Puffergröße	149084
TC3_B	124	Buffer Underwrite	Pointer wird um 8 <code>wchar_t</code> vor den allokierten Puffer gesetzt und anschließend mit <code>wcsncpy</code> kopiert	234602
TC3_G	-	Buffer Underwrite	Datenzeiger korrekt auf allokierten Puffer gesetzt	234602
TC4_B	121, 129	Stack Buffer Overflow, Improper Validation of Array Index	Unzureichende Prüfung des Array-Index vor Zugriff auf lokal allokiertes Array	62542

Fortsetzung von Tabelle 4

ID	CWE	Fehlerart	Beschreibung	SARD-Referenz
TC4_G	-	Stack Buffer Overflow	Korrekte Prüfung des Array-Index vor Zugriff auf lokal allokiertes Array	62542
TC5_B	121	Stack Buffer Overflow	Platzierung eines Objekts mit placement new in zu kleinem Stack-Puffer	67044
TC5_G	-	Stack Buffer Overflow	Objekt wird korrekt in ausreichend großem Stack-Puffer platziert	67044
TC6_B	416	Use After Free	Zugriff auf bereits mit free freigegebenen Heap-Puffer	240375
TC6_G	-	Use After Free	Gültiger Zugriff auf nicht freigegebenen oder korrekt verwalteten Heap-Puffer	240375
TC7_B	416	Use After Free	Zugriff auf bereits mit free freigegebenen Heap-Puffer nach Allokation mit malloc	240322
TC7_G	-	Use After Free	Speicher wird nach malloc nicht vorzeitig freigegeben, Zugriff erfolgt auf gültigen Bereich, aber mit Memory Leak	240322
TC8_B	416	Use After Free	Selbstzuweisung eines Objekts mit fehlerhaft implementiertem operator= führt zu Zugriff auf bereits freigegebenen Speicher	102645
TC8_G	-	Use After Free	Selbstzuweisung wird durch Prüfung im operator= erkannt und verhindert, Speicherzugriff bleibt gültig	102645
TC9_B	416	Use After Free	Rückgabe eines bereits mit free freigegebenen Zeigers aus einer Funktion; anschließende Nutzung führt zu undefiniertem Verhalten	240400
TC9_G	-	Use After Free	Speicher wird nicht vor der Rückgabe freigegeben; Zugriff erfolgt nur auf gültige Daten, aber potenzielles Memory Leak bei fehlendem free im Aufrufer	240400
TC10_B	416	Use After Free	Zugriff auf bereits mit delete gelöscht Objekt, ausgelöst durch verschachtelte Kontrollflussentscheidungen	102524

Fortsetzung von Tabelle 4

ID	CWE	Fehlerart	Beschreibung	SARD-Referenz
TC10_G	-	Use After Free	Zugriff auf gelöscht Objekt wird vollständig vermieden, obwohl Speicher wie in der unsicheren Variante freigegeben wird	102524
TC11_B	401	Memory Leak	<code>wchar_t</code> -Puffer wird per <code>realloc</code> mehrfach alloziert, aber nicht freigegeben	240070
TC11_G	-	Memory Leak	<code>wchar_t</code> -Puffer wird per <code>realloc</code> alloziert und korrekt mit <code>free()</code> freigegeben	240070
TC12_B	401	Memory Leak	<code>char</code> -Puffer wird per <code>calloc</code> alloziert, aber nicht freigegeben	99244
TC12_G	-	Memory Leak	<code>char</code> -Puffer wird per <code>calloc</code> alloziert und korrekt mit <code>free()</code> freigegeben	99244
TC13_B	401	Memory Leak	Speicher wird mit <code>new[]</code> alloziert, aber je nach Kontrollfluss nicht mit <code>delete[]</code> freigegeben	99760
TC13_G	-	Memory Leak	Speicher wird per <code>new[]</code> alloziert und über mehrere Varianten korrekt freigegeben	99760
TC14_B	401	Memory Leak	<code>TwoIntsClass</code> -Array wird mit <code>new[]</code> alloziert, aber nie gelöscht	99829
TC14_G	-	Memory Leak	Vier Varianten vermeiden Memory Leak bei <code>TwoIntsClass</code> durch korrekte Freigabe oder Stack-Allokation	99829
TC15_B	401	Memory Leak	Kein virtueller Destruktor: <code>delete</code> auf Basisklasse ruft nicht den Destruktor der abgeleiteten Klasse auf	100714
TC15_G	-	Memory Leak	Virtueller Destruktor sorgt für vollständige Freigabe aller abgeleiteten Ressourcen	100714

4.4 Definition des Prompts für KI

Für die Analyse von *C/C++*-Quellcode mittels der in dieser Arbeit betrachteten KI-Systeme wird ein zentraler Prompt definiert. Dieser Prompt wird allen KIs übergeben und hat die Aufgabe, sicherheitsrelevante Schwachstellen im Code zu erkennen und die Ergebnisse in einer für Menschen gut lesbaren Form zurückzugeben. Die Qualität dieses Prompts ist dabei entscheidend, da sie maßgeblich die Korrektheit, Konsistenz und Reproduzierbarkeit der Resultate beeinflusst. Tabelle 5 zeigt die Anforderungen an den zentralen Prompt für die Sicherheitsanalyse von *C/C++*-Code die inspiriert von dem Paper White et al., 2023 erstellt wurden. Diese Anforderungen sind in drei Kategorien unterteilt: **Persona Pattern**, **Template Pattern** und **Context Manager Pattern**. Diese Patterns helfen dabei, den Prompt so zu gestalten, dass er qualitativ hochwertigere Antworten liefert und gleichzeitig die Interaktion mit der KI optimiert.

Pattern	Beschreibung
Persona Pattern	Die KI erhält die Rolle eines <i>C/C++</i> -Sicherheitsprüfers, damit sie sich auf sicherheitsrelevante Aspekte konzentriert und keine anderen Code-Reviews (z. B. Stil- oder Architekturfragen) durchführt.
Template Pattern	Für die Ausgabe wird eine Markdown-Tabelle definiert, welche die Analyse der Ausgabe der KI erleichtert.
Context Manager Pattern	Die KI wird angewiesen, ausschließlich auf Speicher sowie Sicherheitsrelevante Schwachstellen und deren Behebung einzugehen.

Tabelle 5: Angewandte Patterns für das Prompt

Prompt für Sicherheitsanalyse

Du bist ein Sicherheitsexperte mit Spezialisierung auf *C/C++*-Codeanalyse. Dein Ziel ist es, sicherheitsrelevante Speicherverwaltungsfehler zu erkennen und zu beheben. Analysiere den folgenden *C/C++*-Code ausschließlich im Hinblick auf sicherheitskritische Speicherfehler wie Buffer Overflows, Use-after-free oder Memory Leaks.

Erzeuge eine Tabelle in Latex mit den folgenden Spalten:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
-----	--------	------------------------	-----------	--------------	------------

Analysiere ausschließlich sicherheitsrelevante Aspekte, keine Stil- oder Architekturfragen.

“*C/C++* Code hier einfügen”

Die KI erhält dabei die Rolle eines Sicherheitsexperten für *C/C++*, wodurch sichergestellt wird, dass der Fokus ausschließlich auf sicherheitskritischen Aspekten der Spei-

cherverwaltung liegt. Dies verhindert, dass sich das Modell mit irrelevanten Themen wie Code-Stil, Namensgebung oder allgemeiner Softwarearchitektur beschäftigt.

Der Prompt lenkt die Aufmerksamkeit gezielt auf drei klar definierte Fehlertypen: Buffer Overflows, Use-after-free und Memory Leaks.

Die Antwort der KI erfolgt in einer **Markdown-Tabelle**, die für die spätere quantitative Auswertung direkt übernommen werden kann. Diese Tabelle enthält die folgenden Spalten:

- **Nr.** – eine fortlaufende Nummerierung der erkannten Schwachstellen,
- **CWE-ID** – eine Referenz auf die entsprechende Schwachstellenklasse aus der *Common Weakness Enumeration*,
- **Name der Schwachstelle** – die Bezeichnung gemäß CWE (z. B. „Heap-Based Buffer Overflow“),
- **Fehlerart** – z. B. „Buffer Overflow“, „Use-after-free“, „Memory Leak“,
- **Beschreibung** – eine kurze Erläuterung der gefundenen Problematik,
- **Empfehlung** – ein Vorschlag zur Behebung, idealerweise inklusive konkreter Codeänderung.

Diese tabellarische Struktur erlaubt eine systematische Auswertung, sowohl hinsichtlich der Fehlererkennung als auch der Qualität der vorgeschlagenen Lösung. Durch das einheitliche Format ist zudem ein direkter Vergleich zwischen den verschiedenen KI-Assistenten möglich.

Beispielhafte Ausgabe der KI im Tabellenformat:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-122	Heap-Based Buffer Overflow	Buffer Overflow	Der Speicherbereich wird ohne Größenprüfung beschrieben, was zu einer Überschreitung führen kann.	Verwendung von <code>strncpy()</code> mit Längenprüfung.
2	CWE-401	Memory Leak	Memory Leak	Allokierter Speicher wird nicht freigegeben, nachdem er nicht mehr benötigt wird.	Einfügen eines <code>free()</code> -Aufrufs am Funktionsende.

Tabelle 6: Beispielhafte KI-Ausgabe im standardisierten Tabellenformat

4.5 Bewertungsschema für die Erkennung und Behebung von Speicherfehlern

In diesem Kapitel wird das Bewertungsschema vorgestellt, mit dem die Leistung der untersuchten Tools bei der Erkennung und Behebung von sicherheitsrelevanten Speicherfehlern beurteilt wird. Bewertet werden sowohl KI-basierte Sprachmodelle (LLMs) als auch klassische statische und dynamische Analysetools. Die Bewertung erfolgt anhand zweier Hauptkategorien:

- **Erkennung der Schwachstelle**
- **Lösungsvorschlag zur Behebung der Schwachstelle**

Da der Fokus auf der Vergleichbarkeit der Werkzeuge liegt, wurde bewusst ein vereinfachtes, klar messbares Bewertungsschema gewählt, das sich streng an der im SARD-Testfall dokumentierten CWE-ID orientiert.

Eine Schwachstelle gilt nur dann als **vollständig erkannt**, wenn das jeweilige Tool **ausschließlich** die CWE-ID benennt, die im zugehörigen SARD-Testfall dokumentiert ist (z.,B. CWE-121 bei einem Stack-basierten Buffer Overflow), und **keine weiteren, nicht dokumentierten Schwachstellen** nennt oder analysiert. Wird eine inhaltlich ähnliche, aber abweichende CWE-ID genannt (z. B. CWE-120), oder wird die Schwachstelle lediglich beschrieben, aber ohne die korrekte Nummer zu nennen, so gilt dies als **teilweise erkannt**. Wird weder die richtige CWE genannt, noch die Schwachstelle inhaltlich korrekt beschrieben, wird der Fall als **nicht erkannt** bewertet.

Diese klare Abgrenzung ermöglicht eine objektive, reproduzierbare Kategorisierung und orientiert sich bewusst an der Ground-Truth-Definition der SARD-Testfälle. Auch wenn semantisch richtige Beschreibungen in Einzelfällen auf Verständnis hinweisen können, steht in dieser Arbeit die *präzise Identifikation im Sinne der CWE-Systematik* im Vordergrund.

Die Bewertung des Lösungsvorschlags erfolgt unabhängig von der Erkennung. Ein Fehler gilt als **vollständig behoben**, wenn der vorgeschlagene Code die dokumentierte Schwachstelle vollständig beseitigt, also das fehlerhafte Verhalten nachweislich verhindert. Wird die Schwachstelle nur teilweise adressiert oder bleiben Sicherheitsrisiken bestehen, so wird dies als **teilweise behoben** bewertet. Vorschläge, die das Problem nicht beheben oder neue Fehler einführen, gelten als **nicht behoben**.

Die Bewertung wurde auf Basis einer qualitativen Analyse des vorgeschlagenen Quellcodes durchgeführt, unter Berücksichtigung typischer Sicherheitsanforderungen (z. B. Bounds-Prüfungen, Speicherfreigabe, Vermeidung von undefiniertem Verhalten).

Tabelle 7 fasst die Kriterien zusammen:

Kategorie	Beschreibung
Erkennung der Schwachstelle	
Vollständig erkannt	Die Schwachstelle wurde erkannt und exakt die CWE-ID genannt, die im SARD-Testfall dokumentiert ist.
Teilweise erkannt	Die Schwachstelle wurde beschrieben oder eine verwandte CWE-ID wurde genannt, jedoch nicht exakt die im SARD-Testfall angegebene.
Nicht erkannt	Die Schwachstelle wurde weder korrekt beschrieben noch eine passende CWE-ID genannt.
Lösungsvorschlag zur Behebung	
Vollständig behoben	Der Lösungsvorschlag beseitigt die dokumentierte Schwachstelle vollständig und nachhaltig, ohne neue Fehler einzuführen.
Teilweise behoben	Der Lösungsvorschlag adressiert das Problem nur unvollständig oder lässt Teile der Schwachstelle bestehen.
Nicht behoben	Der Fehler wird nicht behoben oder es entstehen neue sicherheitskritische Probleme durch die vorgeschlagene Änderung.

Tabelle 7: Definition des Bewertungsschemas der Tools

4.6 Implementierung der Testfälle

4.6.1 Aufbau des eigenen Testframeworks

4.6.1.1 Struktur

Im Rahmen des Experiments wird ein eigenes, leichtgewichtiges Testframework in *C++* entwickelt. Ziel ist es, eine flexible und gut kontrollierbare Testumgebung zu schaffen, die gezielt ausgewählte Speicherverwaltungsfehler reproduziert und so eine effektive Analyse sowohl durch klassische als auch KI-gestützte Werkzeuge ermöglicht.

Während das *Juliet Test Suite* eine große Bandbreite an Testfällen für unterschiedliche Sicherheitslücken bietet, ist sie für die fokussierte Analyse einzelner Fehlerarten – wie etwa *Buffer Overflows*, *Use-after-Free* oder *Memory Leaks* – oft zu umfangreich und komplex. Daher wurde ein eigenes Testframework implementiert, das folgende Vorteile bietet:

- gezielte Auswahl typischer Fehlerfälle mit präzisiertem Fokus,
- vollständige Kontrolle über Aufbau und Ablauf jedes Testfalls,
- einfache Erweiterbarkeit um neue Testfälle oder Fehlerklassen,
- automatische und robuste Ausführung über Kommandozeilenparameter,
- sichere Isolation einzelner Testfälle durch separate Prozesse.

Jeder Testfall ist als eigene Funktion implementiert und wird nicht direkt im Hauptprozess ausgeführt, sondern in einem separaten Kindprozess gestartet. Dadurch wird sichergestellt, dass selbst bei einem Absturz (z. B. durch einen *Segmentation Fault*) nur der jeweilige Testprozess betroffen ist und die restlichen Testfälle weiterhin analysiert werden können.

Das Projekt wurde modular aufgebaut und verwendet *CMake* als Toolchain. Die Quellcode-dateien sind thematisch in Unterordner gegliedert, wobei jeder Ordner eine Fehlerart abbildet. Die Gesamtstruktur ist in Listing 7 dargestellt.

```
Bachelorarbeit_Testframework/  
├── CMakeLists.txt  
└── src/  
    ├── buffer_overflow/  
    │   ├── buffer_tests.cpp  
    │   └── buffer_tests.h  
    ├── use_after_free/  
    │   ├── Testfall11.cpp  
    │   └── Testfall11.h  
    ├── memory_leak/  
    │   ├── Testfall11.cpp  
    │   └── Testfall11.h  
    ├── helper/  
    │   ├── helper_structures.h  
    │   └── test_output.h  
    └── main.cpp
```

Listing 7: Projektstruktur des Testframeworks

In Listing 8 ist die verwendete `CMakeLists.txt` Datei abgebildet, welche das gesamte Projekt konfiguriert und mit dem C++20-Standard kompiliert.

```
1 cmake_minimum_required(VERSION 3.10)  
2 project(BachelorThesisDemo VERSION 0.1.0 LANGUAGES C CXX)  
3  
4 enable_testing()  
5  
6 file(GLOB_RECURSE SRC_FILES  
7     src/*.cpp  
8     src/buffer_overflow/*.cpp  
9     src/use_after_free/*.cpp  
10    src/memory_leak/*.cpp  
11 )  
12  
13 add_executable(BachelorThesisDemo ${SRC_FILES})  
14  
15 set_property(TARGET BachelorThesisDemo PROPERTY CXX_STANDARD 20)
```

Listing 8: CMakeLists.txt für das Testframework

4.6.1.2 Ablaufsteuerung und Ausführung der Testfälle

Die Ablaufsteuerung des Testframeworks erfolgt über den Einstiegspunkt, die `main`-Funktion. Hier werden zunächst grundlegende Initialisierungen vorgenommen, darunter die Aktivierung der Heap-Debugging-Funktionalitäten unter Windows (mittels `_CrtSetDbgFlag`), um die Erkennung von Speicherlecks zu erleichtern, sowie die Setzung der Konsolencodierung auf UTF-8.

Der Programmstart erfolgt über Kommandozeilenargumente. Das erste übergebene Argument bestimmt dabei, ob einzelne Testfälle (z.,B. `TC1_G`), ganze Gruppen von Testfällen (`good`, `bad`), oder sämtliche Testfälle (`all`) ausgeführt werden sollen. Abhängig vom über-

gebenen Parameter werden intern definierte Listen mit Testfallnamen iterativ abgearbeitet und jeder Testfall in einem eigenen Subprozess gestartet.

Ein beispielhafter Auszug aus der `main`-Funktion ist in Listing 9 dargestellt:

```
1 int main(int argc, char** argv) {
2     // Heap-Debugging aktivieren
3     _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
4     srand((unsigned)time(NULL));
5
6     SetConsoleOutputCP(CP_UTF8);
7     std::cout.imbue(std::locale(""));
8     std::wcout.imbue(std::locale(""));
9
10    if (argc < 2) {
11        std::cerr << "Usage: " << argv[0] << " <mode>\n";
12        std::cerr << "Modes: good, bad, all, TC1_B, TC2_B, TC1_G, TC2_G
13        ↪ ... \n";
14        return -1;
15    }
16    std::string mode = argv[1];
17
18    if (mode == "TC1_B" ||
19        mode == "TC2_B" ||
20        // ...)
21
22    if (mode == "good") {
23        run_good_tests();
24    } else if (mode == "bad") {
25        run_bad_tests();
26    } else if (mode == "all") {
27        run_good_tests();
28        run_bad_tests();
29    } else {
30        std::cout << "Unknown mode: " << mode << "\n";
31        return 1;
32    }
33
34    return 0;
35 }
```

Listing 9: Auszug aus der `main`-Funktion des Testframeworks

Die Testfallgruppen (`good`, `bad` und `all`) sind in der Funktion `run_good_tests()` als auch `run_bad_tests()` definiert. Diese Funktionen iterieren über die Liste der Testfälle und rufen für jeden Testfall die Funktion `start_test_process(testname)` auf, die den jeweiligen Testfall im neuen Subprozess startet.

```
1 void run_bad_tests() {
2     for (const auto& testname : bad_tests) {
3         start_test_process(testname);
4     }
5 }
```



```
5 }  
6  
7 void run_good_tests() {  
8     for (const auto& testname : good_tests) {  
9         start_test_process(testname);  
10    }  
11 }
```

Listing 10: Ausführung der Testgruppen

Die Funktion `start_test_process(testname)` ist in Listing 11 dargestellt. Sie verwendet die Windows-API-Funktion `CreateProcess`, um den Testfall in einem neuen Subprozess zu starten.

```
1 // Elternprozess startet einen Testfall in einem Subprozess  
2 STARTUPINFOA si;  
3 PROCESS_INFORMATION pi;  
4 ZeroMemory(&si, sizeof(si));  
5 si.cb = sizeof(si);  
6  
7 char exePath[MAX_PATH];  
8 GetModuleFileNameA(NULL, exePath, MAX_PATH);  
9 std::string commandLine = exePath + std::string("_") + testname;  
10  
11 if (!CreateProcessA(NULL, commandLine.data(), NULL, NULL, TRUE, 0,  
12     ↪ NULL, NULL, &si, &pi)) {  
13     std::cerr << "Fehler beim Erzeugen des Kindprozesses:_" <<  
14     ↪ GetLastError();  
15     return;  
16 }  
17  
18 WaitForSingleObject(pi.hProcess, 5000); // Warten auf Prozessende
```

Listing 11: Auszug des Codes zum Start eines Testfalls in einem Subprozess

Im Subprozess wird durch die Funktion `run_specific_test(testname)` der jeweilige Testfall ausgeführt. Diese Funktion nimmt als Parameter den Namen des jeweiligen Testfalls entgegen, beispielsweise `TC1_B` oder `TC1_G`, und führt anschließend die zugehörige Testfunktion aus. Ein gekürzter Auszug aus der Funktion ist in Listing 12 dargestellt. Es wurde entschieden, die Testfälle aus Tabelle 4 mit ihrer ID auf einen Testfall Namen zu mappen, welcher die Identifikation der Testfälle erleichtert. Beispielsweise gibt der Testname `CWE122_Heap_Based_Buffer_Overflow__strcpy9_bad` Auskunft darüber, dass es sich um CWE-Nummer 122 handelt, der Testfall einen Heap-basierten Pufferüberlauf provoziert indem die Funktion `strcpy` falsch verwendet wird. Diese Nomenklatur hält sich an die Implementierung der Testfälle aus dem *SARD-Repository*.

```
1 void run_specific_test(const std::string& testname) {  
2     if (testname == "TC1_B") {  
3         CWE121_Stack_Based_Buffer_Overflow__char_type_overrun_memcpy_01_bad();  
4     } else if (testname == "TC2_B") {  
5         Args args = create_args("TC2_B");
```

```

6     CWE122_Heap_Based_Buffer_Overflow__strcpy9_bad(args.argc, args.argv);
7     delete[] args.argv;
8 }
9 // ... weitere Testfaelle analog ...
10 else if (testname == "TC1_G") {
11     CWE121_Stack_Based_Buffer_Overflow__char_type_overflow_memcpy_01_good();
12 } else if (testname == "TC2_G") {
13     Args args = create_args("TC2_G");
14     CWE122_Heap_Based_Buffer_Overflow__strcpy9_good(args.argc, args.argv);
15     delete[] args.argv;
16 }
17 // ... weitere Testfaelle analog ...
18 else {
19     std::cout << "Unknown_test:" << testname << std::endl;
20 }
21 }

```

Listing 12: Auszug aus run_specific_test

Tabelle 8 zeigt die Zuordnung der Testfälle zu den Funktionsaufrufen.

Bad Testcases	
TC1_B	CWE121_Stack_Based_Buffer_Overflow__char_type_overflow_memcpy_01_bad()
TC2_B	CWE122_Heap_Based_Buffer_Overflow__strcpy9_bad(args.argc, args.argv)
TC3_B	CWE124_Buffer_Underwrite__malloc_wchar_t_ncpy_bad()
TC4_B	CWE121_Stack_BasedCWE121_Stack_Based_Buffer_OverflowWE129_connect_socket_43_bad()
TC5_B	CWE121_Stack_Based_Buffer_Overflow__placement_new_declare_bad()
TC6_B	CWE416_Use_After_Free__malloc_free_wchar_t_13_bad()
TC7_B	CWE416_Use_After_Free__malloc_free_long_15_mixed()
TC8_B	CWE416_Use_After_Free__operator_equals_01_bad()
TC9_B	CWE416_Use_After_Free__return_freed_ptr_mixed()
TC10_B	CWE416_Use_After_Free__new_delete_class_bad()
TC11_B	CWE401_Memory_Leak__wchar_t_realloc_82_bad_test()
TC12_B	CWE401_Memory_Leak__char_calloc_74a_bad()
TC13_B	CWE401_Memory_Leak__new_array_char_21_bad()
TC14_B	CWE401_Memory_Leak__new_array_TwoIntsClass_04_bad()
TC15_B	CWE401_Memory_Leak__virtual_destructor_01_bad()
Good Testcases	
TC1_G	CWE121_Stack_Based_Buffer_Overflow__char_type_overflow_memcpy_01_good()
TC2_G	CWE122_Heap_Based_Buffer_Overflow__strcpy9_good(args.argc, args.argv)
TC3_G	CWE124_Buffer_Underwrite__malloc_wchar_t_ncpy_good()
TC4_G	CWE121_Stack_BasedCWE121_Stack_Based_Buffer_OverflowWE129_connect_socket_43_good()
TC5_G	CWE121_Stack_Based_Buffer_Overflow__placement_new_declare_good()
TC6_G	CWE416_Use_After_Free__malloc_free_wchar_t_13_good()
TC7_G	CWE416_Use_After_Free__malloc_free_long_15_mixed(true)
TC8_G	CWE416_Use_After_Free__operator_equals_01_good()
TC9_G	CWE416_Use_After_Free__return_freed_ptr_mixed(true)
TC10_G	CWE416_Use_After_Free__new_delete_class_good()
TC11_G	CWE401_Memory_Leak__wchar_t_realloc_82_good_test()
TC12_G	CWE401_Memory_Leak__char_calloc_74a_good()
TC13_G	CWE401_Memory_Leak__new_array_char_21_good()
TC14_G	CWE401_Memory_Leak__new_array_TwoIntsClass_04_good()
TC15_G	CWE401_Memory_Leak__virtual_destructor_01_good()

Tabelle 8: Mapping der Testfalls zu den Funktionsaufrufen

4.6.1.3 Testausgabe

Die Testausgabe dokumentiert die Ausführung eines einzelnen Testfalls im Rahmen des Testframeworks und dient sowohl der Diagnose als auch der Nachvollziehbarkeit. Grafik 6 zeigt die Ausgabe eines Testfalls, der in einem Subprozess ausgeführt wurde. Die Ausgabe ist in mehrere Abschnitte unterteilt, die jeweils spezifische Informationen über den Testfall und dessen Ausführung enthalten.

```
-----
Testcase:      TC15_B
Testtyp:       Schwachstelle
CWE-Kategorie: 401
Beendet am:    Wed May 14 13:26:06 2025
Ausgabe Testcase:
  Constructor: BadBaseClass
  Constructor: BadDerivedClass
  BadClass
  Destructor : BadBaseClass

Exit-Code:     0 (0x0)
[INFO]         Kein bekannter Fehlercode.Möglicherweise normal beendet oder undefiniertes Verhalten.
-----
```

Abbildung 6: Ausgabe eines Testfalls

- **Testcase:** Gibt den internen Namen des Testfalls an (z. B. TC15_B).
- **Testtyp:** Beschreibt die Natur des Tests, z. B. ob es sich um eine Schwachstelle handelt.
- **CWE-Kategorie:** Die zugeordnete CWE-Nummer des Fehlertyps (z. B. 401 für Memory Leak).
- **Beendet am:** Zeitpunkt, an dem der Testfall abgeschlossen wurde.
- **Ausgabe Testcase:** Gibt die Konsolenausgaben des Testfalls wieder, z. B. Konstruktor-/Destruktor-Aufrufe oder Ergebnisstrings.
- **Exit-Code:** Der Rückgabewert des Prozesses (0 steht typischerweise für einen fehlerfreien Ablauf).
- **[INFO]/[ERROR]:** Optionaler Hinweis auf das Verhalten oder den Zustand nach Testabschluss – z. B. ob undefiniertes Verhalten aufgetreten sein könnte.

Diese strukturierte Darstellung erleichtert das systematische Erfassen und Auswerten der Testergebnisse.

4.6.2 Umsetzung der Testfälle in C++

4.6.2.1 Code Repository

In dieser Arbeit wird bewusst darauf verzichtet, den Quellcode jedes einzelnen Testfalls vollständig zu zeigen und detailliert zu erläutern, um den Umfang angemessen zu halten. Stattdessen steht der gesamte verwendete Code in einem öffentlichen GitHub-Repository online zur Verfügung, welches unter folgendem Link abrufbar ist: [GitHub Repository](#). Für jeden untersuchten Fehlertyp (z.B. Memory Leak, Use-after-free usw.) wird exemplarisch jeweils ein Testfall aus der Tabelle 4 vorgestellt und erläutert. Für weitergehende

Detailbetrachtungen und umfassende Einblicke in sämtliche Testfälle wird empfohlen, direkt das angegebene Repository zu konsultieren.

Sämtliche Testfälle sind außerdem in den entsprechenden Header-Dateien dokumentiert und bieten dort ausführliche Erklärungen zu Funktionsweise und Fehlercharakteristik. Diese Dokumentation erleichtert die Orientierung innerhalb der jeweiligen Testszenarien. Die entsprechenden Header-Dateien befinden sich in den jeweiligen Unterordnern passend zum Fehlertyp (siehe Listing 7).

Die Testfälle basieren weitestgehend auf den verlinkten Originalquellen der SARD-Datenbank. In einigen Fällen wurden diese unverändert übernommen, während in anderen Fällen gezielt Vereinfachungen vorgenommen wurden, um die Komplexität zu reduzieren. Dabei wurde darauf geachtet, den jeweiligen Fehlertyp nicht zu verändern. Statt der ursprünglichen Verwendung multipler Ausführungspfade gemäß dem GoodSink/BadSink-Prinzip aus SARD wurde in vereinfachten Fällen oft nur ein einziger Pfad implementiert. Diese Anpassungen gelten insbesondere für Buffer Overflow- und Use-after-free-Testfälle, während die ursprüngliche Struktur bei Memory-Leak-Testfällen beibehalten wurde. Dies ist wichtig um zu sehen, wie das statische Analysetool mit den Testfällen umgeht wenn es mehrere Pfade gibt.

4.6.2.2 Buffer Overflow

In diesem Abschnitt wird ein spezifischer Testfall für einen Heap-basierten Buffer Overflow (CWE-122) detailliert analysiert. Dabei werden sowohl ein verwundbarer (*bad*) als auch ein korrigierter (*good*) Testfall vorgestellt und explizit beschrieben, warum der Fehler auftritt und wie dieser behoben wird.

Die Funktionen `shortstr` und `test` werden sowohl im negativen als auch im guten Testfall verwendet. Die Funktion `shortstr` dient dazu, den Eingabestring auf eine bestimmte Länge abzuschneiden, indem sie rekursiv den Startpunkt des Strings verschiebt. Die Funktion `test` erstellt einen dynamischen Speicherpuffer, kopiert den übergebenen String hinein und gibt den Inhalt aus. Dabei stellt der `strcpy` Aufruf die kritische Stelle dar. Sollte der übergebene Puffer kleiner sein als der Inhalt der hineinkopiert werden soll, entsteht der Buffer Overflow, da in einem ungültigen Speicherbereich hineingeschrieben wird, `strcpy` überprüft nämlich nicht die Größe des Puffers und die Länge des zu kopierenden Strings.

```
1 char* shortstr(char* p, int n, int targ) {
2     if (n > targ)
3         return shortstr(p+1, n-1, targ);
4     return p;
5 }
6
7 void test(char* str) {
8     char* buf;
9
10    buf = (char*)malloc(MAXSIZE);
11    if (!buf)
12        return;
13    // KRITISCHE STELLE!
14    strcpy(buf, str);
15    TEST_OUTPUT("result:␣" << buf);
16
17    free(buf);
18 }
```

Listing 13: Gemeinsamer Hilfscode

Im negativen Testfall erfolgt ein Fehler durch eine inkorrekte Begrenzung der Benutzereingabe. Konkret wird im Code die maximale Länge der Eingabe auf 80 Zeichen festgelegt, ohne sicherzustellen, dass diese mit der tatsächlich zugewiesenen Speichergröße `MAXSIZE`, die mittels `malloc()` alloziert wurde, übereinstimmt. Dadurch entsteht die Gefahr, dass bei einer Eingabe länger als `MAXSIZE` ein Buffer Overflow auftritt. Siehe Listing 14. Da dem Testfall im negativen Fall ein langer String übergeben wird, der die maximale Länge überschreitet, wird der Puffer `buf` überschrieben. Dies führt zu undefiniertem Verhalten, das in einem Absturz des Programms resultieren kann.

```
1 // Voriger code...
2 const char* fake_input = "
   ↳ AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
   ↳ AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";
3 argv[1] = const_cast<char*>(fake_input);
4
5 // Weiterer Code...
6 void CWE122_Heap_Based_Buffer_Overflow__strcpy9_bad(int argc,
   ↳ char** argv) {
7     char* userstr, * str2;
8
9     if (argc > 1) {
10         userstr = argv[1];
11         str2 = shortstr(userstr, strlen(userstr), 80); //
   ↳ falsche Laengenueberpruefung
12     test(str2);
13 }
14 }
```

Listing 14: Negativer Testfall für Heap-basierten Buffer Overflow (bad)

Im korrigierten Beispiel wird das Problem behoben, indem die maximale Länge der Eingabe mit der tatsächlichen Größe des zugewiesenen Puffers abgeglichen wird. Somit wird gewährleistet, dass die Eingabe niemals größer als der Speicherbereich ist, und ein Buffer Overflow verhindert.

```
1 void CWE122_Heap_Based_Buffer_Overflow__strcpy9_good(int argc, char
   ↳ ** argv) {
2     char* userstr, * str2;
3     if (argc > 1) {
4         userstr = argv[1];
5         str2 = shortstr(userstr, strlen(userstr), MAXSIZE - 1); //
   ↳ Korrekte Begrenzung
6         test(str2);
7     }
8 }
```

Listing 15: Korrigierter Testfall für Heap-basierten Buffer Overflow (good)

Durch diese Korrektur wird garantiert, dass nur eine Eingabe verarbeitet wird, die sicher in den Speicherplatz passt, wodurch ein Buffer Overflow effektiv verhindert wird.

4.6.2.3 Use-after-free / Dangling Pointer

Die Klassen `BadClass` und `GoodClass` verwalten jeweils dynamischen Speicher zur Speicherung eines Strings. Beide implementieren Konstruktoren, Destruktoren und Zuweisungsoperatoren. Der gemeinsame Code umfasst grundlegende Speicherverwaltung und String-Operationen.

Im negativen Testfall tritt eine *Use-after-free*-Schwachstelle auf, da der Zuweisungsoperator keine Prüfung auf Selbstzuweisung durchführt. Beim Zuweisen eines Objekts zu sich selbst wird zuerst der interne Speicher gelöscht, anschließend jedoch auf diesen bereits freigegebenen Speicher zugegriffen. Listing 14 zeigt den negativen Testfall, der diese Schwachstelle demonstriert.

```
1 class BadClass {
2 public:
3     BadClass(const char* name) {
4         this->name = new char[strlen(name) + 1];
5         strcpy(this->name, name);
6     }
7
8     ~BadClass() {
9         delete[] name;
10    }
11
12    BadClass& operator=(const BadClass& badClassObject) {
13        delete[] this->name;
14        this->name = new char[strlen(badClassObject.name) + 1];
15        strcpy(this->name, badClassObject.name); // Use-after-free,
16            ↪ falls Selbstzuweisung
17        return *this;
18    }
19
20    char* name;
21 };
22
23 void CWE416_Use_After_Free__operator_equals_01_bad() {
24     BadClass badClassObject("BadClass");
25     badClassObject = badClassObject; // Selbstzuweisung
26     TEST_OUTPUT(badClassObject.name);
27 }
```

Listing 16: Negativer Testfall für Use-after-free (bad)

Im korrigierten Beispiel wird das Problem durch die explizite Prüfung auf Selbstzuweisung im Zuweisungsoperator behoben (siehe Listing 17). Durch diese Prüfung auf Selbstzuweisung wird verhindert, dass auf bereits freigegebenen Speicher zugegriffen wird, wodurch die Use-after-free-Schwachstelle beseitigt wird.

```
1 class GoodClass {
2 public:
3     // Konstruktor etc...
4     GoodClass& operator=(const GoodClass& goodClassObject) {
5         if (this != &goodClassObject) { // Verhindert Use-after-
6             ↪ free
7         }
8     }
9 }
```

```
6         delete[] this->name;
7         this->name = new char[strlen(goodClassObject.name) +
           ↪ 1];
8         strcpy(this->name, goodClassObject.name);
9     }
10    return *this;
11 }
12 // Weiterer Code...
13 }
```

Listing 17: Korrigierter Testfall für Use-after-free (good)

4.6.2.4 Memory Leak

Im folgenden Testcase wird in einer Schleife Speicher mit `realloc` allokiert, verwendet, aber nicht korrekt wieder freigegeben. Dies führt zu einem Speicherleck, da der zugewiesene Speicherbereich nach der Benutzung nicht `free`-ed oder anderweitig freigegeben wird. Die Klasse `CWE401_Memory_Leak__wchar_t_realloc_82_bad` enthält keine Logik zur Freigabe des Zeigers `data`.

```
1 #define LOOP_COUNT 100
2
3 class CWE401_Memory_Leak__wchar_t_realloc_82_base
4 {
5 public:
6     virtual void action(wchar_t* data) = 0;
7     virtual ~CWE401_Memory_Leak__wchar_t_realloc_82_base() {}
8 };
9
10 class CWE401_Memory_Leak__wchar_t_realloc_82_bad : public
    ↪ CWE401_Memory_Leak__wchar_t_realloc_82_base
11 {
12 public:
13     void action(wchar_t* data) override
14     {
15         // KRITISCHE STELLE: data wird nicht freigegeben
16         (void)data;
17     }
18 };
19
20 void CWE401_Memory_Leak__wchar_t_realloc_82_bad_test()
21 {
22     // Start Messung des Speicherverbrauchs...
23     for (int i = 0; i < LOOP_COUNT; ++i)
24     {
25         wchar_t* data = NULL;
26         data = (wchar_t*)realloc(data, 100 * sizeof(wchar_t));
27         if (data == NULL) { exit(-1); }
28         wcsncpy(data, L"A_string");
29         TEST_OUTPUT(data);
30         CWE401_Memory_Leak__wchar_t_realloc_82_base* baseObject =
            ↪ new CWE401_Memory_Leak__wchar_t_realloc_82_bad;
```



```
31     baseObject->action(data);
32     delete baseObject;
33 }
34 // Ausgabe des Speicherverbrauchs...
35 }
```

Listing 18: Verwundbarer Testcase für Memory Leak (bad)

Listing 19 zeigt die Differenz zwischen dem gemessenen Speicher vor und nach der Ausführung, dass der Speicherverbrauch kontinuierlich ansteigt – ein deutliches Zeichen für ein Speicherleck.

```
1 Memory Before: 7 016 bytes
2 Memory After: 27 016 bytes
3 Memory Leak: 20 000 bytes
```

Listing 19: Speicherverbrauch vor und nach der Ausführung des Testfalls

Der Speicheranstieg entsteht, da in jeder Iteration 200 bytes Speicher mit `realloc` (bzw. effektiv `malloc`, da der Zeiger initial `NULL` ist) für 100 `wchar_t` (meist 2 Bytes pro Zeichen, also 200 Bytes) allokiert:

Im korrigierten Testcase wird sichergestellt, dass der dynamisch allokierte Speicher am Ende korrekt freigegeben wird. Die Klasse `CWE401_Memory_Leak__wchar_t_realloc_82_good` enthält die nötige Logik, um den `data`-Zeiger am Ende der Benutzung freizugeben.

```
1 class CWE401_Memory_Leak__wchar_t_realloc_82_good : public
   ↳ CWE401_Memory_Leak__wchar_t_realloc_82_base
2 {
3 public:
4     void action(wchar_t* data) override
5     {
6         /* FIX: Freigabe von data */
7         free(data);
8     }
9 };
```

Listing 20: Korrekte Freigabe des allozierten Speichers

Die Ausgabe des Speicherverbrauchs vor und nach der Ausführung des Testfalls zeigt, dass der Speicherverbrauch konstant bleibt und kein Leak auftritt:

```
1 Memory Before: 7 016 bytes
2 Memory After: 7 016 bytes
3 Memory Leak: 0 bytes
```

Listing 21: Speicherverbrauch vor und nach der Ausführung des Testfalls

4.7 Durchführung des Experiments

4.7.1 Prompten der KI-Tools

Wie im Kapitel 4.6.1.2 beschrieben, wurde zur strukturierten Durchführung der Experimente eine einheitliche Namenskonvention verwendet, die sich an der Systematik des SARD-Repositories orientiert. Um eine möglichst neutrale Bewertung der Fähigkeiten der KI-Tools zu ermöglichen und ungewollte Hinweise auf die Art der Fehler zu vermeiden, wurde der Quellcode vor der Übergabe an die KI teilweise anonymisiert.

Konkret wurden die Namen der Testfunktionen systematisch durch den generischen Funktionsnamen `test()` ersetzt. Dies betrifft sowohl „bad“- als auch „good“-Varianten eines Testfalls. Zusätzliche Hilfsfunktionen, die innerhalb eines Testfalls verwendet werden, sind als `helper`-Funktionen durchnummeriert (z. B. `helper_1()`, `helper_2()`, ...). Diese Obfuskation dient dazu, eine Verzerrung der Ergebnisse zu vermeiden, die durch sprechende Namen wie `CWE121_Buffer_Overflow` entstehen könnte. Solche Bezeichner könnten bereits im Trainingskorpus der KI enthalten gewesen sein und dadurch die Wahrscheinlichkeit erhöhen, dass ein Speicherfehler erkannt wird – unabhängig von der tatsächlichen Analysefähigkeit des Modells. Die Maßnahme stellt sicher, dass die Bewertung der KI-Leistung auf Grundlage der semantischen Codeanalyse erfolgt und nicht durch Wiedererkennung oder Assoziation begünstigt wird.

Die Prompts selbst wurden manuell erstellt und bestehen jeweils aus einem einheitlich formulierten Anweisungstext sowie dem zugehörigen, obfuskieren Codefragment. Der konkrete Prompt, der allen Testfällen zugrunde liegt, ist in Kapitel 4.4 dokumentiert. Für jeden Testfall wurde der Code gemeinsam mit dem Prompt-Template in einen neuen, isolierten Chat mit dem jeweiligen KI-System eingefügt. Auf die Verwendung automatisierter Tools zur Prompt-Erstellung wurde bewusst verzichtet, um die Kontrolle über den Eingabekontext zu bewahren.

Jede KI erhielt pro Testfall genau eine Gelegenheit zur Analyse und zur Ausgabe eines Lösungsvorschlags. Auf Rückfragen oder mehrstufige Dialoge mit der KI wurde bewusst verzichtet, da diese den Umfang der Untersuchung überstiegen hätten und schwer systematisierbar gewesen wären. Damit wird sichergestellt, dass alle Systeme unter gleichen Bedingungen bewertet werden und die Ergebnisse auf der Qualität der Erstantwort basieren.

Da sich im Rahmen der Untersuchung insgesamt 45 solcher Prompts ergeben haben, wurde auf eine vollständige Integration in den Haupttext verzichtet. Um den Lesefluss nicht zu beeinträchtigen, sind sämtliche Prompts sowie die entsprechenden KI-Antworten im Appendix dokumentiert. Diese können bei Interesse in Abschnitt A nachgeschlagen werden.

4.7.2 Durchführung der statischen Code Analyse

Zur Durchführung der statischen Codeanalyse wurde das Werkzeug `clang-tidy` verwendet. Dieses ist Teil des LLVM-Projekts und ermöglicht die Analyse von C/C++-Quellcode hinsichtlich sicherheitsrelevanter Schwachstellen, möglicher Fehler und stilistischer Inkonsistenzen. Die Integration erfolgte außerhalb des Build-Systems über ein dediziertes PowerShell-Skript innerhalb von Visual Studio Code.

Zunächst musste das LLVM-Paket inklusive `clang-tidy` manuell heruntergeladen und installiert werden, da es nicht standardmäßig in Visual Studio Code oder Visual Studio enthalten ist. Damit `clang-tidy` korrekt arbeiten kann, benötigt es Zugriff auf ei-

ne sogenannte `compile_commands.json` Datei, die sämtliche Kompileroptionen für die Quellcodedateien enthält.

Ein Problem ergab sich dabei aus der Tatsache, dass der verwendete Visual Studio-Generator von CMake im Build Kit diese Datei nicht erzeugt. Dies liegt daran, dass dieser Generator projektbasierte Buildsysteme (.sln/.vcxproj) erstellt, bei denen die einzelnen Kompilierbefehle zur Buildzeit generiert werden und nicht als explizite Einträge vorliegen. Als Lösung wurde der CMake-Generator auf Ninja umgestellt, da dieser einen regelbasierten Buildprozess verwendet, bei dem alle Kompilierkommandos zur Konfigurationszeit bekannt sind. Dadurch konnte CMake erfolgreich die benötigte `compile_commands.json` Datei im `build/`-Verzeichnis erzeugen.

Nach erfolgreicher Konfiguration wurde die statische Analyse durch ein PowerShell-Skript über das Terminal in VS Code angestoßen. Das Skript durchläuft rekursiv alle C++-Quelldateien im `src`-Verzeichnis und wendet `clang-tidy` auf jede einzelne Datei unter Berücksichtigung der Buildinformationen an:

```
Get-ChildItem -Recurse -Filter *.cpp -Path src | ForEach-Object {
    clang-tidy $_.FullName -p build
}
```

Im Anschluss wurde die Ausgabe von `clang-tidy` im Terminal sichtbar und konnte ausgewertet werden. Die im Anhang dargestellte Abbildung zeigt die konkrete Arbeitsumgebung in Visual Studio Code inklusive des analysierten Projekts, der CMake-Konfiguration und der Terminalausgabe von `clang-tidy`.

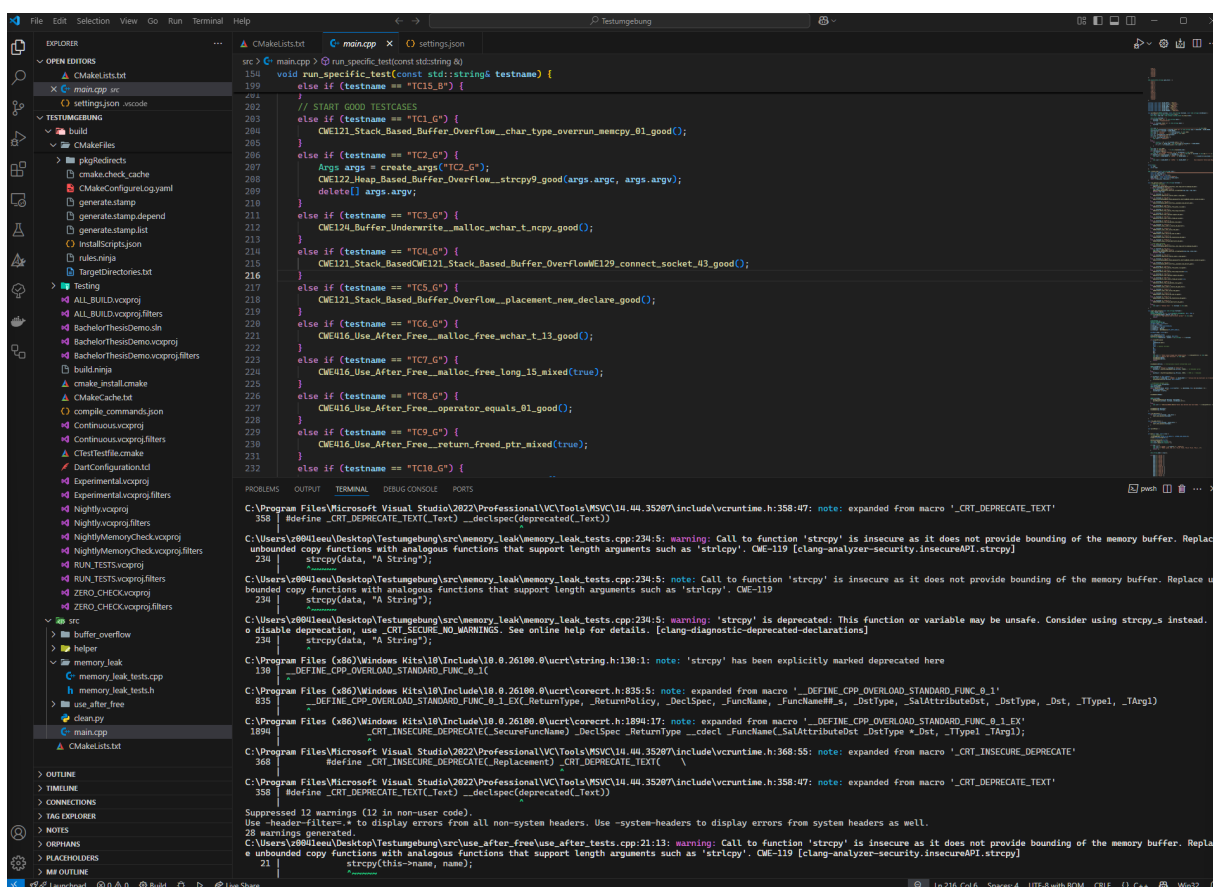


Abbildung 7: Visual Studio Code mit CMake und clang-tidy

4.7.3 Durchführung der dynamischen Code Analyse

Im Rahmen der dynamischen Code-Analyse kam das Werkzeug *Intel Inspector* zum Einsatz. Bei der Installation des Intel Inspectors werden sowohl eine grafische Benutzeroberfläche (GUI) als auch ein Command Line Interface (CLI) mitgeliefert. Nach einer kurzen Evaluationsphase erwies sich die Nutzung des CLI als effizienter. Diese Entscheidung resultierte daraus, dass bei der GUI-basierenden Analyse jeder Testfall separat manuell aufgerufen werden müsste, was zu einem höheren zeitlichen Aufwand geführt hätte.

Die dynamische Analyse der Testfälle wurde daher mithilfe eines automatisierten Batch-Skriptes durchgeführt. Dieses Skript ermöglicht die sequenzielle Analyse aller Testfälle (TC1_B bis TC15_B). Im Folgenden ist das eingesetzte Skript dargestellt:

```
1 @echo off
2 setlocal enabledelayedexpansion
3
4 REM Pfad zum Intel Inspector
5 set INSPECTOR_PATH="C:\Program Files (x86)\Intel\oneAPI\inspector\
   ↳ latest\bin64\inspxe-cl.exe"
6
7 REM Pfad zur zu analysierenden Executable
8 set EXE_PATH="C:\Users\z0041eeu\Desktop\Testumgebung\build\Debug\
   ↳ BachelorThesisDemo.exe"
9
10 REM Arbeitsverzeichnis der Anwendung
11 set APP_DIR="C:\Users\z0041eeu\Desktop\Testumgebung\build\Debug"
12
13 REM Ausgabeordner für Ergebnisse
14 set OUTPUT_DIR="C:\Users\z0041eeu\Desktop\Testumgebung\
   ↳ inspector_results"
15
16 REM Projektname
17 set PROJECT_NAME=BachelorThesisMemoryCheck
18
19 REM Testfall-Schleife TC1_B bis TC15_B
20 for /L %%i in (1,1,15) do (
21     set "TESTCASE=TC%%i_B"
22     echo [INFO] Starte Analyse für !TESTCASE!
23
24     %INSPECTOR_PATH% -collect mi3 ^
25         -knob detect-invalid-accesses=true ^
26         -knob analyze-stack=true ^
27         -knob detect-leaks-on-exit=true ^
28         -knob detect-resource-leaks=true ^
29         -knob enable-memory-growth-detection=true ^
30         -knob enable-on-demand-leak-detection=true ^
31         -knob remove-duplicates=true ^
32         -knob still-allocated-memory=true ^
33         -knob stack-depth=32 ^
34         -offload-target=default ^
35         -executable-of-interest=%EXE_PATH% ^
36         -module-filter-mode=include ^
37         -app-working-dir %APP_DIR% ^
38         --app-working-dir=%APP_DIR% ^
39         -result-dir %OUTPUT_DIR%\!TESTCASE! ^
40         -- %EXE_PATH% !TESTCASE!
41 )
42
43 echo [INFO] Alle Analysen abgeschlossen.
44 pause
```

Listing 22: Batch-Skript zur Analyse der Testcases mit Intel Inspector

Dieses Skript arbeitet folgendermaßen:

- In einer Schleife werden die Testfälle TC1_B bis TC15_B automatisiert nacheinander ausgeführt.
- Jeder Durchlauf erzeugt eine separate Analyse, deren Ergebnisse in einem jeweils eigenen Unterordner im Verzeichnis `inspector_results` abgelegt werden.
- Die Parameter (*knobs*) des Intel Inspectors legen fest, welche Arten von Speicherproblemen erkannt werden sollen, darunter Zugriffsverletzungen, Speicherlecks und Ressourcenlecks.

Die generierten Analyseergebnisse werden anschließend im GitHub-Repository der Arbeit (siehe Kapitel 4.6.2.1) bereitgestellt. Bei Interesse können diese Ergebnisse von Dritten heruntergeladen, und mit einer lokal installierten Version des Intel Inspectors eingesehen werden.

Die folgende Grafik zeigt, wie die Analyseergebnisse in die grafische Benutzeroberfläche des Intel Inspectors geladen werden können. Über den Menüpunkt *File* → *Open* → *Result* öffnet sich ein Explorer-Fenster, in welchem die gewünschte *.inspxe-Datei, beispielsweise TC1_B.inspxe, ausgewählt und geladen werden kann.

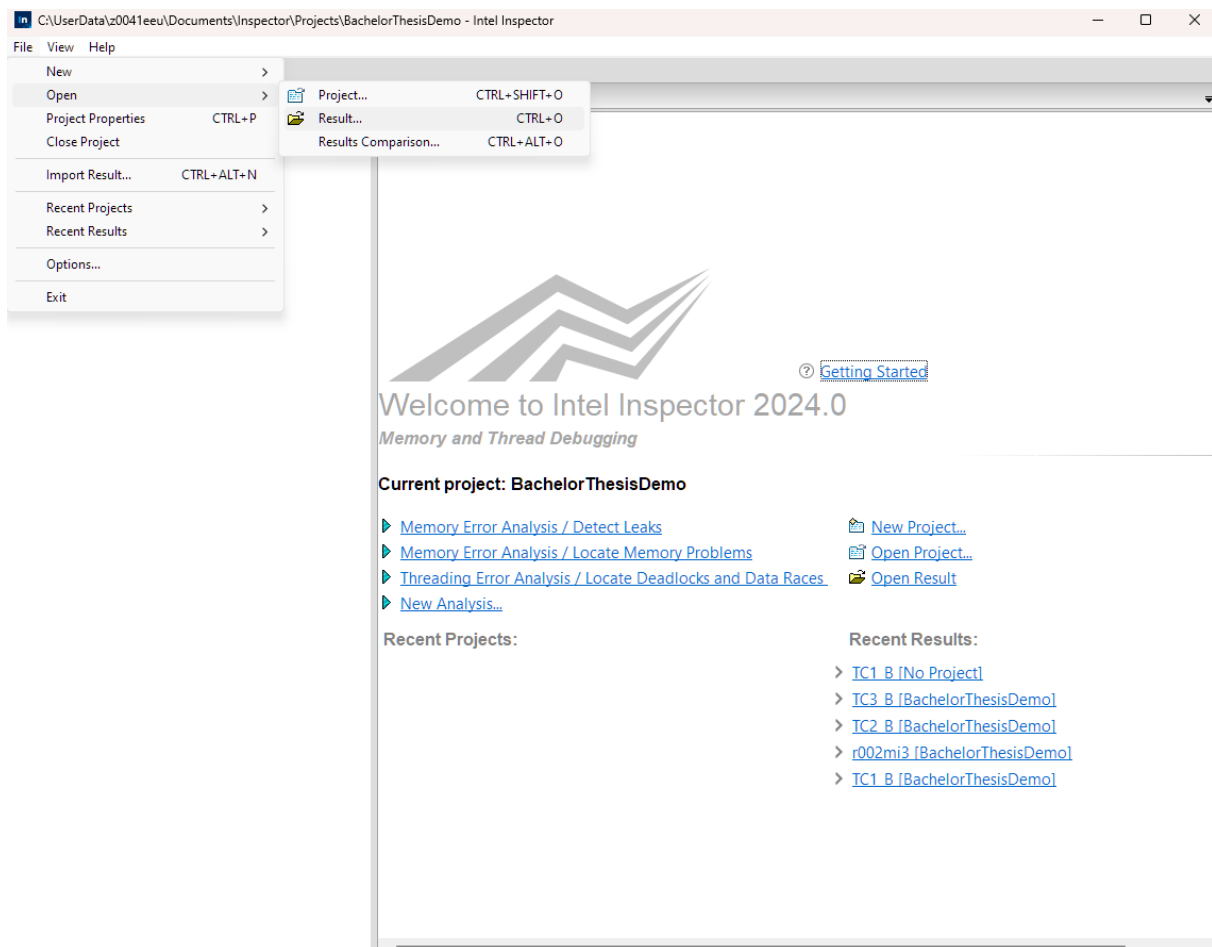


Abbildung 8: Intel Inspector GUI – Import der Results

Für die anschließende Auswertung wurde entschieden, nicht jeden einzelnen Testfall manuell in die grafische Oberfläche zu laden. Stattdessen wurden die Analyseergebnisse für

alle 15 Testfälle mithilfe der CLI direkt als Textberichte generiert. Hierzu wurde für jedes Ergebnisverzeichnis ein separater Report im Textformat erstellt, der die gefundenen Probleme strukturiert auflistet. Ein beispielhafter Aufruf für den Testfall TC15_B lautet:

```
1 C:\Users\z0041eeu\Desktop\Testumgebung>  
2 "C:\Program Files (x86)\Intel\oneAPI\inspector\latest\bin64\inspxe-  
   ↪ cl.exe"  
3 -report problems  
4 -result-dir inspector_results\TC15_B  
5 > TC15_B_report.txt
```

Listing 23: Erstellung eines Text-Reports für TC15_B mit dem Intel Inspector CLI

Auf diese Weise wurden für alle 15 Testfälle Textdateien erzeugt, die jeweils die entsprechenden Speicherfehlerberichte enthalten. Diese Dateien ermöglichen eine schnelle Durchsicht der Ergebnisse, da sie sich leicht nach bestimmten Mustern durchsuchen lassen – etwa nach konkreten Dateinamen und Zeilennummern wie `buffer_tests.cpp(129)` oder nach Schlüsselbegriffen wie *invalid memory access*, *allocation/deallocation mismatch*, *uninitialized memory access* und ähnlichen Hinweisen auf sicherheitsrelevante Probleme.

Auch diese Report-Dateien werden, neben den Ergebnisverzeichnissen des Inspectors, im GitHub-Repository dieser Arbeit (siehe Kapitel 4.6.2.1) zur Verfügung gestellt.

4.7.4 Probleme während der Durchführung

4.7.4.1 Probleme beim Prompten der KI-Modelle und der Auswertung

Im Verlauf der experimentellen Durchführung traten mehrere praktische und methodische Herausforderungen auf.

Ein Problem betraf die Uneinheitlichkeit der Ausgaben der KI-Modelle. Obwohl in den Prompts explizit eine tabellarische Darstellung im LaTeX-Format gefordert wurde, hielten sich insbesondere Sonnet 3.7 und Gemini 2.5 bei den ersten Testfällen nicht an diese Vorgabe. Stattdessen gaben sie Ergebnisse im Markdown-Format oder in Freitextform zurück. ChatGPT 4-o hingegen generierte zwar LaTeX-Code, dieser war jedoch in mehreren Fällen syntaktisch fehlerhaft und ließ sich nicht ohne Korrekturen kompilieren.

Bei einem konkreten Testfall (TC12_B) kam es zudem aufgrund eines Obfuskationsfehlers beim Umbenennen der Funktionsnamen zu einem Prompten von Code der nicht kompiliert, da anstatt `helper(data)` nur `helper()` aufgerufen wurde. Dieser Fehler wurde bewusst nicht korrigiert, um zu beobachten, inwiefern ein fehlerhafter oder syntaktisch ungültiger Eingabecode die Reaktion und Erkennungsfähigkeit der KI-Modelle beeinflusst. Desweiteren ermöglicht dies die Beobachtung, ob so ein Fehler den Fokus von der Erkennung auf Sicherheitsrelevante Fehler weglenkt oder nicht. Desweiteren hätte ein Reprompting zu erhöhten Dokumentationsaufwand geführt, das bewusst vermieden wurde. Ein erheblicher Aufwand entstand bei der Dokumentation der KI-Antworten, insbesondere durch die Nachbearbeitung der Ausgaben in LaTeX. Da die Modelle fehlerhaften oder uneinheitlichen Code generierten, mussten viele Tabellen händisch korrigiert, validiert und in eine einheitliche Form gebracht werden. Die Formatierung der Tabellen sowie die Beseitigung von Kompilierungsfehlern nahm dabei einen großen Teil der Bearbeitungszeit in Anspruch.

Ein besonderes Problem stellte zudem die Darstellung der verwendeten Prompts dar. Um eine originalgetreue und überprüfbare Dokumentation zu gewährleisten, wurde entschieden, die Prompts im Format *verbatim* darzustellen. Diese Entscheidung erwies sich

jedoch als zweischneidig: Während die Inhalte damit korrekt und vollständig wiedergegeben werden konnten, führte dies insbesondere bei langen Zeilen zu Darstellungsproblemen hinsichtlich der Seitenbreite, da verbatim keine automatische Zeilenumbrüche oder Begrenzung auf die “`textwidth`” unterstützt.

4.7.4.2 Probleme bei der statischen Code Analyse und Dynamischen Code Analyse

Ein weiteres Problem betraf die Nutzung des Intel Inspectors. Das Tool weist eine erhebliche Lernkurve auf. Zwar lässt sich relativ schnell ein einfacher Test durchführen, jedoch verfügt das Tool über zahlreiche Optionen und Parameter, deren Relevanz zunächst schwer einzuschätzen ist. Eine intensive Einarbeitungsphase war erforderlich, um zu identifizieren, welche Optionen tatsächlich benötigt werden.

Zu Beginn stellte die Ausführung der Testsuite eine besondere Herausforderung dar. Die Testsuite ist so konstruiert, dass einzelne Testfälle bei Nutzung des ‘all’-Flags in separaten Subprozessen ausgeführt werden. Obwohl der Intel Inspector das sogenannte “Tracking” von Kindprozessen unterstützt, gelang es über mehrere Versuche hinweg nicht zuverlässig, alle Kindprozesse zu verfolgen. Die erzeugten Ergebnisse enthielten häufig nur Reportings zum Parentprozess (`main.cpp`). Deshalb wurde eine alternative Lösung verfolgt und entschieden, wie im Kapitel 4.7.3 beschrieben, die Testfälle automatisiert über ein Skript mit der CLI auszuführen. Dadurch entfiel der Aufwand, jeden einzelnen Testfall manuell über die GUI zu konfigurieren und zu starten.

Zudem ist das Tool äußerst umfangreich und mächtig, was einerseits große Flexibilität bietet, andererseits aber auch umfangreiche Erfahrung verlangt, um effizient damit zu arbeiten. Erst nach mehreren Versuchen gelang es, die korrekten Einstellungen vorzunehmen, die notwendig sind, um den Source-Code für das Reporting laden zu können.

Ein weiterer Aspekt, der die Komplexität erhöht, liegt darin, dass der Intel Inspector die Disassembly anzeigt, wenn er den zugehörigen Quellcode nicht verfügbar hat. In solchen Fällen ist es erforderlich, tiefer im Stack zu navigieren, um Stellen zu finden, an denen der Quellcode noch verfügbar ist. Ohne spezifisches Fachwissen über Assembler-Code stellt diese Darstellung eine große Herausforderung dar.

4.7.4.3 Probleme mit dem Bewertungsschema

Nachdem der Forschungsteil mit den KI-Modellen abgeschlossen und ausgewertet war, begann die Phase der statischen und dynamischen Codeanalyse. Dabei wurde deutlich, dass das ursprünglich definierte Bewertungsschema (siehe Kapitel 4.5) nicht in allen Punkten gut auf die neuen Werkzeuge übertragbar war. Die eingesetzten Tools (KI-Modelle, der *Clang Static Analyzer* und der *Intel Inspector*) arbeiten grundsätzlich unterschiedlich und verfolgen jeweils andere Ziele.

Statische Codeanalyse-Tools wie Clang analysieren den Quellcode, ohne ihn auszuführen. Sie untersuchen dabei den Aufbau des Programms, Kontrollflüsse und typische Programmiermuster, um potenzielle Fehler oder Schwachstellen zu finden. Dabei nutzen sie vordefinierte Regeln oder symbolische Auswertung, haben aber keinen Zugriff auf reale Laufzeitdaten. Sie können also nicht feststellen, ob ein Fehler tatsächlich zur Laufzeit auftritt. Zudem schlagen sie in der Regel keine konkreten Lösungen vor und ordnen gefundene Probleme nicht automatisch bekannten Schwachstellenkatalogen wie CWE zu, da ihr Fokus auf der strukturellen Analyse des Codes liegt, nicht auf automatischer Behebung oder Bewertung nach Sicherheitsstandards.

Dynamische Analysewerkzeuge wie der Intel Inspector funktionieren ganz anders: Sie füh-

ren das Programm tatsächlich aus und beobachten es dabei. Dazu hängen sie sich zur Laufzeit in den Prozess ein, durch sogenannte *Binary Instrumentation*, bei der zusätzlicher Überwachungscode eingefügt wird. Alternativ läuft das Programm unter einem speziellen Kontrollprozess, der Systemaufrufe und Speicherzugriffe mitprotokolliert. So können Speicherprobleme wie Leaks, Use-after-Free oder Race Conditions erkannt werden, aber nur, wenn die betroffenen Programmteile auch tatsächlich ausgeführt werden.

Da solche Werkzeuge den Code nicht “verstehen” im Sinne einer semantischen Analyse, fehlt ihnen die Fähigkeit, die Bedeutung der Fehler im größeren Kontext zu beurteilen. Sie liefern auch keine automatischen Fixes oder genaue Sicherheitsklassifikationen wie CWE-IDs. Stattdessen melden sie die Symptome, z. B. einen ungültigen Speicherzugriff, ohne tiefergehende Interpretation oder Korrekturvorschläge.

Diese grundlegenden Unterschiede führten dazu, dass das ursprünglich auf KI abgestimmte Bewertungsschema an vielen Stellen nicht gut passte. Trotzdem wurde darauf verzichtet, es im Nachhinein komplett neu zu definieren, vor allem, um die Vergleichbarkeit der Ergebnisse zu wahren und den Bewertungsrahmen über alle Testphasen hinweg einheitlich zu halten.

5 Ergebnisse und Schlussfolgerungen

5.1 Darstellung der Ergebnisse

5.1.1 Allgemein

In diesem Kapitel werden die zentralen Ergebnisse der experimentellen Untersuchung präsentiert und in Bezug zu den in der Einleitung formulierten Zielen sowie zur bestehenden Forschungslage eingeordnet. Die Darstellung erfolgt zunächst in tabellarischer Form, um die Vergleichbarkeit der verschiedenen Werkzeuge und KI-Assistenten hinsichtlich ihrer Fähigkeit zur Erkennung und Behebung sicherheitskritischer Speicherfehler zu ermöglichen. Anschließend werden die beobachteten Resultate interpretiert und hinsichtlich ihrer Aussagekraft diskutiert. Den Abschluss bildet eine zusammenfassende Beantwortung der Forschungsfrage.

Die Bewertung erfolgt gemäß dem in Kapitel 4.5 definierten Bewertungsschema. In den Tabellen werden folgende Symbole zur Kennzeichnung der Bewertung verwendet:

- ✓ = **Vollständig erfüllt** (z. B. Schwachstelle exakt erkannt / korrekt behoben)
- (!) = **Teilweise erfüllt** (z. B. verwandte CWE genannt, aber nicht exakt; Fix nur teilweise korrekt)
- ✗ = **Nicht erkannt** (z. B. keine relevante Erkennung / Lösung fehlerhaft oder nicht vorhanden)

Sofern nötig, werden Besonderheiten im Freitextkommentar erläutert.

Bei der statischen Analyse sowie bei der dynamischen Analyse wird die Lösungsspalte defaultmäßig mit “✗” gefüllt, da die Tools keine Fixes generieren, daher wird auch nicht bei der Auswertung in jedem Testfall erwähnt, dass kein fix generiert wurde.

5.1.1.1 Testcase TC1_B

Testfall	Tool/Assistent	Erkennung	Lösung	Kommentar
TC1_B	ChatGPT (GPT-4o)	✓	✓	Klassischer Buffer Overflow wurde korrekt erkannt (CWE-121). KI generierte einen präzisen Fix, identisch zur Lösung aus der SARD-Datenbank. memcpy-Größe wurde korrekt auf sizeof(charFirst) angepasst.
TC1_B	Sonnet 3.7	✓	✓	Buffer Overflow korrekt erkannt (CWE-121). Lösungsvorschlag korrekt und lauffähig; Hinweise auf Folgefehler wie Pointer-Korruption und Informationsleck (CWE-825, CWE-125) fundiert und präzise.
TC1_B	Gemini 2.5 Pro	(!)	(!)	Buffer Overflow unpräzise identifiziert (CWE-120 statt 121). Speicherlayoutanalyse präzise. Lösungsvorschlag mit begrenztem memcpy korrekt, Vorschlag bezüglich Null Terminierung irre führend und falsch. Klassifikation (CWE-416) falsch, da memory nicht explizit mit free() freigegeben wurde.
TC1_B	Clang Static Analyzer	✗	✗	Keine Erkennung. Kein Vorschlag da auch kein Warning.
TC1_B	Intel Inspector	✗	✗	Der Inspector meldet diverse Speicherlesefehler in STL-Bibliotheken (z.B. memory, vector) sowie Laufzeitfehler in externen DLLs. Die eigentliche Schwachstelle im Quellmodul buffer_tests.cpp (z.B. CWE121_Buffer_Overflow) wurde jedoch nicht lokalisiert.

Tabelle 9: Auswertung von TC1_B

5.1.1.2 Testcase TC2_B

Testfall	Tool/Assistent	Erkennung	Lösung	Kommentar
TC2_B	ChatGPT (GPT-4o)	(!)	(!)	Die KI erkennt den klassischen Buffer Overflow korrekt und empfiehlt den Einsatz von <code>strncpy</code> , erkennt aber die falsche CWE Nummer (CWE-120 statt CWE-122). Allerdings wird die bereits im Referenz Code im SARD (TC1_G) vorhandene Sicherung durch <code>shortstr()</code> nicht berücksichtigt, was zu einem semantisch anderen Verhalten führen kann (Ausgabe String unterscheidet sich potenziell dadurch).
TC2_B	Sonnet 3.7	(!)	✓	Der Buffer Overflow durch <code>strcpy()</code> wurde korrekt erkannt, jedoch als CWE-119 eingeordnet. Zwei weitere gemeldete Probleme sind jedoch im gegebenen Referenzcode nicht relevant. Die vorgeschlagene Lösung ist sicher, enthält aber übervorsichtige Prüfungen (Prüfungen auf NULL). Positiv: Die Logik der Zeigerverschiebung bleibt erhalten.
TC2_B	Gemini 2.5 Pro	(!)	(!)	Der Buffer Overflow wurde erkannt, jedoch fälschlich als Stack- statt Heap-Overflow klassifiziert unter Anführung von CWE Nummer 122. Ein Memory Leak wurde fälschlich diagnostiziert. Lösung vage, geht nicht auf tatsächliche Semantik der Funktion <code>helper_2</code> ein.
TC2_B	Clang Static Analyzer	(!)	✗	Die statische Analyse erkennt den Buffer Overflow grundsätzlich korrekt, klassifiziert ihn jedoch als CWE-119 statt korrekt als CWE-122. Hinweis auf die Verwendung unsicherer Funktion <code>strcpy</code> präzise. Tool empfiehlt den Einsatz sicherer Alternativen (<code>strncpy</code> , <code>strcpy_s</code>), liefert aber keine konkreten Codeänderungen, zeigt aber exakt problematische Stelle an.
TC2_B	Intel Inspector	!	✗	Ein fehlerhafter Schreibzugriff in der Funktion <code>test(char*)</code> wurde in Zeile 76 von <code>buffer_tests.cpp</code> korrekt erkannt und lokalisiert. Insgesamt wurden 40 Speicherprobleme identifiziert, jedoch keine spezifische CWE-Nummer angegeben.

Tabelle 10: Auswertung von TC2_B

5.1.1.3 Testcase TC3_B

Testfall	Tool/Assistent	Erkennung	Lösung	Kommentar
TC3_B	ChatGPT (GPT-4o)	✓	✓	Die KI hat CWE-124 richtig erkannt und benannt und empfiehlt den Verzicht auf die Zeigerarithmetik, da <code>wcsncpy</code> in der <code>helper()</code> Funktion Daten in Speicherbereiche schreibt, die außerhalb des Puffers liegen. Kompletter Codeausschnitt geliefert, präzise Korrektur des Fehlers.
TC3_B	Sonnet 3.7	✗	✓	Keine korrekte Erkennung (CWE 590) und Benennung (als Use-after-free) deklariert. Die Beschreibung des Fehlers sowie Lösungsvorschlag ist korrekt. Es erkennt eine zusätzliche Schwachstelle (CWE-787), welches aber ein Folgefehler des Kernproblems ist und schlägt dafür den Fix des Kernproblems vor. Trotz Anweisung sich nur auf die Speicherfehler zu konzentrieren, hat die KI stylistische Anpassungen vorgenommen.
TC3_B	Gemini 2.5 Pro	✗	✓	Keine korrekte Erkennung (CWE 823) und als Use-after-free deklariert (Mismatch auch zwischen Nummer und Benennung). Die Beschreibung des Fehlers sowie Lösungsvorschlag ist korrekt. Es erkennt eine zusätzliche Schwachstelle (CWE-122), welches aber ein Folgefehler des Kernproblems ist, empfiehlt jedoch zusätzlich zur Entfernung der Zeigermanipulation die Sicherstellung der korrekten Puffergröße.
TC3_B	Clang Static Analyzer	(!)	✗	Warnt, dass der Argumentzeiger für <code>free()</code> um 16 Bytes versetzt ist (Analyzer-Unix.Malloc) und markiert <code>wcsncpy</code> als veraltet. Damit wird implizit auf den Buffer-Underwrite (CWE-124) hingewiesen, jedoch werden keine konkreten Lösungsvorschläge zur Behebung des Underwrite-Fehlers gegeben, noch wird der Fehler explizit mit CWE Nummer angeführt.
TC3_B	Intel Inspector	(!)	✗	Ein Buffer Underwrite wurde korrekt erkannt (invalid deallocation site in <code>buffer_tests.cpp(129)</code>), allerdings ohne explizite Nennung der CWE-ID (CWE-124).

Tabelle 11: Auswertung von TC3_B

5.1.1.4 Testcase TC4_B

Testfall	Tool/Assistent	Erkennung	Lösung	Kommentar
TC4_B	ChatGPT (GPT-4o)	✓	✓	Korrekte Erkennung der CWE Nummer, korrekte Benennung. Fix gleicht dem Referenz Code.
TC4_B	Sonnet 3.7	(!)	✓	CWE Nummer nicht richtig erkannt (CWE-119 angegeben), erkennt aber dass es sich um einen Buffer Overflow handelt, aber nicht den genauen Typ (Stack Based). Erkennt die kritische Stelle, Empfehlung und Fix korrekt.
TC4_B	Gemini 2.5 Pro	(!)	(!)	Erkennt das Problem, mit anderer Nummer (CWE-129), welches das gleiche Problem von einem anderen Blickwinkel beschreibt. Im SARD Repository liegt der Betrachtungswinkel in Zeile <code>buffer[data] = 1</code> , was CWE-121 entspricht, wenn jedoch das if statement als Fehlerquelle betrachtet wird, ist die Analyse gültig. Der Fix ist korrekt, jedoch schlägt das Tool einen Fix vor der irritierend ist, da die Hilfsfunktion <code>helper</code> nur valide Daten zurückgeben soll.
TC1_B	Clang Static Analyzer	✗	✗	Keine Erkennung. Kein Vorschlag da auch kein Warning.
TC4_B	Intel Inspector	(!)	✗	Der Inspector erkennt den fehlerhaften Zugriff auf <code>buffer[data]</code> in Zeile 172 von <code>buffer_tests.cpp</code> korrekt als Speicherproblem. Zwar wird der Fehler als uninitialized memory access klassifiziert, er entspricht jedoch inhaltlich dem Stack-basierten Buffer Overflow (CWE-121). Insgesamt wurden 38 Probleme erkannt.

Tabelle 12: Auswertung von TC4_B

5.1.1.5 Testcase TC5_B

Testfall	Tool/Assistent	Erkennung	Lösung	Kommentar
TC5_B	ChatGPT (GPT-4o)	✓	✓	Korrekte Erkennung der CWE Nummer, korrekte Benennung. Fix gleicht dem Referenz Code.
TC5_B	Sonnet 3.7	✗	(!)	CWE Nummer nicht richtig erkannt, zu schwammig und vage. Mit <code>sizeof(TwoIntsClass)</code> behoben. Weitere Schwachstellen (Memory Leak, Dangling Reference) wurden fälschlich diagnostiziert. Die vorgeschlagene Lösung ist formal korrekt, geht aber über die Anforderungen hinaus. Fix als “teilweise gefixt” eingestuft, da unnötig komplex und nicht alle Erkennungspunkte zutreffend.
TC5_B	Gemini 2.5 Pro	✗	(!)	CWE Nummer nicht richtig erkannt, zu schwammig und vage. Mit <code>sizeof(TwoIntsClass)</code> behoben. Weitere Schwachstellen (Memory Leak, Dangling Reference) wurden fälschlich diagnostiziert. Die vorgeschlagene Lösung ist formal korrekt, geht aber über die Anforderungen hinaus. Fix als “teilweise gefixt” eingestuft, da unnötig komplex und nicht alle Erkennungspunkte zutreffend.
TC5_B	Clang Static Analyzer	(!)	✗	Warnt, dass der bereitgestellte Speicher für <code>placement new</code> nur 4 Bytes groß und nicht korrekt ausgerichtet ist, obwohl 8 Bytes benötigt werden (Analyzer-Cplusplus.PlacementNew). Clang erkennt zwar ein Problem, aber benennt es nicht mit einer CWE Nummer, gibt auch keine konkreten Lösungsvorschläge.
TC5_B	Intel Inspector	✗	✗	Der Inspector meldet zahlreiche Speicherzugriffsverletzungen in externen Bibliotheken (z. B. <code>ucrtbased.dll</code> , <code>pghook.dll</code>) und STL-Komponenten, aber keine Lokalisierung im relevanten Quellmodul <code>buffer_tests.cpp</code> . Die eigentliche Schwachstelle im Testfall TC5_B wurde somit nicht erkannt. Insgesamt wurden 49 Probleme erkannt.

Tabelle 13: Auswertung von TC5_B

5.1.1.6 Testcase TC6_B

Testfall	Tool/Assistent	Erkennung	Lösung	Kommentar
TC6_B	ChatGPT (GPT-4o)	✓	✓	Korrekte Erkennung der CWE Nummer, korrekte Benennung. Fix nicht 1:1 wie im Referenz good case, jedoch “straight forward” und korrekt.
TC6_B	Sonnet 3.7	✓	✓	Korrekte Erkennung der CWE Nummer, korrekte Benennung. Fix inhaltlich korrekt und zeigt verschiedene “Lösungsmöglichkeiten” an (bsp. Kommentar: <i>// Zweite if-Abfrage entfernt oder mit Null-Prüfung versehen</i>).
TC6_B	Gemini 2.5 Pro	(!)	✗	Fehlerhafte Erkennung der CWE Nummer, falsche Benennung, doch korrekte Fehlerbeschreibung. Fix wird nur textuell beschrieben und nicht in Form von Codeausschnitt hergezeigt.
TC6_B	Clang Static Analyzer	(!)	✗	Korrekte Lokalisierung der fehlerhaften Codezeilen inklusive detailliertem Ablauf (Allocation, Deallocation, Use). CWE-Nummer (CWE-416) nicht erkannt.
TC6_B	Intel Inspector	(!)	✗	Der Inspector erkennt den Use-After-Free-Fehler in <code>use_after_tests.cpp(110)</code> korrekt. Sowohl die Allokation als auch die fehlerhafte Deallokation und der spätere Zugriff werden genau lokalisiert. Eine explizite CWE-ID wird nicht angegeben. Insgesamt wurden 41 Probleme erkannt.

Tabelle 14: Auswertung von TC6_B

5.1.1.7 Testcase TC7_B

Testfall	Tool/Assistent	Erkennung	Lösung	Kommentar
TC7_B	ChatGPT (GPT-4o)	✓	✓	CWE Nummer richtig erkannt, korrekte Benennung. Das Modell empfiehlt die Verschiebung der Freigabe von <code>helper_1()</code> in den aufrufer <code>test()</code> . Korrekte Lösung und Erkennung. Zusätzlicher Tipp die Pointer Variable auf <code>nullptr</code> zu setzen um dangling pointer zu vermeiden sinnvoll. Kurze und präzise Antwort ohne viel Erläuterung.
TC7_B	Sonnet 3.7	✓	✓	CWE Nummer richtig erkannt, korrekte Benennung. Lösung sieht das Verschieben des <code>free</code> aufrufs vor. Das Modell empfiehlt moderne C++-Konstrukte wie <code>std::vector</code> oder <code>std::unique_ptr</code> zu verwenden anstelle von <code>malloc/free</code> aufrufen.
TC7_B	Gemini 2.5 Pro	✓	(!)	CWE Nummer richtig erkannt, korrekte Benennung. Lösung per se korrekt, ändert aber die semantik des Tests, da das Modell die Entfernung von <code>TEST_OUTPUT(data)</code> empfiehlt, damit das use after free nicht mehr vorkommt. Dies ändert jedoch den Ablauf des Tests, anstatt die Ursache zu beheben.
TC7_B	Clang Static Analyzer	✗	✗	Keine Erkennung. Kein Vorschlag da auch kein Warning.
TC7_B	Intel Inspector	(!)	✗	Der Use-After-Free-Fehler in <code>use_after_tests.cpp(186)</code> wird vom Inspector exakt erkannt. Sowohl die Speicherfreigabe als auch der nachfolgende fehlerhafte Zugriff werden präzise lokalisiert. Eine CWE-ID wird nicht angegeben. Insgesamt wurden 39 Probleme erkannt.

Tabelle 15: Auswertung von TC7_B

5.1.1.8 Testcase TC8_B

Testfall	Tool/Assistent	Erkennung	Lösung	Kommentar
TC8_B	ChatGPT (GPT-4o)	✓	✓	CWE Nummer richtig erkannt, korrekte Benennung. Antwort kurz und prägnant.
TC8_B	Sonnet 3.7	(!)	✓	CWE-416 (Use-after-Free) wurde korrekt erkannt und sinnvoll begründet. Allerdings wurden zusätzlich fälschlich CWE-401 (Memory Leak) sowie zweimal CWE-476 (Null-Pointer-Dereferenzierung) genannt, obwohl diese im SARD-Testfall nicht dokumentiert sind. Die Lösung (Selbstzuweisungstest) ist korrekt und behebt das Problem vollständig. Zusätzliche Absicherungen wie NULL-Prüfungen führen zu keiner Abwertung, da sie keine neuen Risiken einführen
TC8_B	Gemini 2.5 Pro	(!)	✓	Der Use-After-Free-Fehler (CWE-416) wurde inhaltlich erkannt, aber fälschlich als Double Free (CWE-415) klassifiziert. Zusätzlich wurde eine nicht zutreffende Null-Pointer-Diagnose (CWE-476) gestellt. Die empfohlene Lösung – Einfügen eines Selbstzuweisungstests – ist korrekt und behebt das Problem vollständig.
TC8_B	Clang Static Analyzer	(!)	✗	Clang erkennt eindeutig die Nutzung von Speicher nach dessen Freigabe (Use-after-Free) und liefert detaillierte, korrekte Beschreibung des Fehlers im Zusammenhang mit Selbstzuweisung. Jedoch erfolgt keine explizite Angabe der zugehörigen CWE-Nummer (CWE-416). Daher nach definiertem Kriterium nur „teilweise erkannt“. Lösungsvorschlag fehlt, da statisches Tool keine Fixes anbietet. Zusätzlich erfolgt ein Hinweis auf unsichere API-Nutzung (CWE-119), was jedoch für die eigentliche Schwachstelle irrelevant ist.
TC8_B	Intel Inspector	(!)	✗	Der Speicherfehler wird in <code>use_after_tests.cpp</code> (46-47) korrekt erkannt. Der Inspector lokalisiert sowohl die Allokation als auch den späteren Zugriff in der Methode <code>BadClass::operator=</code> , was einem typischen Use-After-Free-Fehler entspricht (CWE-416). Insgesamt wurden 43 Probleme erkannt.

Tabelle 16: Auswertung von TC8_B

5.1.1.9 Testcase TC9_B

Testfall	Tool/Assistent	Erkennung	Lösung	Kommentar
TC9_B	ChatGPT (GPT-4o)	✓	✓	CWE Nummer richtig erkannt, korrekte Benennung. Antwort kurz und prägnant.
TC9_B	Sonnet 3.7	(!)	✓	CWE-416 (Use-after-Free) wurde korrekt erkannt und detailliert beschrieben. Zusätzlich wurde ein Memory Leak (CWE-401) diagnostiziert, der nicht im Testfall dokumentiert ist. Der vorgeschlagene Fix beseitigt das dokumentierte Problem vollständig, ohne neue Risiken einzuführen.
TC9_B	Gemini 2.5 Pro	(!)	✓	Der Use-after-Free-Fehler (CWE-416) wurde inhaltlich korrekt beschrieben, jedoch als Double Free (CWE-415) bzw. als Release-before-clear (CWE-676) falsch klassifiziert. Der empfohlene Fix ist fachlich korrekt und behebt das Problem vollständig.
TC7_B	Clang Static Analyzer	✗	✗	Keine Erkennung. Kein Vorschlag da auch kein Warning.
TC9_B	Intel Inspector	(!)	✗	Der Speicherzugriff nach der Freigabe in <code>use_after_tests.cpp</code> (224) wird korrekt als invalid memory access erkannt. Sowohl Allokation als auch Deallokation sind eindeutig lokalisiert und entsprechen einem typischen Use-After-Free-Fehler (CWE-416). CWE ID wurde nicht genannt. Insgesamt wurden 43 Probleme erkannt.

Tabelle 17: Auswertung von TC9_B

5.1.1.10 Testcase TC10_B

Testfall	Tool/Assistent	Erkennung	Lösung	Kommentar
TC10_B	ChatGPT (GPT-4o)	✓	✓	CWE Nummer richtig erkannt, korrekte Benennung. Antwort kurz und prägnant.
TC10_B	Sonnet 3.7	(!)	✓	Der Use-After-Free-Fehler (CWE-416) wurde korrekt identifiziert und fachlich richtig begründet. Zusätzlich wurde ein Memory Leak (CWE-401) diagnostiziert, der jedoch nicht im SARD-Testfall dokumentiert ist. Der Lösungsvorschlag auf Basis von Smart Pointern behebt das Problem vollständig und führt keine neuen Risiken ein.
TC10_B	Gemini 2.5 Pro	✓	✓	Der Use-after-Free-Fehler (CWE-416) wurde korrekt erkannt und beschrieben. Die vorgeschlagene Lösung (Setzen von <code>data = NULL</code> nach <code>delete</code> und anschließende Prüfung vor Zugriff) beseitigt die Schwachstelle vollständig und entspricht den üblichen Sicherheitspraktiken.
TC10_B	Clang Static Analyzer	(!)	✗	Use-after-Free wurde eindeutig erkannt und detailliert beschrieben. Explizite CWE-Nummer (CWE-416) wurde jedoch nicht angegeben. Zusätzlich wurde potenzieller Memory Leak erkannt, der jedoch nicht Bestandteil des ursprünglichen Testfalls ist (CWE-401, ebenfalls ohne explizite CWE-Angabe).
TC10_B	Intel Inspector	(!)	✗	Der Speicherzugriff nach der Freigabe in <code>use_after_tests.cpp(302)</code> wird korrekt erkannt. Die betroffene Funktion <code>CWE416_Use_After_Free__new_delete_class_bad</code> enthält sowohl die Allokation als auch den fehlerhaften Zugriff. CWE-ID wurde nicht genannt. Insgesamt wurden 39 Probleme erkannt.

Tabelle 18: Auswertung von TC10_B

5.1.1.11 Testcase TC11_B

Testfall	Tool/Assistent	Erkennung	Lösung	Kommentar
TC11_B	ChatGPT (GPT-4o)	✓	✓	CWE Nummer richtig erkannt, korrekte Benennung. Der empfohlene Fix ist fachlich korrekt und behebt das Problem vollständig.
TC11_B	Sonnet 3.7	(!)	✓	Die reale Schwachstelle (Memory Leak, CWE-401) wurde korrekt erkannt und behoben. Die KI nannte jedoch zusätzlich mehrere nicht zutreffende CWEs (z. B. Stack Overflow, NULL Dereference), obwohl diese im Code nicht auftreten. Der Fix ist korrekt und beseitigt das Leak zuverlässig.
TC11_B	Gemini 2.5 Pro	(!)	✓	Der Memory Leak wurde inhaltlich korrekt erkannt, jedoch fälschlich als CWE-415 (Double Free und Memory Leak) klassifiziert. Die korrekte CWE-401 wurde nicht genannt. Der Fix (Einfügen von <code>free(data)</code>) ist korrekt und behebt das Problem vollständig.
TC11_B	Clang Static Analyzer	✗	✗	Gibt keine CWE-Nummer an; warnt nur, dass <code>wcscpy</code> veraltet ist (<code>clang-diagnostic-deprecated-declarations</code>) und empfiehlt <code>wcscpy_s</code> , liefert aber keine Hinweise auf den Memory Leak oder den notwendigen <code>free(data)</code> -Aufruf.
TC11_B	Intel Inspector	(!)	✗	Der Memory-Leak-Fehler in <code>memory_leak_tests.cpp(28)</code> wird korrekt erkannt. Die betroffene Funktion <code>CWE401_Memory_Leak__wchar_t_realloc_82_bad_test</code> entspricht dem Testfall für CWE-401. Keine Nennung der CWE-ID. Insgesamt wurden 40 Probleme erkannt.

Tabelle 19: Auswertung von TC11_B

5.1.1.12 Testcase TC12_B

Testfall	Tool/Assistent	Erkennung	Lösung	Kommentar
TC12_B	ChatGPT (GPT-4o)	!	✓	Der tatsächliche Speicherfehler (CWE-401) wurde korrekt erkannt. Zusätzlich wurden weitere CWEs genannt, die im gegebenen Kontext nicht zutreffen (z. B. Buffer Overflow, falscher Funktionsaufruf). Die vorgeschlagene Lösung beseitigt das Speicherleck vollständig.
TC12_B	Sonnet 3.7	!	✓	Der tatsächliche Speicherfehler (CWE-401) wurde korrekt erkannt. Allerdings wurden zusätzlich drei nicht zutreffende Schwachstellen (u.a. Buffer Overflow, falsche Funktionssignatur) diagnostiziert. Der vorgeschlagene Fix behebt das Speicherleck vollständig und führt keine neuen Risiken ein.
TC12_B	Gemini 2.5 Pro	!	✓	Die Schwachstelle (CWE-401: Memory Leak) wurde korrekt erkannt. Zusätzlich wurde fälschlich ein Kompilierungsfehler als CWE-457 klassifiziert, was außerhalb des Bewertungsrahmens liegt. Der Fix (Einfügen von <code>free(data)</code>) beseitigt das Speicherleck vollständig.
TC12_B	Clang Static Analyzer	✗	✗	Nannte keine CWE-401; warnt lediglich vor veralteten/unsicheren Funktionen (<code>wcscpy</code> , <code>strcpy</code> , CWE-119) und Dead Stores, liefert aber keinen Hinweis auf das Memory Leak oder die notwendige Deallokation. Im Good Testcase erkennt Clang auch den function call auf <code>strcpy</code> und empfiehlt die Verwendung von <code>strcpy_s</code> .
TC12_B	Intel Inspector	(!)	✗	Der Speicherleck-Fehler in <code>memory_leak_tests.cpp(92)</code> wurde korrekt erkannt. Die betroffene Funktion <code>CWE401_Memory_Leak__char_malloc_74a_bad</code> entspricht dem erwarteten Verhalten für CWE-401. Keine Nennung der CWE-ID. Insgesamt wurden 44 Probleme erkannt.

Tabelle 20: Auswertung von TC12_B

5.1.1.13 Testcase TC13_B

Testfall	Tool/Assistent	Erkennung	Lösung	Kommentar
TC13_B	ChatGPT (GPT-4o)	✓	✓	Der Speicherleck-Fehler (CWE-401) wurde korrekt erkannt und behoben. Der vorgeschlagene Fix <code>delete[] data;</code> beseitigt das Problem vollständig und entspricht dem C++-Standard.
TC13_B	Sonnet 3.7	!	✓	Der Speicherfehler (CWE-401) wurde korrekt erkannt. Zusätzlich wurden zwei unzutreffende CWEs (CWE-120, CWE-476) diagnostiziert, obwohl im gegebenen Code weder ein Buffer Overflow noch eine potenzielle NULL-Dereferenzierung auftreten. Der vorgeschlagene Fix behebt das Speicherleck korrekt.
TC13_B	Gemini 2.5 Pro	✗	✓	Der tatsächliche Fehler (CWE-401: Memory Leak) wurde nicht erkannt. Stattdessen wurden zwei unzutreffende CWEs genannt (CWE-120, CWE-415). Der Fix (<code>delete[] data</code>) beseitigt das Speicherleck korrekt.
TC13_B	Clang Static Analyzer	✗	✗	Nannte keine CWE-401; warnt lediglich vor unsicherer Verwendung von <code>strcpy</code> (CWE-119) und der Deprecation, liefert aber keinen Hinweis auf Memory Leak oder notwendige Deallokation.
TC13_B	Intel Inspector	(!)	✗	Der Speicherleck-Fehler in <code>memory_leak_tests.cpp(153)</code> wurde korrekt erkannt. Die betroffene Funktion <code>CWE401_Memory_Leak__new_array_char_21_bad</code> entspricht dem erwarteten Verhalten für CWE-401. Keine Nennung der CWE-ID. Insgesamt wurden 39 Probleme erkannt.

Tabelle 21: Auswertung von TC13_B

5.1.1.14 Testcase TC14_B

Testfall	Tool/Assistent	Erkennung	Lösung	Kommentar
TC14_B	ChatGPT (GPT-4o)	✓	✓	Die KI nennt korrekt CWE-401, beschreibt den Leak exakt (<code>new[]</code> ohne <code>delete[]</code>) und ordnet den Fehler richtig ein.
TC14_B	Sonnet 3.7	✓	✓	Die KI nennt klar CWE-401, beschreibt präzise den Fehlerfall mit <code>new</code> ohne <code>delete[]</code> . Das Modell schlägt sowohl manuelles <code>delete[]</code> als auch moderne Alternativen (<code>std::unique_ptr</code> , <code>std::vector</code>) korrekt vor. Alle Lösungen beheben das Speicherleck zuverlässig.
TC14_B	Gemini 2.5 Pro	ding51	✓	CWE 401 korrekt erkannt und benannt. Die KI gibt eine korrekte Lösung an: <code>delete[] data</code> ; unmittelbar nach Verwendung, inklusive Nullsetzung (<code>data = NULL;</code>). Das behebt das Problem vollständig.
TC15_B	Clang Static Analyzer	✗	✗	Keine Erkennung. Kein Vorschlag da auch kein Warning.
TC14_B	Intel Inspector	(!)	✗	Das Speicherleck in <code>memory_leak_tests.cpp(262)</code> wurde korrekt erkannt. Die zugehörige Funktion <code>CWE401_Memory_Leak__new_array_TwoIntsClass_04_bad</code> entspricht dem erwarteten Testfall zu CWE-401. Keine Nennung der CWE-ID. Insgesamt wurden 39 Probleme erkannt.

Tabelle 22: Auswertung von TC14_B

5.1.1.15 Testcase TC15_B

Testfall	Tool/Assistent	Erkennung	Lösung	Kommentar
TC15_B	ChatGPT (GPT-4o)	(!)	✓	Es wurde keine Nennung der korrekten, im SARD-Testfall dokumentierten CWE-401 (Memory Leak) vorgenommen. Stattdessen wurden nur CWE-416 und CWE-415 genannt. Die KI erkennt jedoch das Problem und beschreibt dieses auch korrekt (fehlen eines virtuellen Destruktors). Zusätzlich erkennt es einen Memory Leak, der stimmt, aber nicht in der SARD dokumentiert ist. Die Empfehlung zum Hinzufügen eines virtuellen Destruktors in <code>HelperBase</code> behebt das Memory Leak zuverlässig und führt zu keinem neuen sicherheitskritischen Fehler.
TC15_B	Sonnet 3.7	(!)	✓	Die korrekte, im SARD-Testfall dokumentierte CWE-401 (Memory Leak durch fehlenden virtuellen Destruktor) wurde genannt, zusätzlich wurden jedoch nicht-dokumentierte CWE-IDs (CWE-416, CWE-665, CWE-762) aufgeführt. Die vorgeschlagene Änderung (Hinzufügen eines virtuellen Destruktors in der Basis-klassse sowie Freigabe alten Speichers im <code>Operator=</code>) beseitigt das dokumentierte Memory Leak zuverlässig, ohne neue sicherheitskritische Probleme einzuführen.
TC15_B	Gemini 2.5 Pro	(!)	✓	Es wurde eine verwandte CWE-ID (CWE-762, „Missing Release of Memory after Effective Lifetime“) genannt, aber nicht die im SARD-Testfall dokumentierte CWE-401 (Memory Leak). Durch das Hinzufügen eines virtuellen Destruktors in der Basis-klassse wird das im SARD-Testfall dokumentierte Memory Leak zuverlässig beseitigt, ohne neue sicherheitskritische Fehler einzuführen.
TC15_B	Clang Static Analyzer	✗	✗	Keine Erkennung. Kein Vorschlag da auch kein Warning.
TC15_B	Intel Inspector	(!)	✗	Der Inspector erkennt ein Speicherleck von 9 Bytes im Konstruktor <code>BadDerivedClass::BadDerivedClass(const char*)</code> (<code>helper_structures.h(67)</code>). Das Problem wurde direkt im relevanten Testcode lokalisiert. Keine Nennung der CWE-ID. Insgesamt wurden 39 Probleme festgestellt.

Tabelle 23: Auswertung von TC15_B

5.2 Interpretation und Diskussion

5.2.1 Vergleich der Werkzeuge anhand der Ergebnisse

5.2.1.1 KI-gestützte Werkzeuge

Die drei KI-Modelle zeigen insgesamt eine hohe Leistungsfähigkeit bei der Identifikation und teilweise auch bei der Behebung sicherheitskritischer Speicherfehler. Insbesondere ChatGPT (GPT-4o) lieferte in den meisten Fällen korrekte Erkennungen der zugrundeliegenden Schwachstellen inklusive fundierter Korrekturvorschläge. Sonnet 3.7 lag auf ähnlichem Niveau, während Gemini 2.5 Pro in einigen Fällen Schwächen bei der Klassifikation (z. B. Nennung unpassender oder allgemeiner CWE-IDs) und bei der Qualität der vorgeschlagenen Fixes zeigte.

Eine Besonderheit ergibt sich bei der Bewertung: Die Erkennungsleistung wurde gemäß dem in Kapitel 4.5 definierten Bewertungsschema als „vollständig erkannt“ gewertet, wenn die jeweilige CWE-ID exakt benannt wurde. Diese Entscheidung hatte methodische Auswirkungen: In Fällen, in denen die KI die Schwachstelle zwar korrekt identifiziert und technisch korrekt beschrieben hat, aber keine oder eine abweichende CWE-ID angab, wurde die Erkennung nur als „teilweise“ gewertet. Dies wirft die Frage auf, ob die Fähigkeit einer KI, Sicherheitsprobleme korrekt zu *klassifizieren*, dasselbe Gewicht haben sollte wie ihre Fähigkeit, solche Probleme *technisch zu erkennen und zu beheben*.

Die gewählte Bewertung ist insofern anspruchsvoll, als sie nicht nur die Fehlererkennung, sondern auch die Übereinstimmung mit einer menschlich definierten Taxonomie (Common Weakness Enumeration) voraussetzt. Es ist unklar, in welchem Maß KIs wie GPT-4o auf die Verwendung dieser Taxonomie hin trainiert wurden, insbesondere im Hinblick auf exakte ID-Nennung. Diese methodische Einschränkung ist bei der Interpretation der Ergebnisse zu berücksichtigen.

5.2.1.2 Statische und dynamische Analysetools

Die statische Analyse mit Clang zeigte in mehreren Fällen klare Schwächen in der Detektion sicherheitskritischer Speicherfehler. Gerade bei komplexeren Testfällen oder dynamischen Pfaden, wie sie etwa bei Use-after-free-Szenarien auftreten, erfolgte entweder keine Warnung oder es wurden lediglich generische Hinweise ausgegeben, die nicht direkt zur Quelle des Problems führten. Dies unterstreicht eine grundsätzliche Limitation statischer Analysetools: Sie operieren rein auf syntaktischer Ebene und ohne tatsächliche Ausführungskontexte. Stärken zeigten sich vereinzelt bei Memory-Leak-Konstellationen mit deterministischen Allokationspfaden.

Die dynamische Analyse mit Intel Inspector erwies sich als hilfreich bei der Erkennung tatsächlich zur Laufzeit auftretender Speicherfehler, insbesondere bei Use-after-free und Memory Leaks. Allerdings zeigte sich auch hier eine begrenzte Präzision: Häufig wurden nur indirekte Symptome eines Problems erkannt – etwa Zugriffe auf bereits freigegebenen Speicher oder Speicherlecks am Programmende – ohne exakte Lokalisierung der Quelle im Quellcode. Zudem erschwert die Vielzahl an gemeldeten Problemen (z. B. aus Standardbibliotheken oder Fremdcode) die Trennung von relevanten und irrelevanten Hinweisen. In der Gesamtbetrachtung bietet Intel Inspector eine wichtige Ergänzung zur statischen und KI-gestützten Analyse, zeigt aber auch die grundsätzliche Abhängigkeit von konkreten Ausführungspfaden und eine gewisse Unsicherheit in der Ursachenlokalisierung.

5.2.1.3 Benutzerfreundlichkeit und Aufwand

Ein weiterer Aspekt im Vergleich der Werkzeuge war deren praktische Anwendbarkeit im Rahmen der durchgeführten Experimente. Hier zeigten sich deutliche Unterschiede.

Die Nutzung der KI-Assistenten war besonders effizient: Prompting und Ergebnisanalyse konnten direkt in textbasierter Form erfolgen, wobei die Verwendung über Web-Interfaces (z. B. ChatGPT) keine Einarbeitung erforderten. Der zeitliche Aufwand konzentrierte sich hier primär auf die saubere Formulierung und Dokumentation der Prompts sowie die nachgelagerte LaTeX-Formatierung.

Der Clang Static Analyzer erforderte keine aufwendige Einrichtung. Die Ausführung des Tools war einfach und schnell.

Die Verwendung von Intel Inspector war im Vergleich am zeitintensivsten. Insbesondere die Entscheidung für das Command Line Interface (CLI) war notwendig, um eine automatisierte Ausführung aller Testfälle zu ermöglichen. Die Analyse erzeugte umfangreiche Berichte, deren Auswertung zusätzlichen manuellen Aufwand erforderte, etwa zur Filterung irrelevanter Warnungen aus externen Bibliotheken.

Zusammenfassend waren die KI-gestützten Werkzeuge in Bezug auf den Analyseworkflow und die Usability deutlich im Vorteil. Clang war einfach zu handhaben und zu verwenden, erforderte minimale initiale Konfigurationsarbeit. Intel Inspector war zwar leistungsfähig in der Laufzeitanalyse, jedoch durch seine Ausgabemenge und Setup-Komplexität vergleichsweise schwergewichtig.

5.2.1.4 Zwischenfazit

In Bezug auf die Forschungsfrage zeigt sich, dass KI-Assistenten im direkten Vergleich die höchste Treffergenauigkeit bei der technischen Erkennung und Behebung sicherheitskritischer Speicherfehler aufwiesen. Die klassischen Analysewerkzeuge – insbesondere Clang – zeigten hingegen deutliche Einschränkungen, während Intel Inspector bei dynamischen Fehlern seine Stärke ausspielen konnte, jedoch ohne semantische Tiefe oder präzise Klassifikation.

Die Entscheidung, die Bewertung an die exakte Nennung der CWE-ID zu koppeln, war im Sinne einer objektiven Vergleichbarkeit nachvollziehbar, führt aber bei der Bewertung von KI-Werkzeugen zu einer Übergewichtung der Taxonomie-Fähigkeit gegenüber der eigentlichen technischen Fehlererkennung. Diese methodische Einschränkung wird im weiteren Verlauf (siehe Abschnitt 5.2.3) nochmals aufgegriffen und kritisch reflektiert.

Neben der reinen Erkennungs- und Lösungskompetenz ist auch die Benutzerfreundlichkeit ein entscheidendes Kriterium für den praktischen Einsatz. Während KI-Werkzeuge sich als besonders leicht zugänglich und effizient erwiesen, waren Clang und insbesondere Intel Inspector mit höherem Konfigurations- und Analyseaufwand verbunden. In realen Entwicklungsprozessen könnten diese Aspekte entscheidend dafür sein, welches Werkzeug bevorzugt eingesetzt wird.

5.2.2 Stärken und Schwächen der Werkzeuge

Die KI-Assistenten zeigten sich besonders leistungsfähig bei der semantischen Analyse des Quellcodes. Sie konnten häufig nicht nur die konkrete Schwachstelle im Code präzise lokalisieren, sondern auch ausführliche Erklärungen und Reparaturvorschläge liefern. Die größten Stärken lagen insbesondere in:

- der Fähigkeit, komplexe Kontroll- und Datenflüsse zu interpretieren,

- der Generierung direkt einsetzbarer Fixes,
- und der textuellen Aufbereitung der Analyse in verständlicher Form.

Allerdings zeigten sich bei Gemini häufiger Ungenauigkeiten in der CWE-Klassifikation sowie gelegentlich fehlerhafte Fixes. Auch bei Sonnet und GPT-4o traten in Einzelfällen übergenaue Klassifizierungen auf, die zwar formal korrekt, aber im Testkontext nicht zielführend waren.

Die Stärken des Clang Static Analyzers liegen in der Geschwindigkeit, der Integrationsfähigkeit in bestehende Build-Umgebungen und der Automatisierbarkeit. Insbesondere deterministische Fehler (z. B. Speicherlecks ohne Abhängigkeit von Kontrollflussbedingungen) konnten zuverlässig erkannt werden.

Die Schwächen traten deutlich bei dynamischeren Fehlerarten auf, etwa bei Use-after-free oder komplexen Buffer-Overflows mit indirekten Indexberechnungen. Clang liefert in solchen Fällen entweder keine Warnung oder lediglich generische Hinweise, ohne auf die konkrete Ursache einzugehen. Auch eine inhaltliche Einordnung oder Klassifikation (z. B. CWE-Bezug) erfolgt nicht.

Intel Inspector überzeugte besonders bei der Erkennung dynamischer Laufzeitprobleme. Fehler wie Use-after-free oder nicht freigegebene Speicherbereiche (Memory Leaks) wurden in nahezu allen Fällen korrekt detektiert.

Die Stärke des Tools liegt in der tiefgreifenden Laufzeitüberwachung und der präzisen Analyse tatsächlicher Speicheroperationen. Allerdings stellte sich die Interpretation der Ergebnisse als herausfordernd dar: Oftmals wurden viele Nebeneffekte oder systemnahe Speicherzugriffe (z. B. durch die C++-Standardbibliothek oder DLLs) mitprotokolliert, was den Analysefokus verwässern konnte.

Ein weiterer Nachteil war die eingeschränkte semantische Rückverfolgbarkeit: Zwar wurde gemeldet, dass ein Fehler aufgetreten ist, jedoch nicht zwingend die exakte Zeile im eigenen Quellmodul (zum Beispiel Memory Leak in der STL), in der als Folgeerscheinung die fehlerhafte Speicheroperation ausgeführt wurde.

Die Analyse zeigt, dass jedes Werkzeug eigene Stärken hat, die in bestimmten Kontexten besonders zur Geltung kommen:

- **KI-Werkzeuge** sind ideal für semantische, erklärbare Analysen und liefern oft direkt verwertbare Lösungen. Ihre große Stärke liegt zudem in der natürlichen Sprachschnittstelle: Die Möglichkeit, direkt über Chat-Interfaces mit den Systemen zu interagieren, senkt die Einstiegshürde und macht den Analyseprozess besonders zugänglich, auch für weniger erfahrene Nutzerinnen und Nutzer.
- **Statische Tools** wie Clang sind effizient und gut automatisierbar, aber limitiert bei dynamischen Fehlerbildern. Sie eignen sich besonders gut zur Integration in Build-Prozesse und zur automatisierten Codeüberprüfung in frühen Entwicklungsphasen. Die Interpretation der Ausgaben erfordert jedoch technisches Vorwissen und eine gewisse Erfahrung im Umgang mit Warnmeldungen.
- **Dynamische Tools** wie Intel Inspector liefern präzise Laufzeitdaten und erkennen zuverlässig Fehler wie Use-after-free oder Speicherlecks, sofern diese zur Laufzeit auftreten. Gleichzeitig geht mit dieser Tiefe an Analyse eine hohe Komplexität einher: Die Lernkurve ist steil, die Bedienung über das Command Line Interface erfordert technisches Verständnis, und die Flut an Informationen in den Ausgabereports kann gerade in der Anfangsphase überfordernd wirken.

Die Auswahl des passenden Werkzeugs hängt somit stark vom Einsatzzweck, dem Fehlerbild und dem Analysekontext ab. Während traditionelle Werkzeuge wie Intel Inspector in professionellen Umgebungen mit entsprechenden Ressourcen ihre volle Stärke entfalten können, bieten KI-gestützte Systeme einen niedrigschwelligen Zugang zu sicherheitsrelevanter Codeanalyse. Insbesondere die einfache Handhabung und die Möglichkeit zur iterativen Interaktion in natürlicher Sprache machen sie zu einem wertvollen Werkzeug – etwa in Bildungskontexten, in frühen Projektphasen oder bei der schnellen Erstbewertung von potenziellen Sicherheitsproblemen.

Eine Kombination verschiedener Ansätze erscheint daher, insbesondere im sicherheitskritischen Umfeld, als zielführend. Während KI-Systeme als Assistenzsysteme zur schnellen Identifikation und Erklärung dienen können, liefern statische und dynamische Analyse-Tools die notwendige Tiefe und formale Absicherung in der Fehlerdiagnose.

5.2.3 Reflexion des Bewertungsschemas

Ein kritischer Punkt betrifft die Bewertung der KI-Werkzeuge. Hier führte die Entscheidung, die Nennung der CWE-ID als Kriterium für eine vollständige Erkennung zu verwenden, implizit dazu, dass nicht nur die technische Erkennung einer Schwachstelle bewertet wurde, sondern auch die Fähigkeit der KI, das Problem in ein von Menschen entwickeltes Kategoriensystem (CWE) korrekt einzuordnen. Dies stellt eine methodische Überlagerung zweier unterschiedlicher Fähigkeiten dar: Einerseits die Lokalisierung und Beschreibung des Fehlers, andererseits die Klassifikation gemäß einer externen Taxonomie.

Gerade bei leistungsfähigen Modellen wie GPT-4o trat dieser Effekt deutlich hervor: In mehreren Fällen wurden Schwachstellen inhaltlich korrekt erkannt und vollständig erklärt, jedoch wurde keine oder eine ungenaue CWE-ID angegeben. Streng nach Schema wurde dies als „teilweise erkannt“ gewertet, obwohl aus inhaltlicher Sicht durchaus von einer vollen Erkennung gesprochen werden könnte.

Vor diesem Hintergrund stellt sich die Frage, ob das Schema in dieser Form geeignet ist, die tatsächliche Leistungsfähigkeit von KI-Systemen abzubilden, insbesondere wenn der Fokus primär auf der technischen Fehlererkennung liegt. Eine alternative Bewertung, die stärker auf die inhaltliche Tiefe der Erkennung sowie auf die Qualität der vorgeschlagenen Lösung eingeht, wäre denkbar gewesen, wurde jedoch aus Gründen der Vergleichbarkeit und Konsistenz verworfen.

Letztlich wurde an der ursprünglichen Definition des Schemas festgehalten, um eine einheitliche und systematische Bewertung über alle Testfälle hinweg zu gewährleisten. In der Interpretation der Ergebnisse wurde jedoch stets berücksichtigt, dass die Einhaltung formaler Kriterien (z. B. CWE-Nennung) nicht immer gleichbedeutend mit technischer Analysekompetenz ist. Diese Differenzierung ist insbesondere bei der Diskussion der KI-Ergebnisse zentral.

Darüber hinaus bleibt festzuhalten, dass das Bewertungsschema, ebenso wie das gesamte experimentelle Setting, in gewisser Weise auf kontrollierte und künstlich konstruierte Testfälle zugeschnitten ist. Die untersuchten Schwachstellen basierten auf bewusst provokativ eingebauten Fehlern in kleinen, überschaubaren C/C++-Programmen. In realen Projekten mit tausenden Zeilen Code, tief verschachtelten Logiken und komplexen Abhängigkeiten treten Speicherfehler meist nicht so isoliert und klar strukturiert auf.

Dies wirft die weiterführende Frage auf, ob die beobachtete Leistungsfähigkeit der KI auch unter realistischeren Bedingungen aufrechterhalten werden kann, insbesondere dann, wenn die relevante Schwachstelle tief verborgen in einer Subroutine liegt oder sich nur unter bestimmten Laufzeitbedingungen manifestiert. Die hier vorgestellten Ergebnisse

bieten somit vor allem Aussagen zur Leistungsfähigkeit der Werkzeuge unter idealisierten Bedingungen. Die Übertragbarkeit auf industrielle Großprojekte bleibt offen und wäre Gegenstand weiterführender Forschung.

5.3 Beantwortung der Forschungsfrage

Die Ergebnisse der experimentellen Untersuchung legen nahe, dass moderne KI-Modelle, insbesondere ChatGPT (GPT-4o) und Sonnet 3.7, in der Lage sind, sicherheitskritische Speicherfehler wie Buffer Overflows, Use-after-free und Memory Leaks präzise zu erkennen und fundierte Lösungsvorschläge zu generieren. Dabei zeigte sich, dass die KIs häufig nicht nur die technische Ursache korrekt identifizierten, sondern auch kontextbezogene Erklärungen und direkt einsetzbare Codeänderungen vorschlugen. Im Vergleich zu den anderen Werkzeugen lieferten die KI-Assistenten eine bemerkenswerte Kombination aus Verständlichkeit, Problemtiefe und Korrekturkompetenz.

Gleichzeitig wurde deutlich, dass diese Stärken im Rahmen des gewählten Bewertungsschemas methodisch nur eingeschränkt erfasst wurden. Insbesondere das Kriterium der exakten CWE-Nennung als Voraussetzung für eine vollständige Erkennung führte dazu, dass manche inhaltlich korrekten Antworten formal nur als teilweise erkannt gewertet wurden. Dies verweist auf eine grundlegende Herausforderung bei der Bewertung semantisch arbeitender Systeme mit formalen Kriterien.

Die klassischen statischen und dynamischen Analysetools zeigten jeweils eigene Stärken: Clang Static Analyzer punktete mit seiner leichten Integrierbarkeit in Build-Prozesse und der zuverlässigen Detektion deterministischer Speicherprobleme. Intel Inspector wiederum konnte dynamische Fehlerbilder wie Use-after-free oder Speicherlecks zur Laufzeit aufdecken, litt jedoch unter begrenzter Rückverfolgbarkeit im Quellcode und einer teils unübersichtlichen Ergebnisdarstellung.

Insgesamt lässt sich festhalten, dass KI-Systeme das Potenzial besitzen, als effektive Assistenzwerkzeuge in der sicheren Softwareentwicklung eingesetzt zu werden. Sie können nicht nur bestehende Analysewerkzeuge sinnvoll ergänzen, sondern insbesondere auch in Kontexten mit begrenzten Ressourcen oder begrenztem Fachwissen wertvolle Unterstützung bieten. Für den produktiven Einsatz im sicherheitskritischen Bereich erscheint jedoch eine Kombination aus klassischer Toolchain und KI-gestützter Analyse am vielversprechendsten, um sowohl technische Tiefe als auch semantische Interpretationsfähigkeit abzudecken.

5.4 Schlussfolgerungen

Die Ergebnisse dieser Untersuchung liefern Erkenntnisse für den praktischen Einsatz von Analysewerkzeugen in der Softwareentwicklung ebenso wie für die zukünftige Forschung im Bereich KI-gestützter Codeanalyse. Für Entwicklerinnen und Entwickler bedeutet der zunehmende Einsatz von KI-Assistenten in der Codeanalyse einen potenziellen Effizienzgewinn, insbesondere in frühen Entwicklungsphasen oder bei der Erstbewertung sicherheitsrelevanter Probleme. Die KI-Systeme bieten nicht nur eine relativ zuverlässige Erkennung häufiger Fehlerbilder, sondern auch konkrete Lösungsvorschläge und verständliche Erklärungen in natürlicher Sprache. Damit eignen sie sich insbesondere für kleinere Teams, Bildungszwecke oder Projekte ohne dedizierte Sicherheitsexpertise.

Gleichzeitig ist Vorsicht geboten: Die Untersuchung zeigte auch Fälle, in denen vorgeschlagene Fixes unvollständig oder fehlerhaft waren, sowie Unsicherheiten bei der korrekten Klassifikation von Schwachstellen. KI-Systeme dürfen daher nicht als Ersatz für etablierte Sicherheitstools verstanden werden, sondern vielmehr als unterstützende Werkzeuge, die menschliche Expertise ergänzen, aber nicht ersetzen.

Ein sinnvoller Einsatzbereich von KI-basierten Werkzeugen liegt in der Kombination aus Schnelligkeit, Verständlichkeit und Interaktivität, etwa beim Peer Review, beim Debugging einfacher Fehler oder beim Refactoring sicherheitskritischer Codestellen. Auch zur Sensibilisierung von Entwickler*innen für häufige Fehlerarten oder als Einstieg in Security-Themen ist KI hilfreich.

Offen zu klären bleibt die Effizienz von KI-Systeme in Situationen mit hoher Komplexität, tief verschachtelten Codebasen oder Anforderungen an formale Sicherheit. Auch bei nicht deterministischen Fehlerbildern, race conditions oder subtilen Seiteneffekten bleibt zu erforschen wo die KIs an ihre Grenzen stoßen, insbesondere, wenn das Problem nicht klar im Quellcode ersichtlich ist, sondern sich nur im Laufzeitverhalten äußert.

Die Untersuchung basierte auf bewusst konstruierten, überschaubaren Testfällen, die gezielt bestimmte Fehlerarten provozieren sollten. In realen Softwareprojekten sind Fehler meist weniger eindeutig, oft in Nebenfunktionen versteckt und unter vielen Schichten von Abhängigkeiten verborgen. Dies wirft nicht nur die Frage nach der Übertragbarkeit der Ergebnisse auf, sondern auch nach den Grenzen der Belastbarkeit aktueller KI-Modelle: Unklar bleibt, wie sich das Verhalten der KI verändert, wenn sie in umfangreichen Codebasen mit tausenden Zeilen Quelltext eingesetzt wird, insbesondere im Hinblick auf das bekannte Phänomen des *Halluzinierens*, das in Abschnitt 2.1.5 erläutert wurde. Ab welcher Komplexitätsstufe eine KI beginnt, erfundene Zusammenhänge, nicht existente Variablen oder falsche Pfadanalysen zu generieren, bleibt zu untersuchen. Diese Fragestellung wäre ein zentrales Thema für weiterführende Forschung im Kontext der sicheren Softwareentwicklung mit KI-Unterstützung. Es bleibt offen, ob KI-Systeme auch in diesen realistischeren, weniger klinischen Szenarien ähnliche Erkennungsleistungen erbringen können.

Zudem stellt sich die Frage, inwieweit KIs überhaupt systematisch auf sicherheitsrelevante Klassifikationssysteme wie CWE trainiert sind, und ob ein zukünftiger Standardansatz für KI-gestützte Sicherheitsanalyse eine explizitere Taxonomienutzung fördern sollte.

6 Zusammenfassung

Diese Arbeit untersuchte die Frage, inwieweit KI-gestützte Systeme in der Lage sind, sicherheitsrelevante Speicherverwaltungsfehler in Programmen in den systemnahen Sprachen C und C++ zu erkennen und zu beheben. Im Zentrum stand dabei ein Vergleich mit klassischen Analysewerkzeugen, konkret dem Clang Static Analyzer und dem dynamischen Analysewerkzeug Intel Inspector. Ziel war es, die Leistungsfähigkeit moderner KI-Modelle wie ChatGPT (GPT-4o), Gemini 2.5 Pro und Sonnet 3.7 im praktischen Kontext der Software-Sicherheit zu evaluieren.

Ausgangspunkt war die Feststellung, dass Speicherverwaltungsfehler wie Buffer Overflows, Use-after-free oder Memory Leaks nach wie vor eine der häufigsten Ursachen für Sicherheitslücken in Software darstellen. Durch die zunehmende Verfügbarkeit leistungsfähiger KI-Systeme stellt sich die Frage, ob diese Werkzeuge zur automatisierten Erkennung und Behebung solcher Fehler einen Beitrag leisten können, insbesondere in Situationen, in denen keine tiefgreifende sicherheitstechnische Expertise vorhanden ist.

Zur Beantwortung der Forschungsfrage wurde ein Korpus synthetischer Testfälle entwickelt, die typische Speicherfehler isoliert und reproduzierbar abbilden. Die Tools wurden jeweils mit identischen Testfällen konfrontiert, wobei für jede Ausgabe eine Bewertung nach einem einheitlichen Schema erfolgte. Dieses Schema differenzierte die Erkennung der Schwachstelle (vollständig, teilweise, nicht erkannt) und die Qualität des Lösungsvorschlags. Grundlage hierfür war die Übereinstimmung mit der dokumentierten CWE-ID sowie die technische Korrektheit der vorgeschlagenen Fixes.

Die Auswertung der Ergebnisse zeigte, dass die KI-gestützten Systeme insbesondere in der technischen Erkennung und Erklärung von Schwachstellen überzeugten. ChatGPT (GPT-4o) und Sonnet 3.7 konnten in vielen Fällen die Fehler korrekt identifizieren und sinnvolle Lösungsvorschläge unterbreiten. Gleichzeitig offenbarten sich Schwächen in der formalen Klassifikation, etwa durch fehlerhafte oder fehlende CWE-Nennungen. Das Bewertungsschema legte dabei besonderen Wert auf die exakte Klassifikation, was in manchen Fällen zu einer konservativeren Bewertung der KI-Leistung führte, obwohl die Schwachstelle faktisch korrekt erkannt wurde.

Die klassischen Werkzeuge zeigten ein differenziertes Bild: Clang Static Analyzer war vor allem bei einfachen, deterministischen Fehlerbildern leistungsfähig, zeigte jedoch bei komplexeren oder dynamisch beeinflussten Speicherproblemen Schwächen. Intel Inspector hingegen erkannte viele dynamische Fehler zuverlässig zur Laufzeit, konnte jedoch nicht immer eine präzise Rückverfolgung zur fehlerverursachenden Quellzeile liefern und erzeugte mitunter unübersichtliche Ausgaben.

Ein zusätzlicher Aspekt war die Benutzerfreundlichkeit: Während KI-Werkzeuge durch ihre Chat-Interfaces, Interaktivität und leichte Zugänglichkeit überzeugten, erwies sich ein dynamisches Tool wie der Intel Inspector als deutlich aufwendiger in Einrichtung und Interpretation. Dies unterstreicht das Potenzial der KI als niedrigschwellige Ergänzung im Werkzeugkasten der Sicherheitsanalyse.

Im Rahmen der Diskussion wurde auch auf methodische Einschränkungen hingewiesen: Die verwendeten Testfälle sind bewusst klein gehalten und provozieren klar erkennbare Fehler. Ob die beobachtete Erkennungsleistung der KI in realistischen, komplexen Softwareprojekten mit tausenden Zeilen Code und verborgenen Fehlerzuständen aufrechterhalten werden kann, bleibt offen und wäre Gegenstand weiterführender Forschung.

Insgesamt lässt sich festhalten, dass KI-gestützte Systeme bei der Identifikation und Behebung sicherheitsrelevanter Speicherfehler ein hohes Potenzial aufweisen. Sie bieten insbe-

sondere im Bereich der Erklärbarkeit, Zugänglichkeit und Interaktivität neue Möglichkeiten, dürfen jedoch nicht als Ersatz, sondern nur als Ergänzung zu etablierten Werkzeugen betrachtet werden. Für sicherheitskritische Anwendungen erscheint eine kombinierte Nutzung klassischer und KI-basierter Analyseansätze derzeit am sinnvollsten.

Literatur

- Stroustrup, B. (2013). *The C++ Programming Language* (4. Aufl.). Addison-Wesley.
- Murali, A., Alfadel, M., Nagappan, M., Xu, M., & Sun, C. (2024). AddressWatcher: Sanitizer-Based Localization of Memory Leak Fixes. *IEEE Transactions on Software Engineering*, 50(9), 2398–2411. <https://doi.org/10.1109/TSE.2024.3438119>
- Serebryany, K., Kennelly, C., Phillips, M., Denton, M., Elver, M., Potapenko, A., Morehouse, M., Tsyrlkevich, V., Holler, C., Lettner, J., Kilzer, D., & Brandt, L. (2024). GWP-ASan: Sampling-Based Detection of Memory-Safety Bugs in Production. *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24)*. <https://doi.org/10.1145/3639477.3640328>
- Szekeres, L., Payer, M., Wei, T., & Song, D. (2013). SoK: Eternal War in Memory. *2013 IEEE Symposium on Security and Privacy*, 48–62. <https://doi.org/10.1109/SP.2013.13>
- Bugden, W., & Alahmar, A. (2022). Rust: The Programming Language for Safety and Performance [arXiv:2206.05503]. <https://doi.org/10.48550/arXiv.2206.05503>
- Qin, B., Chen, Y., Yu, Z., Song, L., & Zhang, Y. (2020). Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, 763–779. <https://doi.org/10.1145/3385412.3386036>
- Fatima, A., Bibi, S., & Hanif, R. (2018). Comparative Study on Static Code Analysis Tools for C/C++. *2018 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, 465–469. <https://doi.org/10.1109/IBCAST.2018.8312265>
- Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why Don't Software Developers Use Static Analysis Tools to Find Bugs? *2013 35th International Conference on Software Engineering (ICSE)*, 672–681. <https://doi.org/10.1109/ICSE.2013.6606613>
- Nethercote, N., & Seward, J. (2007). Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6), 89–100. <https://doi.org/10.1145/1250734.1250746>
- Lee, J., Hong, S., & Oh, H. (2018). MemFix: Static Analysis-Based Repair of Memory Deallocation Errors for C. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18)*, 95–106. <https://doi.org/10.1145/3236024.3236079>
- Krishna, K., Murthy, P., & Sarangi, S. (2024). Exploring the Synergy Between Generative AI and Software Engineering: Automating Code Optimization and Bug Fixing. *World Journal of Advanced Engineering Technology and Sciences*, 13(1), 682–691. <https://doi.org/10.30574/wjaets.2024.13.1.0464>

- Davis, M. R. (2015). Automatic Memory Management Techniques for the Go Programming Language. <https://rest.mars-prod.its.unimelb.edu.au/server/api/core/bitstreams/51b7db52-f369-5f48-b385-38ce3c19dee5/content>
- Will, T. T. (2022). *Das Neue C++ Handbuch* (1. Auflage). Rheinwerk Verlag.
- Wolf, J. (2014). *C von A bis Z: Das umfassende Handbuch* (3., aktualisierte und erweiterte Auflage) [Mit CD und Referenzkarte, 1.190 Seiten]. Rheinwerk Computing. https://openbook.rheinwerk-verlag.de/c_von_a_bis_z/
- Bryant, R. E., & O'Hallaron, D. R. (2019). Lecture 19: Malloc: Basic Concepts [Course 15-213: Introduction to Computer Systems, Carnegie Mellon University].
- Tanenbaum, A. S., & Bos, H. (2015). *Modern Operating Systems* (4. Aufl.). Prentice Hall.
- cppreference.com. (2012). *gets* [Accessed: 2025-04-08]. <https://en.cppreference.com/w/c/io/gets>
- Lhee, K., & Chapin, S. J. (2003). Buffer Overflow and Format String Overflow Vulnerabilities. *Software: Practice and Experience*, 33(5), 423–460. <https://doi.org/10.1002/spe.515>
- Caballero, J., Grieco, G., Marron, M., & Nappa, A. (2012). Undangle: Early Detection of Dangling Pointers in Use-After-Free and Double-Free Vulnerabilities. *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA '12)*, 133–143. <https://doi.org/10.1145/2338965.2336769>
- Xie, Y., & Aiken, A. (2005). Context- and Path-Sensitive Memory Leak Detection. *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '05)*, 115–125. <https://doi.org/10.1145/1081706.1081728>
- MITRE Corporation. (2025a). New to CWE [Seite zuletzt aktualisiert am 05. Juni 2023; abgerufen am 26. April 2025].
- MITRE Corporation. (2025b). About CWE [Abgerufen am 26. April 2025].
- Louridas, P. (2006). Static Code Analysis. *IEEE Software*, 23(4), 58–61. <https://doi.org/10.1109/MS.2006.114>
- Bardas, A. G. (o.J.). STATIC CODE ANALYSIS [Erscheinungsjahr und weitere Angaben nicht verfügbar]. <https://core.ac.uk/download/pdf/6552448.pdf>
- LLVM Project. (o.J.). *clang-tidy* [Accessed: 2025-04-11]. <https://clang.llvm.org/extra/clang-tidy/>
- Bell, T. (1999). The Concept of Dynamic Analysis. *ACM SIGSOFT Software Engineering Notes*, 24(6), 216–234. <https://doi.org/10.1145/318774.318944>
- Larus, J. R., & Ball, T. (1994). Rewriting Executable Files to Measure Program Behavior. *Software: Practice and Experience*, 24(2), 197–218.

- Gosain, A., & Sharma, G. (2015). A Survey of Dynamic Program Analysis Techniques and Tools. In S. C. Satapathy, B. N. Biswal, S. K. Udgata & J. K. Mandal (Hrsg.), *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014* (S. 113–122, Bd. 327). Springer International Publishing. https://doi.org/10.1007/978-3-319-11933-5_13
- Moradi, M., Yan, K., Colwell, D., Samwald, M., & Asgari, E. (2024). Exploring the Landscape of Large Language Models: Foundations, Techniques, and Challenges [arXiv:2404.11973]. <https://doi.org/10.48550/arXiv.2404.11973>
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2023). Attention Is All You Need [arXiv:1706.03762]. <https://doi.org/10.48550/arXiv.1706.03762>
- Fan, L., Li, L., Ma, Z., Lee, S., Yu, H., & Hemphill, L. (2023). A Bibliometric Review of Large Language Models Research from 2017 to 2023 [arXiv:2304.02020]. <https://doi.org/10.48550/arXiv.2304.02020>
- National Institute of Standards and Technology (NIST). (2023). *What the Test Case Status Means (SARD Manual)*. Verfügbar 24. April 2025 unter https://www.nist.gov/itl/ssd/software-quality-group/sard-manual-what-test-case-status-means#Accepted_Test_Cases
- Ozturk, O. S., Ekmekcioglu, E., Cetin, O., Arief, B., & Hernandez-Castro, J. (2023). New Tricks to Old Codes: Can AI Chatbots Replace Static Code Analysis Tools? *European Interdisciplinary Cybersecurity Conference (EICC 2023)*, 13–18. <https://doi.org/10.1145/3590777.3590780>
- Fan, G., Xie, X., Zheng, X., Liang, Y., & Di, P. (2023). Static Code Analysis in the AI Era: An In-depth Exploration of the Concept, Function, and Potential of Intelligent Code Analysis Agents [arXiv:2310.08837]. <https://doi.org/10.48550/arXiv.2310.08837>
- White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., & Schmidt, D. C. (2023). A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. <https://arxiv.org/abs/2302.11382>
- Ma, W., Liu, S., Lin, Z., Wang, W., Hu, Q., Liu, Y., Zhang, C., Nie, L., Li, L., & Liu, Y. (2024). LMs: Understanding Code Syntax and Semantics for Code Analysis [arXiv:2305.12138]. <https://doi.org/10.48550/arXiv.2305.12138>
- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer. <https://doi.org/10.1007/978-3-642-29044-2>

Abbildungsverzeichnis

1	Einsatz von statischen Codeanalyse Tools im Entwicklungsprozess (Bardas, o.J. S. 1).	9
2	Screenshot von clang-tidy in Visual Studio Code.	9
3	Beispiel einer dynamischen Speicheranalyse mit Intel Inspector	11
4	ReAct Bug Detection Agent (G. Fan et al., 2023, S. 7)	17
5	Ablauf der Untersuchung	25
6	Ausgabe eines Testfalls	37
7	Visual Studio Code mit CMake und clang-tidy	44
8	Intel Inspector GUI – Import der Results	47

Tabellenverzeichnis

1	Kategorien und Beispiele der Prompt-Patterns White et al., 2023	18
2	Übersicht der verwendeten KI-Assistenten, Modelle und Plattformen	23
3	Spezifikationen der Testumgebung	23
4	Definition der Testfälle und erwartetes Verhalten	26
5	Angewandte Patterns für das Prompt	29
6	Beispielhafte KI-Ausgabe im standardisierten Tabellenformat	30
7	Definition des Bewertungsschemas der Tools	32
8	Mapping der Testfalls zu den Funktionsaufrufen	36
9	Auswertung von TC1_B	51
10	Auswertung von TC2_B	52
11	Auswertung von TC3_B	53
12	Auswertung von TC4_B	53
13	Auswertung von TC5_B	54
14	Auswertung von TC6_B	54
15	Auswertung von TC7_B	55
16	Auswertung von TC8_B	56
17	Auswertung von TC9_B	56
18	Auswertung von TC10_B	57
19	Auswertung von TC11_B	57
20	Auswertung von TC12_B	58
21	Auswertung von TC13_B	58
22	Auswertung von TC14_B	59
23	Auswertung von TC15_B	59
24	Analyse sicherheitsrelevanter Speicherfehler TC1_B GPT-4o	78
25	Analyse sicherheitsrelevanter Speicherfehler TC1_B Sonnet 3.7	80
26	Analyse sicherheitsrelevanter Speicherfehler TC1_B Gemini 2.5 Pro	82
27	Analyse sicherheitsrelevanter Speicherfehler TC2_B GPT-4o	85
28	Analyse sicherheitsrelevanter Speicherfehler TC2_B Sonnet	87
29	Analyse sicherheitsrelevanter Speicherfehler TC2_B Gemini 2.5 Pro	90
30	Analyse sicherheitsrelevanter Speicherfehler TC3_B ChatGPT-4o	93
31	Analyse sicherheitsrelevanter Speicherfehler TC3_B Sonnet 3.7	93
32	Analyse sicherheitsrelevanter Speicherfehler TC3_B Gemini 2.5 Pro	95
33	Analyse sicherheitsrelevanter Speicherfehler TC4_B ChatGPT-4o	98
34	Analyse sicherheitsrelevanter Speicherfehler TC4_B Sonnet 3.7	99
35	Analyse sicherheitsrelevanter Speicherfehler TC4_B Gemini 2.5 Pro	100
36	Analyse sicherheitsrelevanter Speicherfehler TC5_B GPT-4o	105
37	Analyse sicherheitsrelevanter Speicherfehler TC5_B Sonnet 3.7	106
38	Analyse sicherheitsrelevanter Speicherfehler TC5_B Gemini 2.5 Pro	108
39	Analyse sicherheitsrelevanter Speicherfehler TC6_B GPT-4o	113
40	Analyse sicherheitsrelevanter Speicherfehler TC6_B Sonnet 3.7	114
41	Analyse sicherheitsrelevanter Speicherfehler TC6_B Gemini 2.5 Pro	115
42	Analyse sicherheitsrelevanter Speicherfehler TC7_B GPT-4o	118
43	Analyse sicherheitsrelevanter Speicherfehler TC7_B Sonnet 3.7	119
44	Analyse sicherheitsrelevanter Speicherfehler TC7_B Gemini 2.5 Pro	121
45	Analyse sicherheitsrelevanter Speicherfehler TC8_B ChatGPT-4o	125
46	Analyse sicherheitsrelevanter Speicherfehler TC8_B Sonnet 3.7	126

47	Analyse sicherheitsrelevanter Speicherfehler	TC8_B Gemini 2.5 Pro	128
48	Analyse sicherheitsrelevanter Speicherfehler	TC9_B ChatGPT-4o	133
49	Analyse sicherheitsrelevanter Speicherfehler	TC9_B Sonnet 3.7	135
50	Analyse sicherheitsrelevanter Speicherfehler	TC9_B Gemini 2.5 Pro	137
51	Analyse sicherheitsrelevanter Speicherfehler	TC10_B ChatGPT-4o	141
52	Analyse sicherheitsrelevanter Speicherfehler	TC10_B Sonnet 3.7	142
53	Analyse sicherheitsrelevanter Speicherfehler	TC10_B Gemini 2.5 Pro . . .	146
54	Analyse sicherheitsrelevanter Speicherfehler	TC11_B ChatGPT (GPT-4o)	149
55	Analyse sicherheitsrelevanter Speicherfehler	TC11_B Sonnet 3.7	150
56	Analyse sicherheitsrelevanter Speicherfehler	TC11_B Gemini 2.5 Pro . . .	153
57	Analyse sicherheitsrelevanter Speicherfehler	TC12_B ChatGPT-4o	158
58	Analyse sicherheitsrelevanter Speicherfehler	TC12_B Sonnet 3.7	159
59	Analyse sicherheitsrelevanter Speicherfehler	TC12_B Gemini 2.5 Pro . . .	162
60	Analyse sicherheitsrelevanter Speicherfehler	TC13_B ChatGPT-4o	166
61	Analyse sicherheitsrelevanter Speicherfehler	TC13_B Sonnet 3.7	167
62	Analyse sicherheitsrelevanter Speicherfehler	TC13_B Gemini 2.5 Pro . . .	169
63	Analyse sicherheitsrelevanter Speicherfehler	TC14_B ChatGPT-4o	174
64	Analyse sicherheitsrelevanter Speicherfehler	TC14_B Sonnet 3.7	175
65	Analyse sicherheitsrelevanter Speicherfehler	TC14_B Gemini 2.5 Pro . . .	180
66	Analyse sicherheitsrelevanter Speicherfehler	TC15_B ChatGPT-4o	185
67	Analyse sicherheitsrelevanter Speicherfehler	TC15_B Sonnet 3.7	186
68	Analyse sicherheitsrelevanter Speicherfehler	TC15_B Gemini 2.5 Pro . . .	191

A KI-Prompts und Antworten

A.1 Vorwort zum Appendix

In diesem Anhang sind alle Prompts und KI-Antworten dokumentiert, die im Rahmen der Testdurchführung verwendet wurden. Für jeden Testfall wurde der Quellcode zusammen mit einem standardisierten Prompt an das jeweilige Modell übergeben. Die Antworten der KI sind inhaltlich unbearbeitet wiedergegeben. Da die Ki-Tools zwar den LaTeX Code für die Tabellen generieren, diese jedoch vom Format nicht passen oder gar nicht kompilieren, werden diese direkt angepasst. Der generierte LaTeX-Code ist somit nicht in diesen Listings erhalten.

Listing 24 zeigt ein Beispiel eines solchen Outputs. Dieser Code kompiliert zum Beispiel nicht, da die KI Sonderzeichen wie den “_”-Unterstrich nicht escaped und das *LaTeX*-Listing Paket ein escaping erwartet. Desweiteren ist die Formatierung der Zellen nicht passend zu dem *LaTeX*-Projekt dieser Arbeit, da Texte einer Zelle in die nächste Zelle schreiben, aufgrund unpassend gewählten Column-Properties (c, l, X...).

```

1 \begin{table}[H]
2 \centering
3 \caption{Analyse sicherheitsrelevanter Speicherfehler im Codebeispiel}
4 \begin{tabularx}{\textwidth}{@{} c c l l X X @{}}
5 \toprule
6 \textbf{Nr.} & \textbf{CWE-ID} & \textbf{Name der Schwachstelle} & \textbf{Fehlerart} \\
7 & \textbf{Beschreibung} & \textbf{Empfehlung} & \\
8 \midrule
9 1 & CWE-121 & Stack-based Buffer Overflow & Buffer Overflow & \\
10 Die Funktion \texttt{memcpy(cv.charFirst, SRC_STR, sizeof(cv));} kopiert mehr Bytes \\
11 ↳ (typisch 24) als der Zielpuffer \texttt{charFirst} (nur 16 Bytes) aufnehmen \\
12 ↳ kann. Dadurch wird angrenzender Speicher, u.a. \texttt{voidSecond}, ü \\
13 ↳ beschrieben, was zu undefiniertem Verhalten und potenziellen \\
14 ↳ Sicherheitslücken führt. & \\
15 Die Länge des \texttt{memcpy}-Aufrufs muss auf \texttt{sizeof(cv.charFirst)} \\
16 ↳ beschränkt werden: \texttt{memcpy(cv.charFirst, SRC_STR, \\
17 ↳ sizeof(cv.charFirst));} \\
18 \bottomrule
19 \end{tabularx}
20 \end{table}

```

Listing 24: Beispiel eines generierten LaTeX Codes von ChatGPT

Listing 25 zeigt den korrigierten Code, der die oben genannten Probleme behebt.

```

1 \begin{table}[H]
2 \centering
3 \caption{Analyse sicherheitsrelevanter Speicherfehler}
4 \begin{tabularx}{\textwidth}{@{} c p{1cm} p{2cm} p{2cm} X X @{}}
5 \toprule
6 \textbf{Nr.} & \textbf{CWE-ID} & \textbf{Name der Schwachstelle} & \textbf{Fehlerart} \\
7 & \textbf{Beschreibung} & \textbf{Empfehlung} & \\
8 \midrule
9 1 & 121 & Stack-based Buffer Overflow & Buffer Overflow & \\
10 Die Funktion \texttt{memcpy(cv.charFirst, SRC\_STR, sizeof(cv));} kopiert mehr Bytes \\
11 ↳ als der Puffer \texttt{cv.charFirst} aufnehmen kann (nur 16 Bytes). Es wird \\
12 ↳ \texttt{sizeof(cv)} (typisch 24 Bytes) kopiert, wodurch angrenzende

```

```

10 ↪ Speicherbereiche, darunter \texttt{voidSecond}, überschrieben werden können.
    & Die Länge des \texttt{memcpy}-Aufrufs muss auf \texttt{sizeof(cv.charFirst)}
    ↪ beschränkt werden, um eine Pufferüberschreitung zu vermeiden:
    ↪ \texttt{memcpy(cv.charFirst, SRC\_STR, sizeof(cv.charFirst));} \\
11 \bottomrule
12 \end{tabularx}
13 \end{table}

```

Listing 25: Korrigierter Code

A.2 Testfall *TC1_B*

A.2.1 Prompt

Du bist ein Sicherheitsexperte mit Spezialisierung auf C/C++-Codeanalyse. Dein Ziel ist es, sicherheitsrelevante Speicherverwaltungsfehler zu erkennen und zu beheben. Analysiere den folgenden C/C++-Code ausschließlich im Hinblick auf sicherheitskritische Speicherfehler wie Buffer Overflows, Use-after-free oder Memory Leaks und behebe diese auch gleich falls der Code solche Fehler hat.

Erzeuge eine Tabelle in Latex mit den folgenden Spalten:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
-----	--------	------------------------	-----------	--------------	------------

Analysiere ausschließlich sicherheitsrelevante Aspekte, keine Stil- oder Architekturfragen.

```
```cpp
```

```

#define ENABLE_TEST_OUTPUT 1

#ifdef ENABLE_TEST_OUTPUT
#include <sstream>
#include <iostream>
#include <codecvt>
#include <locale>

#define TEST_OUTPUT(x) do { \
 std::wstringstream wss; \
 wss << x; \
 std::wcout << "\t" << wss.str() << std::endl; \
} while (0)

#else
#define TEST_OUTPUT(x)
#endif

typedef struct charVoid {
 char charFirst[16];
 void* voidSecond;

```



```
void* voidThird;
} charVoid;

void test() {

 const char* SRC_STR = "0123456789abcdef0123456789abcde";

 charVoid cv;
 cv.voidSecond = (void*)SRC_STR;

 TEST_OUTPUT("Vor memcpy: " << (char*)cv.voidSecond);

 memcpy(cv.charFirst, SRC_STR, sizeof(cv));

 cv.charFirst[sizeof(cv.charFirst) - 1] = '\\0';

 TEST_OUTPUT("Nach memcpy: " << cv.charFirst);

 TEST_OUTPUT("Möglicherweise beschädigt: " << (char*)cv.voidSecond);
}
...
```

## A.2.2 GPT-4o Antwort TC1\_B

Hier ist die sicherheitsrelevante Analyse des gegebenen C++-Codes, fokussiert auf Speicherverwaltungsfehler. Der kritische Teil ist die falsche Verwendung von memcpy, die zu einem Stack-basierten Buffer Overflow führen kann. Die Tabelle ist im gewünschten LaTeX-Format ausgegeben:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	121	Stack-based Buffer Overflow	Buffer Overflow	Die Funktion memcpy(cv.charFirst, SRC_STR, sizeof(cv)); kopiert mehr Bytes als der Puffer cv.charFirst aufnehmen kann (nur 16 Bytes). Es wird sizeof(cv) (typisch 24 Bytes) kopiert, wodurch angrenzende Speicherbereiche, darunter voidSecond, überschrieben werden können.	Die Länge des memcpy-Aufrufs muss auf sizeof(cv.charFirst) beschränkt werden, um eine Pufferüberschreitung zu vermeiden: memcpy(cv.charFirst, SRC_STR, sizeof(cv.charFirst));

Tabelle 24: Analyse sicherheitsrelevanter Speicherfehler TC1\_B GPT-4o

### Sichere korrigierte Version des Codes

```

1 void test() {
2
3 const char* SRC_STR = "0123456789abcdef0123456789abcde";
4
5 charVoid cv;
6 cv.voidSecond = (void*)SRC_STR;
7
8 TEST_OUTPUT("Vor memcpy: " << (char*)cv.voidSecond);
9
10 // Fix: Verwende die korrekte Zielpuffergröße
11 memcpy(cv.charFirst, SRC_STR, sizeof(cv.charFirst));
12
13 cv.charFirst[sizeof(cv.charFirst) - 1] = '\\0';
14
15 TEST_OUTPUT("Nach memcpy: " << cv.charFirst);
16
17 TEST_OUTPUT("Möglicherweise beschädigt: " << (char*)cv.
18 ↪ voidSecond);
19 }
```

Listing 26: Korrigierte Version des Codes für TC1\_B

### Hintergrund zur Analyse:

**Fehlertyp:** Klassischer Stack-basierter Buffer Overflow.

**Grund:** memcpy() kopiert 24 Bytes (sizeof(cv)), obwohl der Zielpuffer cv.charFirst nur 16 Bytes groß ist.

**Auswirkung:** voidSecond wird möglicherweise überschrieben, was zu undefiniertem Verhalten oder Sicherheitslücken führen kann.

**Empfohlene Änderung:** Begrenzung der memcpy-Länge auf die tatsächliche Größe des Puffers.

### A.2.3 Sonnet 3.7 Antwort TC1\_B

Der gegebene C++-Code enthält mehrere kritische Speicherverwaltungsfehler. Hier die Analyse:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-121	Stack-based Buffer Overflow	Buffer Overflow	memcpy kopiert 32 Bytes (Größe von charVoid) in ein 16-Byte-Array charFirst, überschreibt angrenzende Stack-Daten inkl. Pointer	memcpy auf sizeof(cv.charFirst) (16 Bytes) begrenzen
2	CWE-825	Use of Invalid Pointer	Use-after-corrupt	Zugriff auf cv.voidSecond nach dem Overflow: Pointer wurde durch Buffer-Overflow möglicherweise korumpiert	Buffer-Overflow beheben (siehe CWE-121)
3	CWE-125	Out-of-bounds Read	Informationsleck	Testausgabe von (char*)cv.voidSecond nach dem Overflow könnte nicht mehr terminierte Daten oder geheimen Speicherinhalt offenlegen	Pointer-Zugriffe erst nach Behebung des Buffer-Overflows validieren

Tabelle 25: Analyse sicherheitsrelevanter Speicherfehler TC1\_B Sonnet 3.7

#### Erläuterung der Hauptprobleme:

##### 1. Kritischer Buffer Overflow

Die Anweisung `memcpy(cv.charFirst, SRC_STR, sizeof(cv))` kopiert 32 Bytes (Größe der gesamten charVoid-Struktur) in das 16-Byte-Array charFirst. Dies überschreibt:

- Den voidSecond-Pointer (Bytes 16–23)
- Den voidThird-Pointer (Bytes 24–31)
- Eventuell andere Stack-Daten

##### 2. Pointer-Korruption

Durch den Overflow wird voidSecond mit Daten aus SRC\_STR überschrieben. Der spätere Zugriff (`(char*)cv.voidSecond` in TEST\_OUTPUT) liest somit einen invaliden Speicherbereich.

##### 3. String-Terminierungsproblem

Die manuelle Null-Terminierung `cv.charFirst = '\0'` ist wirkungslos, da der Overflow bereits alle 16 Bytes inkl. des letzten Array-Elements überschrieben hat.

#### Korrigierter Codeausschnitt:

```

1 // Korrekte memcpy-Anweisung:
2 memcpy(cv.charFirst, SRC_STR, sizeof(cv.charFirst) - 1);
3 cv.charFirst[sizeof(cv.charFirst) - 1] = '\0'; // Explizite
 ↪ Terminierung
4
5 // Pointer nach memcpy nicht mehr verwenden, da ungültig:

```

```
6 // TEST_OUTPUT("Möglicherweise beschädigt: " << (char*)cv.
 ↪ voidSecond); // entfernen
```

Listing 27: Korrigierte Version des Codes für TC1\_B Sonnet

**Schlussfolgerung:**

Der Code demonstriert klassische Speicherkorruptionsfehler durch ungeprüfte Kopiervorgänge. Die Behebung erfordert strikte Begrenzung der Kopiergröße auf die Zielpufferkapazität und Validierung aller Pointer-Zugriffe nach Pufferoperationen.

## A.2.4 Gemini 2.5 Pro Antwort TC1\_B

Der gegebene C++-Code enthält zwei kritische Speichersicherheitslücken, die zu Buffer Overflows und Speicherbeschädigung führen können. Hier die detaillierte Analyse:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	120	Buffer Copy Without Checking Size	Buffer Overflow	memcpy kopiert 32 Bytes (Größe der gesamten charVoid-Struktur) in ein 16-Byte-Array charFirst. Dies überschreibt die nachfolgenden Pointer voidSecond und voidThird	memcpy auf Zielgröße begrenzen: memcpy(cv.charFirst, SRC_STR, sizeof(cv.charFirst))
2	416	Use After Free	Speicherbeschädigung	Durch den Buffer Overflow werden voidSecond und voidThird überschrieben. Der spätere Zugriff auf (char*)cv.voidSecond verwendet einen ungültigen Pointer	Pointer-Zugriffe nach Buffer Overflow vermeiden, Strukturlayout überprüfen

Tabelle 26: Analyse sicherheitsrelevanter Speicherfehler TC1\_B Gemini 2.5 Pro

### Erläuterung der Fehler:

#### 1. Kritischer Buffer Overflow

Die Struktur charVoid hat folgende Speicherbelegung auf 64-Bit-Systemen:

```

1 char charFirst[16]; // 16 Bytes
2 void* voidSecond; // 8 Bytes
3 void* voidThird; // 8 Bytes
4 // Gesamt: 32 Bytes

```

Listing 28: Speicherlayout der Struktur charVoid

Der memcpy-Befehl sizeof(cv) kopiert 32 Bytes in das 16-Byte-Array, wodurch die nachfolgenden Pointer überschrieben werden.

#### 2. Ungültige Pointerverwendung

Nach dem Overflow enthält voidSecond keine gültige Adresse mehr. Die Ausgabe (char\*)cv.voidSecond im Test führt zu undefiniertem Verhalten.

### Korrigierter Codeausschnitt:

```

1 // Korrekte memcpy-Operation mit begrenzter Größe
2 memcpy(cv.charFirst, SRC_STR, sizeof(cv.charFirst));
3
4 // Sicherstellen, dass Pointer nicht beschädigt werden
5 cv.charFirst[sizeof(cv.charFirst)-1] = '\0'; // Null -
 ↪ Terminierung

```

Listing 29: Korrekte memcpy-Nutzung mit Null-Terminierung

Diese Korrekturen verhindern den Buffer Overflow und erhalten die Integrität der Pointer in der Struktur.

## A.3 Testfall *TC2\_B*

### A.3.1 Prompt

Du bist ein Sicherheitsexperte mit Spezialisierung auf C/C++-Codeanalyse. Dein Ziel ist es, sicherheitsrelevante Speicherverwaltungsfehler zu erkennen und zu beheben. Analysiere den folgenden C/C++-Code ausschließlich im Hinblick auf sicherheitskritische Speicherfehler wie Buffer Overflows, Use-after-free oder Memory Leaks und behebe diese auch gleich falls der Code solche Fehler hat.

Erzeuge eine Tabelle in Latex mit den folgenden Spalten:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
-----	--------	------------------------	-----------	--------------	------------

Analysiere ausschließlich sicherheitsrelevante Aspekte, keine Stil- oder Architekturfragen.

```
```cpp
#define ENABLE_TEST_OUTPUT 1

#if ENABLE_TEST_OUTPUT
#include <sstream>
#include <iostream>
#include <codecvt>
#include <locale>

#define TEST_OUTPUT(x) do { \
    std::wstringstream wss; \
    wss << x; \
    std::wcout << "\t" << wss.str() << std::endl; \
} while (0)

#else
#define TEST_OUTPUT(x)
#endif

#define MAXSIZE 80

char* helper_2(char* p, int n, int targ) {
    if (n > targ)
        return helper_2(p+1, n-1, targ);
    return p;
}

void helper_1(char* str) {
    char* buf;

    buf = (char*)malloc(MAXSIZE);
    if (!buf)
```



```
    return;

    strcpy(buf, str);
    TEST_OUTPUT("result: " << buf);

    free(buf);
}

void test(int argc, char** argv) {
    char* userstr, * str2;

    if (argc > 1) {
        userstr = argv[1];
        str2 = helper_2(userstr, strlen(userstr), 80);
        helper_1(str2);
    }
}
...
```

A.3.2 GPT-4o Antwort TC2_B

Hier ist die sicherheitsfokussierte Analyse des C-Codes in LaTeX-Tabellenform. Die Funktion `strcpy` wird ohne Längenprüfung verwendet, was potenziell zu einem Buffer Overflow führt.

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-120	Classic Buffer Overflow	Buffer Overflow	In der Funktion <code>helper_1</code> wird <code>strcpy(buf, str)</code> verwendet, ohne die Länge von <code>str</code> zu prüfen. Wenn <code>str</code> länger als 79 Zeichen ist (wegen <code>MAXSIZE = 80</code>), kommt es zu einem Überlauf.	Ersetzen durch <code>strncpy(buf, str, MAXSIZE - 1)</code> und anschließendes manuelles Nullterminieren mit <code>buf[MAXSIZE - 1] = '\0';</code> , um einen Überlauf zu verhindern.

Tabelle 27: Analyse sicherheitsrelevanter Speicherfehler TC2_B GPT-4o

Fix-Vorschlag für den Code:

```
1  strncpy(buf, str, MAXSIZE - 1);
2  buf[MAXSIZE - 1] = '\0';
```

Listing 30: Korrigierte Version des Codes für TC2_B ChatGPT

Begründung

Die Funktion `helper_2` verschiebt zwar den Zeiger innerhalb des übergebenen Strings, prüft aber nicht, ob das verbleibende Substring eine Länge größer als `MAXSIZE` hat. Selbst wenn der Nutzer 100 Zeichen eingibt und `targ = 80`, kann `str2` auf ein Substring mit 80 Zeichen zeigen - was dann `helper_1` potenziell überlastet.

Wenn du möchtest, kann ich die `helper_2`-Funktion auch noch absichern.

A.3.2.1 Reprompt

kannst du den gesamten Code mit deinem fix anzeigen?

A.3.2.2 GPT-4o Antwort TC2_B Reprompt

Natürlich! Hier ist der vollständig überarbeitete Code mit einem Fix gegen den Buffer Overflow in `helper_1`, unter Verwendung von `strncpy` mit zusätzlicher Nullterminierung:

```
1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>
7 #include <locale>
8
9 #define TEST_OUTPUT(x) do { \
10     std::wstringstream wss; \
11     wss << x; \
12     std::wcout << "\t" << wss.str() << std::endl; \
13 } while (0)
```

```
14
15 #else
16 #define TEST_OUTPUT(x)
17 #endif
18
19 #include <stdlib.h>
20 #include <string.h>
21
22 #define MAXSIZE 80
23
24 char* helper_2(char* p, int n, int targ) {
25     if (n > targ)
26         return helper_2(p + 1, n - 1, targ);
27     return p;
28 }
29
30 void helper_1(char* str) {
31     char* buf;
32
33     buf = (char*)malloc(MAXSIZE);
34     if (!buf)
35         return;
36
37     // Fix: sicheres Kopieren mit Begrenzung und explizitem Null-
38     //      ↪ Terminator
39     strncpy(buf, str, MAXSIZE - 1);
40     buf[MAXSIZE - 1] = '\0';
41
42     TEST_OUTPUT("result: " << buf);
43
44     free(buf);
45 }
46
47 void test(int argc, char** argv) {
48     char* userstr, * str2;
49
50     if (argc > 1) {
51         userstr = argv[1];
52         str2 = helper_2(userstr, strlen(userstr), 80);
53         helper_1(str2);
54     }
55 }
```

Listing 31: Korrigierte Version des Codes für TC2_B ChatGPT Prompt 2

Hinweise zur Sicherheit

- `strncpy` wird verwendet, um maximal `MAXSIZE - 1` Zeichen zu kopieren.
- Ein expliziter `\0`-Terminator wird gesetzt, um sicherzustellen, dass `buf` korrekt null-terminiert ist – selbst wenn `str` genau `MAXSIZE - 1` Zeichen lang ist.

- Keine weiteren kritischen Speicherfehler (wie Use-after-free oder Memory Leaks) sind in diesem Code vorhanden.

Wenn du möchtest, kann ich zusätzlich auch `helper_2` so absichern, dass sie niemals zu einem zu langen Substring führt.

A.3.3 Sonnet 3.7 Antwort TC2_B

Sicherheitsanalyse des C/C++-Codes

Nach einer gründlichen Analyse des bereitgestellten C/C++-Codes habe ich mehrere sicherheitsrelevante Speicherverwaltungsfehler identifiziert. Diese Fehler können zu schwerwiegenden Sicherheitsproblemen wie Buffer Overflows und potenziellen Speicherlecks führen.

identifizierte Sicherheitsprobleme

Der Code enthält mehrere kritische Sicherheitslücken, die mit bekannten Common Weakness Enumeration (CWE) Kategorien übereinstimmen. Hier ist eine detaillierte Analyse:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-787	Out-of-Bounds Write	Buffer Overflow	In der Funktion <code>helper_1</code> wird <code>strcpy()</code> ohne Größenprüfung verwendet. Wenn <code>str</code> länger als <code>MAXSIZE</code> ist, wird ein Buffer Overflow verursacht.	Verwende <code>strncpy()</code> oder besser noch <code>strncpy()</code> mit expliziter Größenbegrenzung: <code>strncpy(buf, str, MAXSIZE-1); buf[MAXSIZE-1] = '\0';</code>
2	CWE-119	Improper Restriction of Operations within Memory Buffer	Buffer Overflow	Die Funktion <code>helper_2</code> führt keine Validierung der Eingabeparameter durch. Der Wert <code>targ (80)</code> wird mit der Stringlänge verglichen, aber es gibt keine Garantie, dass der zurückgegebene Zeiger auf einen null-terminierten String zeigt.	Implementiere eine Prüfung, ob der zurückgegebene Zeiger auf einen gültigen String zeigt und ob dieser in den Zielpuffer passt.
3	CWE-476	NULL Pointer Dereference	Null-Dereferenzierung	In der Funktion <code>test</code> wird nicht geprüft, ob der von <code>helper_2</code> zurückgegebene Zeiger gültig ist, bevor er an <code>helper_1</code> übergeben wird.	Füge eine Null-Prüfung hinzu: <code>if (str2 != NULL) helper_1(str2);</code>

Tabelle 28: Analyse sicherheitsrelevanter Speicherfehler TC2_B Sonnet

Korrigierter Code:

```

1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>

```

```
7 #include <locale>
8
9 #define TEST_OUTPUT(x) do { \
10     std::wstringstream wss; \
11     wss << x; \
12     std::wcout << "\t" << wss.str() << std::endl; \
13 } while (0)
14
15 #else
16 #define TEST_OUTPUT(x)
17 #endif
18
19 #include <string.h>
20 #include <stdlib.h>
21
22 #define MAXSIZE 80
23
24 char* helper_2(char* p, int n, int targ) {
25     // Sicherheitsprüfung hinzugefügt
26     if (p == NULL || n < 0 || targ < 0)
27         return NULL;
28
29     if (n > targ)
30         return helper_2(p+1, n-1, targ);
31     return p;
32 }
33
34 void helper_1(char* str) {
35     if (str == NULL)
36         return;
37
38     char* buf;
39
40     buf = (char*)malloc(MAXSIZE);
41     if (!buf)
42         return;
43
44     // Sicheres Kopieren mit Größenbegrenzung
45     strncpy(buf, str, MAXSIZE-1);
46     buf[MAXSIZE-1] = '\0'; // Sicherstellen, dass der String null-
47                             // ↳ terminiert ist
48
49     TEST_OUTPUT("result: " << buf);
50
51     free(buf);
52 }
53
54 void test(int argc, char** argv) {
55     if (argc <= 1)
56         return;
```

```
57     char* userstr = argv[1];
58
59     // Sicherheitsprüfung für userstr
60     if (userstr == NULL)
61         return;
62
63     size_t len = strlen(userstr);
64     char* str2 = helper_2(userstr, len, 80);
65
66     // Prüfen, ob str2 gültig ist
67     if (str2 != NULL) {
68         helper_1(str2);
69     }
70 }
```

Listing 32: Korrigierte Version des Codes für TC2_B Sonnet

Diese korrigierte Version behebt die identifizierten Sicherheitsprobleme durch:

1. Verwendung von `strncpy()` mit expliziter Größenbegrenzung anstelle von `strcpy()`
2. Hinzufügen von Null-Prüfungen für alle Zeiger
3. Sicherstellen, dass Strings immer null-terminiert sind
4. Validierung der Eingabeparameter in allen Funktionen

A.3.4 Gemini 2.5 Pro Antwort TC2_B

Als Sicherheitsexperte für C/C++-Code werde ich den Code auf speicherbezogene Sicherheitslücken untersuchen.

Hier ist die LaTeX-Tabelle mit einer Analyse der Sicherheitslücken in Bezug auf die Speicherverwaltung:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-120	Stack-based Buffer Overflow	Buffer Overflow	Die Funktion <code>helper_1</code> verwendet <code>strcpy</code> , um den Inhalt von <code>str</code> in <code>buf</code> zu kopieren. Wenn <code>str</code> größer als <code>MAXSIZE</code> ist, kommt es zu einem Pufferüberlauf.	Verwenden Sie <code>strncpy</code> anstelle von <code>strcpy</code> oder überprüfen Sie vor dem Kopieren die Länge des Eingangsstrings.
2	CWE-401	Memory Leak	Memory Leak	Die Funktion <code>test</code> ruft <code>helper_2</code> mit dem dritten Argument fest auf 80 gesetzt auf. Wenn die Länge von <code>userstr</code> immer größer als 80 ist, gibt <code>helper_2</code> immer <code>userstr</code> zurück. Wenn <code>argc</code> größer als 1 ist und <code>strlen(userstr)</code> immer größer als 80 ist, wird <code>str2</code> auf <code>userstr</code> zeigen, ohne dass Speicher zugewiesen wird, was zu einem Memory Leak führen kann, wenn der Wert von <code>userstr</code> geändert wird.	Stellen Sie sicher, dass <code>str</code> die Größe des Puffers <code>buf</code> in <code>helper_1</code> nicht überschreitet.

Tabelle 29: Analyse sicherheitsrelevanter Speicherfehler TC2_B Gemini 2.5 Pro

Erläuterungen:

- **CWE-120 Stack-based Buffer Overflow:** Die Verwendung von `strcpy` in `helper_1` ist gefährlich, da die Funktion keine Möglichkeit hat, die Größe der Eingabe (`str`) zu überprüfen, bevor sie in `buf` kopiert wird. Wenn `str` größer als `MAXSIZE` ist, schreibt `strcpy` über das Ende des Puffers `buf` hinaus, was zu einem Pufferüberlauf führt.
- **CWE-401 Memory Leak:** Wenn `strlen(userstr)` immer größer als 80 ist, führt der Aufruf von `helper_2` dazu, dass `str2` auf `userstr` zeigt, ohne Speicher zuzuweisen. Wenn der Wert von `userstr` geändert wird, führt dies zu einem Memory Leak.

Empfehlungen:

- Um Pufferüberläufe zu verhindern, sollte `strcpy` durch `strncpy` ersetzt werden, wobei die Größe des Zielpuffers angegeben wird. Alternativ kann die Länge des Eingangsstrings vor dem Kopieren überprüft werden.
- Um Memory Leaks zu vermeiden, sollte sichergestellt werden, dass Speicher, der dynamisch zugewiesen wurde, auch wieder freigegeben wird, wenn er nicht mehr benötigt wird.

A.4 Testfall *TC3_B*

A.4.1 Prompt

Du bist ein Sicherheitsexperte mit Spezialisierung auf C/C++-Codeanalyse. Dein Ziel ist es, sicherheitsrelevante Speicherverwaltungsfehler zu erkennen und zu beheben. Analysiere den folgenden C/C++-Code ausschließlich im Hinblick auf sicherheitskritische Speicherfehler wie Buffer Overflows, Use-after-free oder Memory Leaks und behebe diese auch gleich falls der Code solche Fehler hat.

Erzeuge eine Tabelle in Latex mit den folgenden Spalten:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
-----	--------	------------------------	-----------	--------------	------------

Analysiere ausschließlich sicherheitsrelevante Aspekte, keine Stil- oder Architekturfragen.

```
```cpp
#define ENABLE_TEST_OUTPUT 1

#if ENABLE_TEST_OUTPUT
#include <sstream>
#include <iostream>
#include <codecvt>
#include <locale>

#define TEST_OUTPUT(x) do { \
 std::wstringstream wss; \
 wss << x; \
 std::wcout << "\t" << wss.str() << std::endl; \
} while (0)

#else
#define TEST_OUTPUT(x)
#endif

#define SIZE 100

void helper(wchar_t* data) {
{
 wchar_t source[SIZE];
 wmemset(source, L'C', SIZE - 1);
 source[SIZE - 1] = L'\0';
 TEST_OUTPUT(L"data vor wcsncpy: " << data);
 wcsncpy(data, source, SIZE - 1);
 data[SIZE - 1] = L'\0';
 TEST_OUTPUT(L"data nach wcsncpy: " << data);
}
}
```

```
void test() {
 wchar_t* data;
 data = NULL;
 {
 wchar_t* dataBuffer = (wchar_t*)malloc(SIZE * sizeof(wchar_t));
 if (dataBuffer == NULL) { exit(-1); }
 wmemset(dataBuffer, L'A', SIZE - 1);
 dataBuffer[SIZE - 1] = L'\0';
 data = dataBuffer - 8;
 }
 helper(data);
}

...
```

#### A.4.2 GPT-4o Antwort TC3\_B

Hier ist die sicherheitsrelevante Analyse deines C-Codes. Die Kernproblematik liegt bei einem manipulierten Zeiger vor dem Pufferanfang, was zu undefiniertem Verhalten und potentiell einem Buffer Overflow führen kann. Zusätzlich besteht ein hohes Risiko für Use-after-free oder Memory Corruption.

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-124	Buffer Underwrite (aka Buffer Underflow)	Buffer Overflow	Im Funktionsblock <code>test()</code> wird der Pointer <code>data</code> absichtlich auf 8 <code>wchar_t</code> vor dem zugewiesenen Speicher gesetzt ( <code>dataBuffer - 8</code> ). Dies führt dazu, dass <code>wcsncpy()</code> in <code>helper()</code> Daten in Speicherbereiche schreibt, die außerhalb des allokierten Puffers liegen.	Zeigerarithmetik vermeiden: <code>data = dataBuffer;</code> setzen, um nur innerhalb des allokierten Puffers zu schreiben.

Tabelle 30: Analyse sicherheitsrelevanter Speicherfehler TC3\_B ChatGPT-4o

#### Behebungsvorschlag (korrigierter Codeausschnitt):

```

1 void test() {
2 wchar_t* data;
3 data = NULL;
4 {
5 wchar_t* dataBuffer = (wchar_t*)malloc(SIZE * sizeof(wchar_t));
6 if (dataBuffer == NULL) { exit(-1); }
7 wmemset(dataBuffer, L'A', SIZE - 1);
8 dataBuffer[SIZE - 1] = L'\0'; // Null-Terminierung
9 data = dataBuffer; // Fix: kein pointer offset
10 helper(data);
11 }
12 }
```

Listing 33: Korrigierte Version des Codes für TC3\_B

#### A.4.3 Sonnet 3.7 Antwort TC3\_B

Der gegebene C++-Code enthält mehrere kritische Speicherverwaltungsfehler, die zu Sicherheitslücken führen können. Hier sind die identifizierten Probleme und Lösungsvorschläge:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	590	Free of Invalid Pointer	Use-after-free	<code>data</code> zeigt durch Zeigerarithmetik ( <code>dataBuffer - 8</code> ) auf ungültigen Speicher	Zeigerarithmetik entfernen: <code>data = dataBuffer</code> statt <code>dataBuffer - 8</code>
2	787	Out-of-bounds Write	Buffer Overflow	Schreibzugriff auf <code>data</code> überschreitet die allokierte Speichergrenze	Korrekte Initialisierung: <code>data = dataBuffer</code> beibehalten

Tabelle 31: Analyse sicherheitsrelevanter Speicherfehler TC3\_B Sonnet 3.7

**Korrigierter Codeausschnitt:**

```
1 void test() {
2 wchar_t* data = nullptr;
3 wchar_t* dataBuffer = static_cast<wchar_t*>(malloc(SIZE *
4 ↪ sizeof(wchar_t)));
5 if (!dataBuffer) exit(-1);
6
7 wmemset(dataBuffer, L'A', SIZE - 1);
8 dataBuffer[SIZE - 1] = L'\0';
9 data = dataBuffer; // Korrekte Zuordnung
10
11 helper(data);
12 }
```

Listing 34: Verbesserter Umgang mit Speicher und Zeigern für TC3\_B Sonnet 3.7

**Erläuterung der Änderungen:**

1. **Zeigerarithmetik entfernt:** Die ursprüngliche Operation `dataBuffer - 8` wurde eliminiert, da sie den Zeiger außerhalb des allokierten Speicherblocks positioniert.
2. **Konsistente Speicherverwaltung:** Der ursprüngliche `dataBuffer`-Zeiger wird jetzt direkt verwendet.
3. **Größenkonsistenz:** Durch die Beibehaltung der originalen Zeigerposition wird ein sicheres Schreiben in `helper()` gewährleistet.

Diese Änderungen verhindern Buffer Overflows, ungültige Speicherzugriffe und garantieren eine konsistente Speicherverwaltung.

**A.4.4 Gemini 2.5 Pro Antwort TC3\_B**

Der gegebene C/C++-Code enthält mehrere kritische Speicherverwaltungsfehler, die zu Sicherheitslücken führen können. Hier die Analyse:

**Sicherheitsrelevante Speicherfehler:**

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	823	Falsche Zeigerarithmetik	Use-after-free	In Zeile 33 wird <code>data = dataBuffer - 8</code> gesetzt, was den Zeiger vor den allokierten Speicherbereich verschiebt. Dies führt später in <code>helper()</code> zu Zugriffen auf ungültigen Speicher.	Zeigerarithmetik vermeiden und immer auf den ursprünglich allokierten Block verweisen
2	122	Heap-basiertes Buffer-Overflow	Buffer Overflow	Der <code>wcsncpy()</code> -Aufruf in Zeile 16 kopiert bis zu <code>SIZE-1</code> Zeichen, aber der Zielpuffer <code>data</code> wurde durch die fehlerhafte Zeigerarithmetik verkürzt.	Korrekte Puffergröße sicherstellen und Zeigermanipulationen entfernen

Tabelle 32: Analyse sicherheitsrelevanter Speicherfehler TC3\_B Gemini 2.5 Pro

### Korrigierter Code:

```

1 void test() {
2 wchar_t* data = nullptr;
3 wchar_t* dataBuffer = static_cast<wchar_t*>(malloc(SIZE *
4 ↪ sizeof(wchar_t)));
5 if (!dataBuffer) exit(-1);
6
7 data = dataBuffer; // Korrekte Zuweisung
8 wmemset(data, L'A', SIZE - 1);
9 data[SIZE - 1] = L'\0';
10
11 helper(data);
12 }
```

Listing 35: Korrigierte Version des Codes für TC3\_B Gemini 2.5 Pro

Die kritischste Schwachstelle ist die fehlerhafte Zeigerarithmetik (CWE-823), die alle nachfolgenden Operationen kompromittiert. Der korrigierte Code entfernt die gefährliche Zeigermanipulation und stellt sicher, dass alle Speicheroperationen innerhalb der allokierten Grenzen bleiben.

## A.5 Testfall *TC4\_B*

### A.5.1 Prompt

Du bist ein Sicherheitsexperte mit Spezialisierung auf C/C++-Codeanalyse. Dein Ziel ist es, sicherheitsrelevante Speicherverwaltungsfehler zu erkennen und zu beheben. Analysiere den folgenden C/C++-Code ausschließlich im Hinblick auf sicherheitskritische Speicherfehler wie Buffer Overflows, Use-after-free oder Memory Leaks und behebe diese auch gleich falls der Code solche Fehler hat.

Erzeuge eine Tabelle in Latex mit den folgenden Spalten:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
-----	--------	------------------------	-----------	--------------	------------

Analysiere ausschließlich sicherheitsrelevante Aspekte, keine Stil- oder Architekturfragen.

```
```cpp
#define ENABLE_TEST_OUTPUT 1

#if ENABLE_TEST_OUTPUT
#include <sstream>
#include <iostream>
#include <codecvt>
#include <locale>

#define TEST_OUTPUT(x) do { \
    std::wstringstream wss; \
    wss << x; \
    std::wcout << "\t" << wss.str() << std::endl; \
} while (0)

#else
#define TEST_OUTPUT(x)
#endif

void helper(int& data, bool good) {
    if (good)
        data = 7;
    else if (!good)
        data = 1024;

    return;
}

void test() {
    int data;

    data = -1;
```

```
helper(data, false);
{
    int i;
    int buffer[10] = { 0 };

    if (data >= 0)
    {
        buffer[data] = 1;
        /* Print the array values */
        for (i = 0; i < 10; i++)
        {
            TEST_OUTPUT(buffer[i]);
        }
    }
    else
    {
        TEST_OUTPUT("ERROR: Array index is negative.");
    }
}
}
```

...

A.5.2 GPT-4o Antwort TC4_B

Hier ist die sicherheitstechnische Analyse des bereitgestellten Codes, konzentriert auf Buffer Overflows und verwandte Speicherfehler:

Befund:

Der Wert data kann auf 1024 gesetzt werden, wird aber als Index in ein Array der Größe 10 verwendet, ohne obere Schrankenprüfung. Das führt zu einem klassischen Out-of-Bounds Write (Buffer Overflow).

Korrigierter Codeausschnitt (Empfehlung implementiert):

```

1 if (data >= 0 && data < 10)
2 {
3     buffer[data] = 1;
4     for (i = 0; i < 10; i++)
5     {
6         TEST_OUTPUT(buffer[i]);
7     }
8 }
9 else
10 {
11     TEST_OUTPUT("ERROR: Array index is out-of-bounds.");
12 }

```

Listing 36: Korrigierter Codeausschnitt TC4_B ChatGPT-4o:

LaTeX-Tabelle zur Dokumentation:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-121	Stack-based Buffer Overflow	Buffer Overflow	Im Zweig if (data >= 0) kann der Wert data = 1024 außerhalb der Arraygrenze von buffer[10] geschrieben werden. Es fehlt eine Prüfung auf die obere Grenze, wodurch ein Out-of-Bounds Write entsteht.	Prüfung auf obere Grenze ergänzen: if (data >= 0 && data < 10), um das Schreiben außerhalb des Puffers zu verhindern.

Tabelle 33: Analyse sicherheitsrelevanter Speicherfehler TC4_B ChatGPT-4o

A.5.3 Sonnet 3.7 Antwort TC4_B

Ich werde den C/C++-Code auf sicherheitskritische Speicherfehler analysieren und eine Tabelle mit den identifizierten Schwachstellen erstellen.

Nach der Analyse des Codes habe ich einen kritischen Speicherfehler gefunden:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-119	Buffer Overflow	Out-of-bounds Write	In der Funktion <code>test()</code> wird <code>buffer[data]</code> geschrieben, wobei <code>data</code> den Wert 1024 haben kann, was außerhalb des definierten Arrays mit 10 Elementen liegt.	Vor dem Zugriff auf das Array sollte eine Bereichsprüfung durchgeführt werden: <code>if (data >= 0 && data < 10)</code>

Tabelle 34: Analyse sicherheitsrelevanter Speicherfehler TC4_B Sonnet 3.7

Detaillierte Analyse:

Der Hauptfehler im Code ist ein potenzieller Buffer Overflow in der Funktion `test()`. Hier ist der problematische Abschnitt:

```

1 void helper(int& data, bool good) {
2     if (good)
3         data = 7;
4     else if (!good)
5         data = 1024; // Hier wird data auf 1024 gesetzt
6
7     return;
8 }
9
10 void test() {
11     int data;
12
13     data = -1;
14     helper(data, false); // data erhält den Wert 1024
15     {
16         int i;
17         int buffer[10] = { 0 }; // Array mit nur 10 Elementen
18
19         if (data >= 0)
20         {
21             buffer[data] = 1; // Zugriff auf buffer[1024], was
22                               // ↳ weit außerhalb des Arrays liegt
23             /* ... */
24         }
25         /* ... */
26     }
27 }
```

Listing 37: Problematischer Codeabschnitt TC4_B Sonnet 3.7

Der Fehler entsteht, weil:

1. Die Variable `data` wird in `helper()` auf 1024 gesetzt, wenn `good` `false` ist.
2. In `test()` wird `helper(data, false)` aufgerufen, was `data` auf 1024 setzt.
3. Das Array `buffer` hat nur 10 Elemente (Indizes 0–9).
4. Der Code prüft nur, ob `data >= 0` ist, nicht aber, ob `data < 10`.

5. Dadurch erfolgt ein Schreibzugriff auf `buffer[1024]`, was weit außerhalb des gültigen Bereichs liegt.

Korrigierter Code:

Hier ist der korrigierte Code mit der Behebung des Buffer Overflow-Problems:

```

1 void test() {
2     int data;
3
4     data = -1;
5     helper(data, false);
6     {
7         int i;
8         int buffer[10] = { 0 };
9
10        if (data >= 0 && data < 10) // Korrekte Bereichsprüfung
11            ↪ hinzugefügt
12        {
13            buffer[data] = 1;
14            /* Print the array values */
15            for (i = 0; i < 10; i++)
16            {
17                TEST_OUTPUT(buffer[i]);
18            }
19        }
20        else
21        {
22            TEST_OUTPUT("ERROR: Array index is out of bounds.");
23        }
24    }
25 }
```

Listing 38: Korrigierter Code TC4_B Sonnet 3.7

Diese Korrektur stellt sicher, dass auf das Array nur innerhalb seiner definierten Grenzen zugegriffen wird, wodurch der Buffer Overflow verhindert wird.

A.5.4 Gemini 2.5 Pro Antwort TC4_B

Als Sicherheitsexperte habe ich den C/C++-Code auf sicherheitsrelevante Speicherverwaltungsfehler analysiert und behebe diese gegebenenfalls. Hier ist die Analyse in Tabellenform:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-129	Unzulässige Array-Indizierung	Buffer Overflow	Die Funktion helper kann den Wert von data auf 1024 setzen, was zu einem Buffer Overflow in buffer[data] führt, da buffer nur 10 Elemente hat.	Überprüfen Sie die Array-Grenzen vor dem Zugriff auf das Array, um einen Buffer Overflow zu vermeiden.

Tabelle 35: Analyse sicherheitsrelevanter Speicherfehler TC4_B Gemini 2.5 Pro

Behebung des Codes:

Um den Buffer Overflow zu beheben, muss der Wert von data vor dem Zugriff auf das Array buffer überprüft werden. Hier ist der korrigierte Code:

```
1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>
7 #include <locale>
8
9 #define TEST_OUTPUT(x) do { \
10     std::wstringstream wss; \
11     wss << x; \
12     std::wcout << "\t" << wss.str() << std::endl; \
13 } while (0)
14
15 #else
16 #define TEST_OUTPUT(x)
17 #endif
18
19 void helper(int& data, bool good) {
20     if (good)
21         data = 7;
22     else if (!good)
23         data = 9; // Korrigierter Wert, um Buffer Overflow zu
24                 ↪ verhindern
25
26     return;
27 }
28
29 void test() {
30     int data;
31
32     data = -1;
33     helper(data, false);
34     {
35         int i;
36         int buffer[10] = { 0 };
37
38         if (data >= 0 && data < 10)
39         {
40             buffer[data] = 1;
41             /* Print the array values */
42             for (i = 0; i < 10; i++)
43             {
44                 TEST_OUTPUT(buffer[i]);
45             }
46         }
47     }
48     else
```

```
47     {  
48         TEST_OUTPUT("ERROR: Array index is out of bounds.");  
49     }  
50 }  
51 }
```

Listing 39: Korrigierter Code TC4_B Gemini 2.5 Pro

Änderungen:

- In der Funktion `helper` wurde der Wert, der `data` zugewiesen wird, wenn `good` `false` ist, von 1024 auf 9 geändert. Dies stellt sicher, dass `data` immer innerhalb der Grenzen des Arrays `buffer` liegt.
- In der Funktion `test` wurde eine zusätzliche Überprüfung hinzugefügt, um sicherzustellen, dass `data` nicht nur nicht negativ ist, sondern auch kleiner als die Größe des Arrays `buffer`, bevor auf `buffer[data]` zugegriffen wird. Wenn `data` außerhalb der Grenzen liegt, wird eine Fehlermeldung ausgegeben.

A.6 Testfall *TC5_B*

A.6.1 Prompt

Du bist ein Sicherheitsexperte mit Spezialisierung auf C/C++-Codeanalyse. Dein Ziel ist es, sicherheitsrelevante Speicherverwaltungsfehler zu erkennen und zu beheben. Analysiere den folgenden C/C++-Code ausschließlich im Hinblick auf sicherheitskritische Speicherfehler wie Buffer Overflows, Use-after-free oder Memory Leaks und behebe diese auch gleich falls der Code solche Fehler hat.

Erzeuge eine Tabelle in Latex mit den folgenden Spalten:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
-----	--------	------------------------	-----------	--------------	------------

Analysiere ausschließlich sicherheitsrelevante Aspekte, keine Stil- oder Architekturfragen.

```
```cpp
#define ENABLE_TEST_OUTPUT 1

#if ENABLE_TEST_OUTPUT
#include <sstream>
#include <iostream>
#include <codecvt>
#include <locale>

#define TEST_OUTPUT(x) do { \
 std::wstringstream wss; \
 wss << x; \
 std::wcout << "\\t" << wss.str() << std::endl; \
} while (0)

#else
#define TEST_OUTPUT(x)
#endif

void test() {
 char* data;
 char*& dataRef = data;
 char dataBadBuffer[sizeof(OneIntClass)];

 data = dataBadBuffer;
 {
 char* data = dataRef;
 {
 TwoIntsClass* classTwo = new(data) TwoIntsClass;
 classTwo->intOne = 5;
```

```
 classTwo->intTwo = 10;
 std::wcout << sizeof(OneIntClass) << std::endl;
 std::wcout << sizeof(TwoIntsClass) << std::endl;
 TEST_OUTPUT("Zahl 1: " << classTwo->intOne);
 TEST_OUTPUT("Zahl 2: " << classTwo->intTwo);
}
}
}

class OneIntClass {
public:
 int intOne;
};

class TwoIntsClass
{
public:
 int intOne;
 int intTwo;
};
...
```

### A.6.2 GPT-4o Antwort TC5\_B

Hier ist die sicherheitsfokussierte Analyse des gegebenen C++-Codes. Der relevante Fehler ist ein klassischer Stack-basierter Buffer Overflow, verursacht durch das Platzieren eines größeren Objekts in einem zu kleinen statischen Speicherbereich mithilfe von Placement `new`.

#### Analyseergebnisse in LaTeX:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-121	Stack-based Buffer Overflow	Buffer Overflow	Im Stack wird ein Puffer <code>dataBadBuffer</code> mit der Größe von <code>sizeof(OneIntClass)</code> alloziert. Anschließend wird mit Placement <code>new</code> ein Objekt vom Typ <code>TwoIntsClass</code> darin erzeugt, das größer ist. Dies überschreibt benachbarte Speicherbereiche im Stack und verursacht einen klassischen Buffer Overflow.	Stellen Sie sicher, dass der Zielpuffer ausreichend groß ist für das zu konstruierende Objekt. Verwenden Sie z.B. <code>char dataBadBuffer[sizeof(TwoIntsClass)]</code> oder verzichten Sie auf Placement <code>new</code> und verwenden Sie <code>new</code> <code>TwoIntsClass</code> für eine heapbasierte Allokation.

Tabelle 36: Analyse sicherheitsrelevanter Speicherfehler TC5\_B GPT-4o

#### Fix Vorschlag im Code:

```
1 char dataBadBuffer[sizeof(TwoIntsClass)]; // Fix: ausreichend groß
 ↳ er Puffer
```

Listing 40: Korrigierter Codeausschnitt TC5\_B GPT-4o:

#### Optional (sicherer und klarer):

```
1 TwoIntsClass* classTwo = new TwoIntsClass;
2 ...
3 delete classTwo;
```

Listing 41: Alternativer Codeausschnitt TC5\_B GPT-4o:

Damit wird auch das Risiko durch manuelle Speicherlayout-Verletzungen reduziert. Wenn du möchtest, kann ich dir auch eine vollständig überarbeitete Version des Codes mit Fix anzeigen.

### A.6.3 Sonnet 3.7 Antwort TC5\_B

Nach einer detaillierten Analyse des vorgelegten C++-Codes habe ich mehrere sicherheitskritische Speicherverwaltungsfehler identifiziert. Hier ist meine Bewertung der Schwachstellen:



Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-119	Buffer Overflow	Heap/Stack Overflow	Der Puffer <code>dataBadBuffer</code> ist für <code>OneIntClass</code> dimensioniert, wird aber für <code>TwoIntsClass</code> verwendet, was zu einem Pufferüberlauf führt, da <code>TwoIntsClass</code> größer ist.	Puffergröße an die tatsächlich verwendete Klasse anpassen: <code>char dataBadBuffer[sizeof(TwoIntsClass)];</code>
2	CWE-758	Undefined Behavior	Falsche Objektinitialisierung	Verwendung von <code>placement new</code> ohne Sicherstellung der korrekten Ausrichtung (Alignment) des Speichers.	<code>std::aligned_storage&lt;sizeof(TwoIntsClass), alignof(TwoIntsClass)&gt;::type buffer;</code> verwenden.
3	CWE-401	Memory Leak	Fehlender Destruktoraufruf	Bei Verwendung von <code>placement new</code> wird der Destruktor nicht automatisch aufgerufen.	Expliziten Destruktoraufruf hinzufügen: <code>classTwo-&gt;~TwoIntsClass();</code>
4	CWE-562	Return of Stack Variable Address	Dangling Reference	Die Variable <code>dataRef</code> referenziert einen lokalen Stack-Pointer, was nach Funktionsende zu einer ungültigen Referenz führt.	Speicher dynamisch allokalieren oder Lebensdauer sicherstellen.

Tabelle 37: Analyse sicherheitsrelevanter Speicherfehler TC5\_B Sonnet 3.7

### Korrigierter Code:

Hier ist der korrigierte Code, der die identifizierten Sicherheitsprobleme behebt:

```

1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>
7 #include <locale>
8 #include <memory> // Für std::aligned_storage
9
10 #define TEST_OUTPUT(x) do { \
11 std::wstringstream wss; \
12 wss << x; \
13 std::wcout << "\t" << wss.str() << std::endl; \
14 } while (0)
15
16 #else
17 #define TEST_OUTPUT(x)
18 #endif
19
20 // Forward-Deklarationen der Klassen
21 class OneIntClass {
22 public:
23 int intOne;
24 };
25
26 class TwoIntsClass

```

```
27 {
28 public:
29 int intOne;
30 int intTwo;
31 };
32
33 void test() {
34 // Korrekte Größe für den benötigten Typ verwenden
35 typename std::aligned_storage<sizeof(TwoIntsClass), alignof(
 ↪ TwoIntsClass)>::type buffer;
36
37 // Pointer auf den ausgerichteten Speicher
38 char* data = reinterpret_cast<char*>(&buffer);
39 char*& dataRef = data;
40
41 {
42 char* data = dataRef;
43 {
44 TwoIntsClass* classTwo = new(data) TwoIntsClass;
45 classTwo->intOne = 5;
46 classTwo->intTwo = 10;
47 std::wcout << sizeof(OneIntClass) << std::endl;
48 std::wcout << sizeof(TwoIntsClass) << std::endl;
49 TEST_OUTPUT("Zahl 1: " << classTwo->intOne);
50 TEST_OUTPUT("Zahl 2: " << classTwo->intTwo);
51
52 // Expliziter Aufruf des Destruktors
53 classTwo->~TwoIntsClass();
54 }
55 }
56 }
```

Listing 42: Korrigierter Codeausschnitt TC5\_B Sonnet 3.7:

### Erklärung der Korrekturen:

1. **Buffer Overflow behoben:** Die Puffergröße wurde von `sizeof(OneIntClass)` auf `sizeof(TwoIntsClass)` angepasst, um ausreichend Speicher für das größere Objekt bereitzustellen.
2. **Ausrichtungsproblem behoben:** `std::aligned_storage` wird verwendet, um korrekt ausgerichteten Speicher für den Typ `TwoIntsClass` zu gewährleisten.
3. **Memory Leak behoben:** Der Destruktor wird explizit aufgerufen, um Ressourcen ordnungsgemäß freizugeben.
4. **Dangling Reference vermieden:** Die Struktur wurde so angepasst, dass der Speicher korrekt verwaltet wird und keine ungültigen Referenzen entstehen.
5. **Forward Deklarationen hinzugefügt:** : Die Klassen werden vor ihrer Verwendung deklariert, um Kompilierungsprobleme zu vermeiden.

Diese Änderungen stellen sicher, dass der Code keine sicherheitskritischen Speicherverwaltungsfehler mehr enthält und die Speichernutzung den Best Practices entspricht.

#### A.6.4 Gemini 2.5 Pro Antwort TC5\_B

Als Sicherheitsexperte mit Spezialisierung auf C/C++-Codeanalyse konzentriere ich mich auf sicherheitsrelevante Speicherverwaltungsfehler. Nachfolgend findest du eine Analyse des Codes in Bezug auf sicherheitskritische Speicherfehler wie Pufferüberläufe, Use-after-free oder Speicherlecks.

Hier ist die Tabelle im LaTeX-Format mit den gefundenen Schwachstellen und Empfehlungen:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-131	Falsche Berechnung der Puffergröße	Stack-basierter Pufferüberlauf	Die Größe von <code>dataBadBuffer</code> ist <code>sizeof(OneIntClass)</code> , aber im Code wird ein Objekt vom Typ <code>TwoIntsClass</code> in diesen Speicherbereich platziert. Da <code>TwoIntsClass</code> größer als <code>OneIntClass</code> ist, führt dies zu einem Pufferüberlauf auf dem Stack.	Die Größe von <code>dataBadBuffer</code> muss mindestens so groß wie <code>sizeof(TwoIntsClass)</code> sein, um den Pufferüberlauf zu verhindern.
2	CWE-762	Fehlende Freigabe von Speicher nach Verwendung	Speicherleck	Obwohl der Speicher für das <code>TwoIntsClass</code> -Objekt im bereitgestellten Puffer platziert wird, wird der Speicher, der potenziell von dem platzierten Objekt gehalten wird, nicht explizit freigegeben oder zerstört. Dies kann zu Ressourcenverlusten führen, insbesondere wenn das Objekt Ressourcen wie dynamisch zugewiesenen Speicher besitzt.	Füge einen expliziten Destruktoraufruf für das platzierte Objekt hinzu, um sicherzustellen, dass alle zugehörigen Ressourcen freigegeben werden. Dies kann mit <code>classTwo-&gt;~TwoIntsClass();</code> vor dem Ende des Gültigkeitsbereichs von <code>classTwo</code> erreicht werden.

Tabelle 38: Analyse sicherheitsrelevanter Speicherfehler TC5\_B Gemini 2.5 Pro

#### Erläuterungen der Schwachstellen und Empfehlungen:

##### 1. CWE-131 - Falsche Berechnung der Puffergröße:

- **Fehlerart:** Stack-basierter Pufferüberlauf
- **Beschreibung:**
  - Der Code deklariert einen Puffer `dataBadBuffer` mit der Größe `sizeof(OneIntClass)`. Danach wird jedoch versucht, ein Objekt vom Typ `TwoIntsClass` in diesem Puffer zu platzieren. Da `TwoIntsClass` zwei `int` enthält, während `OneIntClass` nur einen enthält, ist `TwoIntsClass` größer. Das Platzieren eines größeren Objekts in einem kleineren Puffer führt zu einem Pufferüberlauf, was bedeutet, dass der Code über das Ende des reservierten Speicherbereichs hinaus schreibt.
- **Empfehlung:**
  - Um diesen Fehler zu beheben, muss sichergestellt werden, dass der Puffer `dataBadBuffer` groß genug ist, um das Objekt vom Typ `TwoIntsClass`

aufzunehmen. Dies kann erreicht werden, indem die Größe des Puffers auf `sizeof(TwoIntsClass)` gesetzt wird.

## 2. CWE-762: Fehlende Freigabe von Speicher nach Verwendung:

- **Fehlerart:** Speicherleck

- **Beschreibung:**

- Im Code wird `Placement-new` verwendet, um ein Objekt vom Typ `TwoIntsClass` in einem zuvor zugewiesenen Speicherbereich zu erstellen. Obwohl `Placement-new` selbst keinen neuen Speicher zuweist, initialisiert es ein Objekt in dem vorhandenen Speicher. Wenn das Objekt, das in diesem Speicher platziert wird, Ressourcen wie dynamisch zugewiesenen Speicher oder Dateihandles besitzt, sollten diese Ressourcen freigegeben werden, bevor der Speicherbereich ungültig wird oder überschrieben wird. Im vorliegenden Fall wird das Objekt `classTwo` jedoch nicht explizit zerstört, bevor der zugehörige Speicherbereich ungültig wird.

- **Empfehlung:**

- Um sicherzustellen, dass alle Ressourcen, die von dem mit `Placement-new` erstellten Objekt gehalten werden, ordnungsgemäß freigegeben werden, sollte der Destruktor des Objekts explizit aufgerufen werden, bevor der Speicherbereich freigegeben oder überschrieben wird. Dies kann mit `classTwo->~TwoIntsClass();` erreicht werden. Dieser Aufruf stellt sicher, dass alle Aufräumarbeiten, die der Destruktor durchführt (z. B. das Freigeben von Speicher), ausgeführt werden, bevor der Speicherbereich wiederverwendet wird.

### Korrigierter Code:

Hier ist der korrigierte Code, der die oben genannten Empfehlungen umsetzt:

```
1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>
7 #include <locale>
8
9 #define TEST_OUTPUT(x) do { \
10 std::wstringstream wss; \
11 wss << x; \
12 std::wcout << "\t" << wss.str() << std::endl; \
13 } while (0)
14
15 #else
16 #define TEST_OUTPUT(x)
17 #endif
```

```
18
19 void test() {
20 char* data;
21 char*& dataRef = data;
22 char dataBadBuffer[sizeof(TwoIntsClass)]; // Korrektur: Größe
 ↳ angepasst
23
24 data = dataBadBuffer;
25 {
26 char* data = dataRef;
27 {
28 TwoIntsClass* classTwo = new(data) TwoIntsClass;
29 classTwo->intOne = 5;
30 classTwo->intTwo = 10;
31 std::wcout << sizeof(OneIntClass) << std::endl;
32 std::wcout << sizeof(TwoIntsClass) << std::endl;
33 TEST_OUTPUT("Zahl 1: " << classTwo->intOne);
34 TEST_OUTPUT("Zahl 2: " << classTwo->intTwo);
35
36 classTwo->~TwoIntsClass(); // Korrektur: Destruktor
 ↳ aufrufen
37 }
38 }
39 }
40
41 class OneIntClass {
42 public:
43 int intOne;
44 };
45
46 class TwoIntsClass
47 {
48 public:
49 int intOne;
50 int intTwo;
51
52 ~TwoIntsClass() {
53 // Hier Ressourcen freigeben, falls vorhanden
54 }
55 };
```

Listing 43: Korrigierter Codeausschnitt TC5\_B Gemini 2.5 Pro:

## A.7 Testfall *TC6\_B*

### A.7.1 Prompt

Du bist ein Sicherheitsexperte mit Spezialisierung auf C/C++-Codeanalyse. Dein Ziel ist es, sicherheitsrelevante Speicherverwaltungsfehler zu erkennen und zu beheben. Analysiere den folgenden C/C++-Code ausschließlich im Hinblick auf sicherheitskritische Speicherfehler wie Buffer Overflows, Use-after-free oder Memory Leaks und behebe diese auch gleich falls der Code solche Fehler hat.

Erzeuge eine Tabelle in Latex mit den folgenden Spalten:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
-----	--------	------------------------	-----------	--------------	------------

Analysiere ausschließlich sicherheitsrelevante Aspekte, keine Stil- oder Architekturfragen.

```
```cpp
#define ENABLE_TEST_OUTPUT 1

#if ENABLE_TEST_OUTPUT
#include <sstream>
#include <iostream>
#include <codecvt>
#include <locale>

#define TEST_OUTPUT(x) do { \
    std::wstringstream wss; \
    wss << x; \
    std::wcout << "\t" << wss.str() << std::endl; \
} while (0)

#else
#define TEST_OUTPUT(x)
#endif

void test() {
    wchar_t* data;
    data = NULL;
    if (GLOBAL_CONST_FIVE == 5)
    {
        data = (wchar_t*)malloc(100 * sizeof(wchar_t));
        if (data == NULL) { exit(-1); }
        wmemset(data, L'A', 100 - 1);
        data[100 - 1] = L'\0';
        free(data);
    }
    if (GLOBAL_CONST_FIVE == 5)
    {
```

```
        TEST_OUTPUT(data);  
    }  
}  
...
```

A.7.2 GPT-4o Antwort TC6_B

Hier ist die sicherheitstechnische Analyse des Codes mit Fokus auf Speicherverwaltungsfehler. Der zentrale Fehler liegt in der Verwendung eines bereits freigegebenen Zeigers (`data`) – ein klassischer Use-after-free.

Analyseergebnis in LaTeX-Tabellenform:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-416	Use After Free	Dangling Pointer	Der Speicherbereich, auf den <code>data</code> zeigt, wird mit <code>free(data)</code> ; freigegeben, danach jedoch in <code>TEST_OUTPUT(data)</code> ; weiterverwendet. Dies kann zu undefiniertem Verhalten führen, da der Speicher nicht mehr gültig ist.	Nach dem Aufruf von <code>free(data)</code> ; sollte der Zeiger auf <code>nullptr</code> gesetzt werden und vor der Nutzung auf Gültigkeit überprüft werden. Alternativ: Datenverwendung vor dem <code>free()</code> .

Tabelle 39: Analyse sicherheitsrelevanter Speicherfehler TC6_B GPT-4o

Vorschlag zur Behebung:

```

1 void test() {
2     wchar_t* data = nullptr;
3     if (GLOBAL_CONST_FIVE == 5)
4     {
5         data = (wchar_t*)malloc(100 * sizeof(wchar_t));
6         if (data == NULL) { exit(-1); }
7         wmemset(data, L'A', 100 - 1);
8         data[100 - 1] = L'\0';
9     }
10    if (GLOBAL_CONST_FIVE == 5)
11    {
12        TEST_OUTPUT(data);
13    }
14    // Speicher erst nach der Nutzung freigeben
15    free(data);
16    data = nullptr;
17 }

```

Listing 44: Korrigierter Codeausschnitt TC6_B ChatGPT 4o:

Dieser Fix stellt sicher, dass der Speicherzugriff nur bei gültigen Speicheradressen erfolgt und ein Use-after-free-Fehler vermieden wird.

A.7.3 Sonnet 3.7 Antwort TC6_B

Als Sicherheitsexperte für C/C++-Codeanalyse habe ich den bereitgestellten Code auf sicherheitsrelevante Speicherverwaltungsfehler untersucht. Der Code enthält tatsächlich eine kritische Sicherheitslücke.

Der Hauptfehler im Code ist ein klassischer Use-after-free-Fehler. Im Code wird ein Speicherbereich für `data` allokiert, dann mit Daten gefüllt und anschließend freigegeben.

Danach wird jedoch versucht, auf diesen bereits freigegebenen Speicherbereich zuzugreifen, was zu undefiniertem Verhalten führt und ein erhebliches Sicherheitsrisiko darstellt.

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-416	Use-after-free	Speicherzugriffsfehler	Nach dem Aufruf von <code>free(data)</code> wird in der zweiten if-Abfrage auf <code>data</code> zugegriffen, obwohl der Speicher bereits freigegeben wurde.	Den Zugriff auf <code>data</code> nach <code>free(data)</code> vermeiden oder <code>data</code> auf <code>NULL</code> setzen und vor der Verwendung prüfen.

Tabelle 40: Analyse sicherheitsrelevanter Speicherfehler TC6_B Sonnet 3.7

Hier ist der korrigierte Code:

```

1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>
7 #include <locale>
8
9 #define TEST_OUTPUT(x) do { \
10     std::wstringstream wss; \
11     wss << x; \
12     std::wcout << "\t" << wss.str() << std::endl; \
13 } while (0)
14
15 #else
16 #define TEST_OUTPUT(x)
17 #endif
18
19 void test() {
20     wchar_t* data;
21     data = NULL;
22     if (GLOBAL_CONST_FIVE == 5)
23     {
24         data = (wchar_t*)malloc(100 * sizeof(wchar_t));
25         if (data == NULL) { exit(-1); }
26         wmemset(data, L'A', 100 - 1);
27         data[100 - 1] = L'\0';
28
29         // Hier können wir auf data zugreifen, bevor es freigegeben
30         ↪ wird
31         TEST_OUTPUT(data);
32
33         free(data);
34         data = NULL; // Nach dem Freigeben auf NULL setzen
35     }
36
37     // Zweite if-Abfrage entfernt oder mit Null-Prüfung versehen
38     if (GLOBAL_CONST_FIVE == 5 && data != NULL)

```

```

38 {
39     TEST_OUTPUT(data);
40 }
41 }

```

Listing 45: Korrigierter Codeausschnitt TC6_B Sonnet 3.7

Die Korrektur besteht aus zwei wesentlichen Änderungen:

1. Der Aufruf von `TEST_OUTPUT(data)` wurde vor die Freigabe des Speichers mit `free(data)` verschoben.
2. Nach dem Aufruf von `free(data)` wird `data` auf `NULL` gesetzt, um zu verhindern, dass versehentlich auf den freigegebenen Speicher zugegriffen wird.
3. In der zweiten if-Abfrage wurde eine Prüfung hinzugefügt, ob `data` nicht `NULL` ist, bevor darauf zugegriffen wird

Diese Änderungen beheben den Use-after-free-Fehler und verbessern die Sicherheit des Codes erheblich.

A.7.4 Gemini 2.5 Pro Antwort TC6_B

Als Sicherheitsexperte für C/C++-Code werde ich den bereitgestellten Code auf sicherheitsrelevante Speicherverwaltungsfehler analysieren und eine Tabelle der gefundenen Schwachstellen erstellen.

Hier ist die LaTeX-Tabelle mit den Ergebnissen meiner Analyse:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-415	Double Free	Speicherfehler	'data' wird freigegeben, aber im zweiten Block wird immer noch darauf zugegriffen.	Stellen Sie sicher, dass der Speicher nicht freigegeben wird, bevor er nicht mehr benötigt wird, oder setzen Sie den Zeiger nach dem Freigeben auf NULL.

Tabelle 41: Analyse sicherheitsrelevanter Speicherfehler TC6_B Gemini 2.5 Pro

- **Beschreibung:** Im ersten if-Block wird Speicher für `data` mit `malloc` allokiert, mit 'A's gefüllt und dann mit `free(data)` freigegeben. Im zweiten if-Block wird versucht, auf `data` zuzugreifen und den Inhalt über `TEST_OUTPUT` auszugeben. Da der Speicher bereits freigegeben wurde, führt dies zu einem Use-After-Free-Fehler.
- **Empfehlung:** Stellen Sie sicher, dass auf freigegebenen Speicher nicht mehr zugegriffen wird. Falls der Wert nach dem `free` noch benötigt wird, sollte der Speicher nicht freigegeben werden oder der Zeiger nach dem Freigeben auf `NULL` gesetzt werden.

A.8 Testfall *TC7_B*

A.8.1 Prompt

Du bist ein Sicherheitsexperte mit Spezialisierung auf C/C++-Codeanalyse. Dein Ziel ist es, sicherheitsrelevante Speicherverwaltungsfehler zu erkennen und zu beheben. Analysiere den folgenden C/C++-Code ausschließlich im Hinblick auf sicherheitskritische Speicherfehler wie Buffer Overflows, Use-after-free oder Memory Leaks und behebe diese auch gleich falls der Code solche Fehler hat.

Erzeuge eine Tabelle in Latex mit den folgenden Spalten:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
-----	--------	------------------------	-----------	--------------	------------

Analysiere ausschließlich sicherheitsrelevante Aspekte, keine Stil- oder Architekturfragen.

```
```cpp
#define ENABLE_TEST_OUTPUT 1

#if ENABLE_TEST_OUTPUT
#include <sstream>
#include <iostream>
#include <codecvt>
#include <locale>

#define TEST_OUTPUT(x) do { \
 std::wstringstream wss; \
 wss << x; \
 std::wcout << "\t" << wss.str() << std::endl; \
} while (0)

#else
#define TEST_OUTPUT(x)
#endif

void helper_1(int*& data) {
 data = (int*)malloc(100 * sizeof(int));
 if (data == NULL) { exit(-1); }
 {
 size_t i;
 for (i = 0; i < 100; i++)
 {
 data[i] = 5;
 }
 }

 free(data);
}
```

```
void test(bool flag) {
 int* data;
 data = NULL;
 helper_1(data);
 TEST_OUTPUT(data[0]);
}
...
```

## A.8.2 GPT-4o Antwort TC7\_B

Hier ist die sicherheitsrelevante Analyse des Codes im Hinblick auf Speicherverwaltungsfehler. Der relevante Fehler entsteht durch die Nutzung eines bereits freigegebenen Speicherbereichs, also ein Use-after-free.

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-416	Use After Free	Dangling Pointer	Die Funktion <code>helper_1</code> allokiert Speicher für ein Integer-Array, füllt es und gibt den Speicher anschließend wieder frei. Danach wird im Aufrufer ( <code>test</code> ) auf <code>data[0]</code> zugegriffen, obwohl der Speicherbereich bereits mit <code>free</code> freigegeben wurde. Dies führt zu undefiniertem Verhalten und kann sicherheitskritisch sein.	Der Zugriff auf <code>data[0]</code> muss vor dem <code>free(data)</code> erfolgen. Alternativ kann die Speicherfreigabe aus <code>helper_1</code> entfernt und im Aufrufer nach dem Zugriff durchgeführt werden. Zusätzlich sollte <code>data</code> nach dem <code>free</code> auf <code>nullptr</code> gesetzt werden.

Tabelle 42: Analyse sicherheitsrelevanter Speicherfehler TC7\_B GPT-4o

### Behebt Version des Codes:

```

1 void helper_1(int*& data) {
2 data = (int*)malloc(100 * sizeof(int));
3 if (data == NULL) { exit(-1); }
4 for (size_t i = 0; i < 100; i++) {
5 data[i] = 5;
6 }
7 // Speicherfreigabe wird verschoben in test()
8 }
9
10 void test(bool flag) {
11 int* data = NULL;
12 helper_1(data);
13 TEST_OUTPUT(data[0]); // Zugriff erfolgt nun vor der Freigabe
14 free(data); // Speicher wird jetzt hier sicher
15 ↪ freigegeben
16 data = nullptr; // Dangling Pointer verhindern
17 }

```

Listing 46: Korrigierter Codeausschnitt TC7\_B ChatGPT 4o:

## A.8.3 Sonnet 3.7 Antwort TC7\_B

Als Sicherheitsexperte für C/C++-Codeanalyse habe ich den vorliegenden Code auf sicherheitsrelevante Speicherverwaltungsfehler untersucht. Ich habe mehrere kritische Probleme identifiziert, die ich im Folgenden detailliert erläutern werde.

### Identifizierte Sicherheitsprobleme:

Der Code enthält einen schwerwiegenden Use-after-free-Fehler, der zu unvorhersehbarem Verhalten, Programmabstürzen oder sogar zur Ausführung von Schadcode führen kann.

In der Funktion `helper_1` wird Speicher für 100 Integer-Werte allokiert und mit dem Wert 5 initialisiert. Allerdings wird dieser Speicher am Ende der Funktion durch den Aufruf von `free(data)` wieder freigegeben. Das Problem entsteht, wenn in der `test`-Funktion anschließend auf `data` zugegriffen wird, obwohl der Speicher bereits freigegeben wurde. Hier ist eine Tabelle mit den identifizierten Sicherheitsproblemen:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-416	Use After Free	Speicherzugriffsfehler	In <code>helper_1</code> wird der Speicher mit <code>free()</code> freigegeben, aber in <code>test</code> wird danach noch auf <code>data</code> zugegriffen, was zu undefiniertem Verhalten führt.	Speicher erst nach der letzten Verwendung freigegeben oder alternativ <code>data = nullptr;</code> setzen und vor jeder Nutzung prüfen.

Tabelle 43: Analyse sicherheitsrelevanter Speicherfehler TC7\_B Sonnet 3.7

### Korrigierter Code:

Hier ist der korrigierte Code, der das Use-after-free-Problem behebt:

```

1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>
7 #include <locale>
8
9 #define TEST_OUTPUT(x) do { \
10 std::wstringstream wss; \
11 wss << x; \
12 std::wcout << "\t" << wss.str() << std::endl; \
13 } while (0)
14
15 #else
16 #define TEST_OUTPUT(x)
17 #endif
18
19 void helper_1(int*& data) {
20 data = (int*)malloc(100 * sizeof(int));
21 if (data == NULL) { exit(-1); }
22 {
23 size_t i;
24 for (i = 0; i < 100; i++)
25 {
26 data[i] = 5;
27 }
28 }
29 // free(data) wurde entfernt, da der Speicher nach der Funktion
30 ↪ noch verwendet wird
31 }

```

```
32 void test(bool flag) {
33 int* data;
34 data = NULL;
35 helper_1(data);
36 TEST_OUTPUT(data[0]);
37
38 // Speicher nach der Verwendung freigeben
39 free(data);
40 data = NULL; // Setze Zeiger auf NULL nach der Freigabe
41 }
```

Listing 47: Korrigierter Codeausschnitt TC7\_B Sonnet 3.7:

### Erklärung der Korrektur

1. Der `free(data)`-Aufruf wurde aus der `helper_1`-Funktion entfernt, da der Speicher nach dem Rücksprung in die `test`-Funktion noch verwendet wird.
2. Stattdessen wurde der `free(data)`-Aufruf in die `test`-Funktion verschoben, und zwar nach der letzten Verwendung des Speichers.
3. Nach der Freigabe wird der Zeiger auf `NULL` gesetzt, um versehentliche weitere Zugriffe zu verhindern.

Diese Änderungen stellen sicher, dass kein Zugriff auf bereits freigegebenen Speicher erfolgt und dass der Speicher ordnungsgemäß freigegeben wird, um Memory Leaks zu vermeiden.

### Zusätzliche Empfehlungen

- Verwenden Sie moderne C++-Konstrukte wie `std::vector` oder `std::unique_ptr` anstelle von direkten `malloc` / `free`-Aufrufen, um Speicherverwaltungsfehler zu vermeiden.
- Implementieren Sie eine konsistente Fehlerbehandlung anstelle von direkten `exit`-Aufrufen.
- Erwägen Sie die Verwendung von statischen Codeanalysewerkzeugen wie `Valgrind` oder `AddressSanitizer`, um solche Probleme frühzeitig zu erkennen.

#### A.8.4 Gemini 2.5 Pro Antwort TC7\_B

Als Sicherheitsexperte mit Fokus auf C/C++-Codeanalyse werde ich den bereitgestellten Code auf sicherheitsrelevante Speicherverwaltungsfehler untersuchen und eine Tabelle mit den gefundenen Schwachstellen erstellen.

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-416	Use-after-free	Speicherverwaltungsfehler	In der Funktion <code>test</code> wird der Speicher, der von <code>data</code> referenziert wird, in <code>helper_1</code> freigegeben, aber danach wird in <code>test</code> noch darauf zugegriffen.	Stelle sicher, dass auf freigegebenen Speicher nicht mehr zugegriffen wird. Entferne den Zugriff <code>data[0]</code> oder verschiebe den <code>free</code> Aufruf.

Tabelle 44: Analyse sicherheitsrelevanter Speicherfehler TC7\_B Gemini 2.5 Pro

## Analyse und Behebung:

Der kritische Fehler in dem Code ist ein „Use-after-free“-Fehler.

### 1. Use-after-free:

- **CWE-ID:** CWE-416
- **Beschreibung:** In der Funktion `helper_1` wird Speicher für `data` allokiert und initialisiert. Danach wird der Speicher freigegeben. In der Funktion `test` wird `helper_1` aufgerufen, wodurch `data` Speicher zugewiesen und sofort wieder freigegeben wird. Direkt danach wird aber noch auf den freigegebenen Speicher mit `TEST_OUTPUT(data)` zugegriffen.
- **Empfehlung:** Um diesen Fehler zu beheben, sollte der Zugriff auf `data` in der Funktion `test` entfernt oder vor den `free(data)`-Aufruf in `helper_1` verschoben werden. Da die Variable `data` nach dem Aufruf von `helper_1` nicht mehr benötigt wird, ist es am einfachsten, die Zeile `TEST_OUTPUT(data);` zu entfernen.

## Korrigierter Code:

```

1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>
7 #include <locale>
8
9 #define TEST_OUTPUT(x) do { \
10 std::wstringstream wss; \
11 wss << x; \
12 std::wcout << "\t" << wss.str() << std::endl; \
13 } while (0)
14
15 #else
16 #define TEST_OUTPUT(x)
17 #endif
18
19 void helper_1(int*& data) {
20 data = (int*)malloc(100 * sizeof(int));
21 if (data == NULL) { exit(-1); }

```



```
22 {
23 size_t i;
24 for (i = 0; i < 100; i++)
25 {
26 data[i] = 5;
27 }
28 }
29
30 free(data);
31 }
32
33 void test(bool flag) {
34 int* data;
35 data = NULL;
36 helper_1(data);
37 // TEST_OUTPUT(data[0]); // Entfernt, um Use-after-free zu
 ↪ vermeiden
38 }
```

Listing 48: Korrigierter Codeausschnitt TC7\_B Gemini 2.5 Pro

## A.9 Testfall *TC8\_B*

### A.9.1 Prompt

Du bist ein Sicherheitsexperte mit Spezialisierung auf C/C++-Codeanalyse. Dein Ziel ist es, sicherheitsrelevante Speicherverwaltungsfehler zu erkennen und zu beheben. Analysiere den folgenden C/C++-Code ausschließlich im Hinblick auf sicherheitskritische Speicherfehler wie Buffer Overflows, Use-after-free oder Memory Leaks und behebe diese auch gleich falls der Code solche Fehler hat.

Erzeuge eine Tabelle in Latex mit den folgenden Spalten:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
-----	--------	------------------------	-----------	--------------	------------

Analysiere ausschließlich sicherheitsrelevante Aspekte, keine Stil- oder Architekturfragen.

```
```cpp
#define ENABLE_TEST_OUTPUT 1

#if ENABLE_TEST_OUTPUT
#include <sstream>
#include <iostream>
#include <codecvt>
#include <locale>

#define TEST_OUTPUT(x) do { \
    std::wstringstream wss; \
    wss << x; \
    std::wcout << "\t" << wss.str() << std::endl; \
} while (0)

#else
#define TEST_OUTPUT(x)
#endif

class Helper
{
public:
    Helper()
    {
        name = NULL;
    }

    Helper(const char* name)
    {
        if (name)
        {
            this->name = new char[strlen(name) + 1];
```

```
        strcpy(this->name, name);
    }
    else
    {
        this->name = new char[1];
        *(this->name) = '\\0';
    }
}

~Helper()
{
    delete[] name;
}

Helper(Helper& helperObject)
{
    this->name = new char[strlen(helperObject.name) + 1];
    strcpy(this->name, helperObject.name);
}

Helper& operator=(const Helper& helperObject)
{
    delete[] this->name;
    this->name = new char[strlen(helperObject.name) + 1];
    strcpy(this->name, helperObject.name);
    return *this;
}

char* name;
};

void test() {
    Helper helperObject("Helper");
    helperObject = helperObject;
    TEST_OUTPUT(helperObject.name);
}
...
```

A.9.2 GPT-4o Antwort TC8_B

Der gegebene C++-Code enthält eine sicherheitskritische Schwachstelle im Zusammenhang mit Selbstzurodnung (self-assignment) im benutzerdefinierten Zuweisungsoperator. Dies kann zu Use-after-free führen, wenn das Objekt sich selbst zugewiesen wird. Der Speicher wird gelöscht, bevor geprüft wird, ob die Quelle dieselbe ist wie das Ziel.

Sicherheitsanalyse (LaTeX-Tabelle):

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-416	Use After Free	Selbstzurodnung ohne Prüfung	Im benutzerdefinierten Assignment-Operator erfolgt ein <code>delete[]</code> auf <code>name</code> , auch wenn das Objekt sich selbst zugewiesen wird (<code>helperObject = helperObject;</code>). Dies führt zu einem Use-after-free beim anschließenden <code>strcpy</code> .	Vor dem Löschen sollte geprüft werden, ob das Objekt sich selbst zugewiesen wird: <code>if (this != &helperObject)</code> .

Tabelle 45: Analyse sicherheitsrelevanter Speicherfehler TC8_B ChatGPT-4o

Verbesserter Zuweisungsoperator (Fix):

```

1 Helper& operator=(const Helper& helperObject)
2 {
3     if (this != &helperObject) {
4         delete[] this->name;
5         this->name = new char[strlen(helperObject.name) + 1];
6         strcpy(this->name, helperObject.name);
7     }
8     return *this;
9 }

```

Listing 49: Korrigierter Codeausschnitt TC8_B ChatGPT-4o

Mit dieser Anpassung wird verhindert, dass sich das Objekt versehentlich selbst löscht und anschließend auf ungültigen Speicher zugreift. Dadurch wird die potenziell schwerwiegende Sicherheitslücke zuverlässig geschlossen.

A.9.3 Sonnet 3.7 Antwort TC8_B

Nach einer gründlichen Analyse des C++-Codes habe ich mehrere sicherheitsrelevante Speicherverwaltungsfehler identifiziert. Hier ist eine detaillierte Aufstellung der gefundenen Probleme:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-401	Memory Leak	Ressourcenleck	Im Standardkonstruktor wird name auf NULL gesetzt, aber nicht initialisiert, was zu undefiniertem Verhalten führen kann.	Initialisieren Sie name mit einem leeren String oder verwenden Sie einen konsistenten Ansatz wie im parametrisierten Konstruktor.
2	CWE-416	Use-After-Free	Speicherzugriffsfehler	In der Zuweisungsoperator-Methode wird this->name gelöscht, bevor geprüft wird, ob this und helperObject identisch sind (Selbstzuweisung).	Implementieren Sie einen Selbstzuweisungstest am Anfang der Methode.
3	CWE-476	NULL Pointer Dereference	Speicherzugriffsfehler	Im Copy-Konstruktor wird nicht geprüft, ob helperObject.name NULL ist, bevor strlen() aufgerufen wird.	Fügen Sie eine NULL-Prüfung hinzu, bevor auf helperObject.name zugegriffen wird.
4	CWE-476	NULL Pointer Dereference	Speicherzugriffsfehler	Im Zuweisungsoperator wird nicht geprüft, ob helperObject.name NULL ist.	Fügen Sie eine NULL-Prüfung hinzu, bevor auf helperObject.name zugegriffen wird.

Tabelle 46: Analyse sicherheitsrelevanter Speicherfehler TC8_B Sonnet 3.7

Korrigierter Code:

```

1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>
7 #include <locale>
8
9 #define TEST_OUTPUT(x) do { \
10     std::wstringstream wss; \
11     wss << x; \
12     std::wcout << "\t" << wss.str() << std::endl; \
13 } while (0)
14
15 #else
16 #define TEST_OUTPUT(x)
17 #endif
18
19 class Helper
20 {
21 public:
22     Helper()
23     {
24         // Problem 1: Korrektur - Initialisierung mit leerem String
25         name = new char[1];
26         name[0] = '\0';
27     }
28
29     Helper(const char* name)

```

```
30 {
31     if (name)
32     {
33         this->name = new char[strlen(name) + 1];
34         strcpy(this->name, name);
35     }
36     else
37     {
38         this->name = new char[1];
39         *(this->name) = '\\0';
40     }
41 }
42
43 ~Helper()
44 {
45     delete[] name;
46 }
47
48 // Problem 3: Korrektur - NULL-Prüfung hinzugefügt
49 Helper& Helper(const Helper& helperObject)
50 {
51     if (helperObject.name)
52     {
53         this->name = new char[strlen(helperObject.name) + 1];
54         strcpy(this->name, helperObject.name);
55     }
56     else
57     {
58         this->name = new char[1];
59         *(this->name) = '\\0';
60     }
61 }
62
63 // Problem 2 und 4: Korrektur - Selbstzuweisungstest und NULL-
64 // ↪ Prüfung
65 Helper& operator=(const Helper& helperObject)
66 {
67     // Selbstzuweisungstest
68     if (this == &helperObject)
69         return *this;
70
71     char* temp = nullptr;
72
73     // Sicheres Kopieren
74     if (helperObject.name)
75     {
76         temp = new char[strlen(helperObject.name) + 1];
77         strcpy(temp, helperObject.name);
78     }
79     else
80     {
81         this->name = new char[1];
82         *(this->name) = '\\0';
83     }
84 }
```

```

80     temp = new char[1];
81     temp[0] = '\\0';
82 }
83
84     // Erst nach erfolgreicher Allokation alten Speicher
85     ↪ freigeben
86     delete[] this->name;
87     this->name = temp;
88
89     return *this;
90 }
91
92 char* name;
93 };
94 void test() {
95     Helper helperObject("Helper");
96     helperObject = helperObject;
97     TEST_OUTPUT(helperObject.name);
98 }

```

Listing 50: Korrigierter Codeausschnitt TC8_B Sonnet 3.7

Die Korrekturen beheben die identifizierten Sicherheitsprobleme und verbessern die Robustheit des Codes gegen Speicherverwaltungsfehler.

A.9.4 Gemini 2.5 Pro Antwort TC8_B

Hier ist die Analyse des C/C++-Codes auf sicherheitsrelevante Speicherverwaltungsfehler, zusammen mit einer tabellarischen Übersicht im LaTeX-Format und den entsprechenden Korrekturen:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-415	Double Free	Speicherfehler	Im Zuweisungsoperator wird der Speicher von <code>this->name</code> freigegeben und neu zugewiesen. Wenn die Zuweisung auf dasselbe Objekt erfolgt (<code>helperObject = helperObject</code>), wird der Speicher freigegeben und anschließend versucht, den freigegebenen Speicher zu kopieren, was zu einem Double Free führen kann.	Füge eine Überprüfung auf Selbstzuweisung im Zuweisungsoperator hinzu, um zu vermeiden, dass der Speicher freigegeben wird, wenn das Objekt sich selbst zugewiesen wird.
2	CWE-476	Null Pointer Dereference	Speicherfehler	Der Standardkonstruktor initialisiert <code>name</code> mit NULL. Wenn <code>name</code> nicht später zugewiesen wird und dereferenziert wird, führt dies zu einem Null Pointer Dereference.	Stelle sicher, dass <code>name</code> immer auf einen gültigen Speicherbereich zeigt, bevor es dereferenziert wird. Initialisiere <code>name</code> mit einem leeren String anstatt NULL.

Tabelle 47: Analyse sicherheitsrelevanter Speicherfehler TC8_B Gemini 2.5 Pro

Korrigierter Code:

Hier ist der korrigierte Code, der die oben genannten Sicherheitslücken behebt:

```
1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>
7 #include <locale>
8
9 #define TEST_OUTPUT(x) do { \
10     std::wstringstream wss; \
11     wss << x; \
12     std::wcout << "\t" << wss.str() << std::endl; \
13 } while (0)
14
15 #else
16 #define TEST_OUTPUT(x)
17 #endif
18
19 class Helper
20 {
21 public:
22     Helper()
23     {
24         name = new char[1];
25         name[0] = '\0';
26     }
27
28     Helper(const char* name)
29     {
30         if (name)
31         {
32             this->name = new char[strlen(name) + 1];
33             strcpy(this->name, name);
34         }
35         else
36         {
37             this->name = new char[1];
38             *(this->name) = '\0';
39         }
40     }
41
42     ~Helper()
43     {
44         delete[] name;
45     }
46
47
48     Helper(const Helper& helperObject)
```



```
49     {
50         this->name = new char[strlen(helperObject.name) + 1];
51         strcpy(this->name, helperObject.name);
52     }
53
54     Helper& operator=(const Helper& helperObject)
55     {
56         if (this != &helperObject)
57         {
58             delete[] this->name;
59             this->name = new char[strlen(helperObject.name) + 1];
60             strcpy(this->name, helperObject.name);
61         }
62         return *this;
63     }
64
65     char* name;
66 };
67
68 void test() {
69     Helper helperObject("Helper");
70     helperObject = helperObject;
71     TEST_OUTPUT(helperObject.name);
72 }
```

Listing 51: Korrigierter Codeausschnitt TC8_B Gemini 2.5 Pro

Erläuterungen zu den Korrekturen

1. Double Free im Zuweisungsoperator:

- Es wurde eine Überprüfung auf Selbstzuweisung (`if (this != &helperObject)`) hinzugefügt. Dadurch wird sichergestellt, dass der Speicher nur freigegeben und neu zugewiesen wird, wenn es sich nicht um dasselbe Objekt handelt.

2. Null Pointer Dereference im Standardkonstruktor:

- Der Standardkonstruktor wurde geändert, um `name` mit einem leeren String zu initialisieren (`name = new char[1]; name[0] = '\0';`). Dies verhindert, dass `name` ein Null Pointer ist.

Diese Korrekturen beheben die identifizierten Speicherverwaltungsfehler und erhöhen die Sicherheit des Codes.

A.10 Testfall *TC9_B*

A.10.1 Prompt

Du bist ein Sicherheitsexperte mit Spezialisierung auf C/C++-Codeanalyse. Dein Ziel ist es, sicherheitsrelevante Speicherverwaltungsfehler zu erkennen und zu beheben. Analysiere den folgenden C/C++-Code ausschließlich im Hinblick auf sicherheitskritische Speicherfehler wie Buffer Overflows, Use-after-free oder Memory Leaks und behebe diese auch gleich falls der Code solche Fehler hat.

Erzeuge eine Tabelle in Latex mit den folgenden Spalten:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
-----	--------	------------------------	-----------	--------------	------------

Analysiere ausschließlich sicherheitsrelevante Aspekte, keine Stil- oder Architekturfragen.

```
```cpp
#define ENABLE_TEST_OUTPUT 1

#if ENABLE_TEST_OUTPUT
#include <sstream>
#include <iostream>
#include <codecvt>
#include <locale>

#define TEST_OUTPUT(x) do { \
 std::wstringstream wss; \
 wss << x; \
 std::wcout << "\t" << wss.str() << std::endl; \
} while (0)

#else
#define TEST_OUTPUT(x)
#endif

char* helper_1(const char* aString)
{
 size_t i = 0;
 size_t j;
 char* reversedString = NULL;
 if (aString != NULL)
 {
 i = strlen(aString);
 reversedString = (char*)malloc(i + 1);
 if (reversedString == NULL) { exit(-1); }
 for (j = 0; j < i; j++)
 {
 reversedString[j] = aString[i - j - 1];
 }
 }
}
```

```
 }
 reversedString[i] = '\0';
 free(reversedString);
 return reversedString;
 }
 else
 {
 return NULL;
 }
}

void test()
{
 char* reversedString;
 reversedString = helper_1("Another nice string");
 TEST_OUTPUT(reversedString);
}
...
```

### A.10.2 GPT-4o Antwort TC9\_B

Der bereitgestellte Code enthält eine sicherheitsrelevante Schwachstelle in der Funktion `helper_1()`. Die Funktion gibt einen Zeiger auf einen bereits freigegebenen Speicherbereich zurück, was zu undefiniertem Verhalten und potenziellen Sicherheitslücken führen kann (Use-after-free). Nachfolgend die Analyse im gewünschten LaTeX-Tabellenformat:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-416	Use After Free	Dangling Pointer	In der Funktion <code>helper_1()</code> wird der Speicher von <code>reversedString</code> mit <code>free()</code> freigegeben, bevor der Pointer zurückgegeben wird. Dies führt dazu, dass im weiteren Programmverlauf auf freigegebenen Speicher zugegriffen wird.	Speicher sollte nicht vor dem Rückgabewert freigegeben werden. Die <code>free()</code> -Anweisung muss aus der Funktion entfernt oder vom aufrufenden Code übernommen werden.

Tabelle 48: Analyse sicherheitsrelevanter Speicherfehler TC9\_B ChatGPT-4o

Korrigierte Version der Funktion `helper_1()`:

```

1 char* helper_1(const char* aString)
2 {
3 size_t i = 0;
4 size_t j;
5 char* reversedString = NULL;
6 if (aString != NULL)
7 {
8 i = strlen(aString);
9 reversedString = (char*)malloc(i + 1);
10 if (reversedString == NULL) { exit(-1); }
11 for (j = 0; j < i; j++)
12 {
13 reversedString[j] = aString[i - j - 1];
14 }
15 reversedString[i] = '\0';
16 return reversedString; // Speicher wird nicht freigegeben!
17 }
18 else
19 {

```

```
20 return NULL;
21 }
22 }
```

Listing 52: Korrigierte Version der Funktion `helper_1()`

Und im aufrufenden Code (`test()`), sollte der Speicher anschließend korrekt freigegeben werden:

```
1 void test()
2 {
3 char* reversedString;
4 reversedString = helper_1("Another nice string");
5 TEST_OUTPUT(reversedString);
6 free(reversedString); // Speicherfreigabe im Aufrufer
7 }
```

Listing 53: Korrigierte Version der Funktion `test()`

Wenn du möchtest, kann ich auch einen Testfall zur Laufzeiterkennung solcher Use-after-free-Probleme mit AddressSanitizer vorschlagen.

### A.10.3 Sonnet 3.7 Antwort TC9\_B

Als Sicherheitsexperte für C/C++-Codeanalyse habe ich den vorliegenden Code auf sicherheitsrelevante Speicherverwaltungsfehler untersucht. Ich habe mehrere kritische Probleme identifiziert, die ich nachfolgend detailliert erläutere.

#### Sicherheitsanalyse:

Der Code enthält einen schwerwiegenden Use-after-free-Fehler sowie potenzielle Probleme bei der Speicherverwaltung. Hier ist eine detaillierte Analyse der gefundenen Schwachstellen:

Die Funktion `helper_1()` enthält einen kritischen Use-after-free-Fehler. Der Speicher für `reversedString` wird korrekt allokiert und mit Daten befüllt, wird jedoch vor der Rückgabe mit `free(reversedString)` freigegeben. Anschließend wird der Zeiger auf den bereits freigegebenen Speicherbereich zurückgegeben. Dies führt in der aufrufenden Funktion `test()` zu einem Zugriff auf bereits freigegebenen Speicher, was ein undefiniertes Verhalten verursacht.

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-416	Use After Free	Kritisch	In <code>helper_1</code> wird der Speicher mit <code>free()</code> freigegeben, bevor der Zeiger zurückgegeben wird. Dies führt zu einem Zugriff auf bereits freigegebenen Speicher in der <code>test()</code> -Funktion.	Die <code>free()</code> -Anweisung aus der Funktion entfernen und die Speicherfreigabe dem Aufrufer überlassen.
2	CWE-401	Memory Leak	Mittel	In der <code>test()</code> -Funktion wird der zurückgegebene Speicher nicht freigegeben, was zu einem Memory Leak führen würde (falls der Use-after-free-Fehler behoben wird).	Nach der Verwendung von <code>reversedString</code> in <code>test()</code> sollte <code>free(reversedString);</code> aufgerufen werden.

Tabelle 49: Analyse sicherheitsrelevanter Speicherfehler TC9\_B Sonnet 3.7

### Korrigierter Code:

Hier ist der korrigierte Code, der die identifizierten Sicherheitsprobleme behebt:

```

1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>
7 #include <locale>
8
9 #define TEST_OUTPUT(x) do { \
10 std::wstringstream wss; \
11 wss << x; \
12 std::wcout << "\t" << wss.str() << std::endl; \
13 } while (0)
14
15 #else
16 #define TEST_OUTPUT(x)
17 #endif
18
19 char* helper_1(const char* aString)
20 {
21 size_t i = 0;
22 size_t j;
23 char* reversedString = NULL;
24 if (aString != NULL)
25 {
26 i = strlen(aString);
27 reversedString = (char*)malloc(i + 1);
28 if (reversedString == NULL) { exit(-1); }
29 for (j = 0; j < i; j++)
30 {
31 reversedString[j] = aString[i - j - 1];
32 }
33 }

```

```
33 reversedString[i] = '\\0';
34 // free(reversedString); -- Diese Zeile wurde entfernt, um
 ↪ den Use-after-free-Fehler zu beheben
35 return reversedString;
36 }
37 else
38 {
39 return NULL;
40 }
41 }
42
43 void test()
44 {
45 char* reversedString;
46 reversedString = helper_1("Another nice string");
47 TEST_OUTPUT(reversedString);
48 free(reversedString); // Speicher freigeben, um Memory Leaks zu
 ↪ vermeiden
49 }
```

Listing 54: Korrigierte Version TC9\_B Sonnet 3.7

### Zusätzliche Anmerkungen

Obwohl nicht direkt Teil der Sicherheitsanalyse, möchte ich darauf hinweisen, dass der Code auch andere potenzielle Probleme aufweist:

1. **Die Fehlerbehandlung mit `exit(-1)`** bei fehlgeschlagener Speicherallokation ist sehr drastisch und beendet das gesamte Programm. In einer Produktionsumgebung wäre eine elegantere Fehlerbehandlung angebracht.
2. **Die `strlen()`-Funktion** wird ohne Header-Einbindung verwendet. Der Code sollte `#include <string.h>` oder `#include <cstring>` enthalten.

Die durchgeführten Korrekturen beheben die kritischen Sicherheitsprobleme und stellen sicher, dass der Speicher ordnungsgemäß verwaltet wird.

### A.10.4 Gemini 3.7 Antwort TC9\_B

Als Sicherheitsexperte für C/C++-Code werde ich den Code auf sicherheitsrelevante Speicherwaltungsfehler analysieren und Empfehlungen zur Behebung geben.

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-415	Double Free	Speicherverwaltungsfehler	<code>reversedString</code> wird in <code>helper_1</code> freigegeben und der freigegebene Speicher wird in <code>test()</code> zurückgegeben, was zu einem Double Free führt, wenn der zurückgegebene Wert auch freigegeben wird.	Entferne <code>free(reversedString)</code> in <code>helper_1</code> , da der Speicher von der aufrufenden Funktion (z.B. <code>test()</code> ) verwaltet werden soll.
2	CWE-676	Release of Memory Before All References Are Cleared	Speicherverwaltungsfehler	Der in der Funktion <code>helper_1</code> allokierte Speicher wird freigegeben, bevor der Wert zurückgegeben wird. Somit dereferenziert der Aufrufer einen ungültigen Speicherbereich.	Entferne <code>free(reversedString)</code> in <code>helper_1</code> , da der Speicher von der aufrufenden Funktion (z.B. <code>test()</code> ) verwaltet werden soll.

Tabelle 50: Analyse sicherheitsrelevanter Speicherfehler TC9\_B Gemini 2.5 Pro

## Erklärung der Schwachstellen und Empfehlungen

### 1. CWE-415 Double Free

- **Beschreibung:** Die Funktion `helper_1` allokiert Speicher für `reversedString`, füllt diesen mit umgekehrtem Inhalt und gibt den Speicher dann mit `free(reversedString)` frei. Anschließend wird der freigegebene Pointer zurückgegeben. In der Funktion `test()` wird dieser freigegebene Pointer der Variablen `reversedString` zugewiesen. Wenn `reversedString` in `test()` freigegeben würde, entstünde ein Double Free.
- **Empfehlung:** Entferne den Aufruf von `free(reversedString)` in der Funktion `helper_1`. Die Funktion sollte den allokierten Speicher zurückgeben, ohne ihn freizugeben, da die aufrufende Funktion für die Verwaltung des Speichers verantwortlich sein sollte.

### 2. CWE-676 Release of Memory Before All References Are Cleared

- **Beschreibung:** In der Funktion `helper_1` wird der Speicher freigegeben, bevor der Wert zurückgegeben wird. Wenn also der Aufrufer, in diesem Fall die Funktion `test()`, den Wert dereferenziert, liest er einen ungültigen Speicherbereich.
- **Empfehlung:** Entferne den Aufruf von `free(reversedString)` in der Funktion `helper_1`, da der Speicher von der aufrufenden Funktion (z.B. `test()`) verwaltet werden soll.

## Korrigierter Code:

```

1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>

```



```
7 #include <locale>
8
9 #define TEST_OUTPUT(x) do { \
10 std::wstringstream wss; \
11 wss << x; \
12 std::wcout << "\t" << wss.str() << std::endl; \
13 } while (0)
14
15 #else
16 #define TEST_OUTPUT(x)
17 #endif
18
19 char* helper_1(const char* aString)
20 {
21 size_t i = 0;
22 size_t j;
23 char* reversedString = NULL;
24 if (aString != NULL)
25 {
26 i = strlen(aString);
27 reversedString = (char*)malloc(i + 1);
28 if (reversedString == NULL) { exit(-1); }
29 for (j = 0; j < i; j++)
30 {
31 reversedString[j] = aString[i - j - 1];
32 }
33 reversedString[i] = '\0';
34 // free(reversedString); // Entfernt, um Double Free zu
35 ↪ vermeiden
36 return reversedString;
37 }
38 else
39 {
40 return NULL;
41 }
42 }
43
44 void test()
45 {
46 char* reversedString;
47 reversedString = helper_1("Another nice string");
48 TEST_OUTPUT(reversedString);
49 free(reversedString); // Speicher nach Gebrauch freigeben
50 }
```

Listing 55: Korrigierter Code TC9\_B Gemini 2.5 Pro

## A.11 Testfall *TC10\_B*

### A.11.1 Prompt

Du bist ein Sicherheitsexperte mit Spezialisierung auf C/C++-Codeanalyse. Dein Ziel ist es, sicherheitsrelevante Speicherverwaltungsfehler zu erkennen und zu beheben. Analysiere den folgenden C/C++-Code ausschließlich im Hinblick auf sicherheitskritische Speicherfehler wie Buffer Overflows, Use-after-free oder Memory Leaks und behebe diese auch gleich falls der Code solche Fehler hat.

Erzeuge eine Tabelle in Latex mit den folgenden Spalten:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
-----	--------	------------------------	-----------	--------------	------------

Analysiere ausschließlich sicherheitsrelevante Aspekte, keine Stil- oder Architekturfragen.

```
```cpp
#define ENABLE_TEST_OUTPUT 1

#if ENABLE_TEST_OUTPUT
#include <sstream>
#include <iostream>
#include <codecvt>
#include <locale>

#define TEST_OUTPUT(x) do { \
    std::wstringstream wss; \
    wss << x; \
    std::wcout << "\t" << wss.str() << std::endl; \
} while (0)

#else
#define TEST_OUTPUT(x)
#endif

class TwoIntsClass
{
public:
    int intOne;
    int intTwo;
};

int helper()
{
    return (rand() % 2);
}

void test()
```

```
{
    TwoIntsClass* data;

    data = NULL;
    if (helper())
    {
        data = new TwoIntsClass;
        data->intOne = 1;
        data->intTwo = 2;
        delete data;
    }
    else
    {
        data = new TwoIntsClass;
        data->intOne = 1;
        data->intTwo = 2;
    }
    if (helper())
    {
        TEST_OUTPUT(data->intOne);
    }
    else
    {
    }
}
...
```

A.11.2 GPT-4o Antwort TC10_B

Hier ist die sicherheitsrelevante Analyse des gegebenen C++-Codes in Bezug auf Speicherverwaltungsfehler:

Analyse:

Im gezeigten Code wird dynamisch Speicher für ein Objekt der Klasse TwoIntsClass allokiert. Je nach Ausführungspfad wird das Objekt gelöscht – oder nicht. Anschließend kann (abhängig von einem weiteren Zufallspfad) auf das Objekt zugegriffen werden, unabhängig davon, ob es noch gültig ist. Daraus ergibt sich eine klassische Use-after-free-Problematik.

Latex-Tabelle:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-416	Use After Free	Dangling Pointer	In einem Ausführungspfad wird das Objekt <code>data</code> nach der Allokation direkt wieder freigegeben. Anschließend wird <code>data->intOne</code> möglicherweise erneut gelesen, was zu undefiniertem Verhalten führen kann, wenn auf bereits freigegebenen Speicher zugegriffen wird.	Nach dem <code>delete</code> sollte <code>data</code> auf <code>nullptr</code> gesetzt werden. Zusätzlich sollte vor der Nutzung geprüft werden, ob <code>data</code> noch gültig ist (z.B. durch ein zusätzliches Flag oder durch Umstrukturierung der Logik, um doppelte Pfade zu vermeiden).

Tabelle 51: Analyse sicherheitsrelevanter Speicherfehler TC10_B ChatGPT-4o

Korrigierter Code (Fix):

```

1 void test()
2 {
3     TwoIntsClass* data = nullptr;
4     bool wasFreed = false;
5
6     if (helper())
7     {
8         data = new TwoIntsClass;
9         data->intOne = 1;
10        data->intTwo = 2;
11        delete data;
12        data = nullptr;
13        wasFreed = true;
14    }
15    else
16    {
17        data = new TwoIntsClass;
18        data->intOne = 1;
19        data->intTwo = 2;
20    }
21

```

```

22     if (!wasFreed && data != nullptr && helper())
23     {
24         TEST_OUTPUT(data->intOne);
25     }
26
27     // Speicherfreigabe im else-Zweig
28     if (!wasFreed && data != nullptr)
29     {
30         delete data;
31         data = nullptr;
32     }
33 }

```

Listing 56: Korrigierter Codeausschnitt TC10_B ChatGPT-4o:

Wenn du zusätzlich eine Version mit Kommentaren oder erweiterten Sicherungsmaßnahmen brauchst (z.B. Smart Pointer), sag einfach Bescheid.

A.11.3 Sonnet 3.7 Antwort TC10_B

Als Sicherheitsexperte für C/C++-Codeanalyse habe ich den bereitgestellten Code untersucht und mehrere sicherheitsrelevante Speicherverwaltungsfehler identifiziert. Hier ist meine Analyse:

Der Code enthält einen potenziellen Use-after-free-Fehler und einen Memory Leak. Diese Probleme entstehen durch die unzureichende Verwaltung des dynamisch allokierten TwoIntsClass-Objekts.

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-416	Use After Free	Speicherzugriff	In der if-Verzweigung wird auf <code>data->intOne</code> zugegriffen, obwohl <code>data</code> möglicherweise bereits im vorherigen if-Zweig mit <code>delete data</code> freigegeben wurde.	Vor jedem Zugriff auf <code>data</code> prüfen, ob der Pointer gültig ist oder alternativ Smart Pointer wie <code>std::unique_ptr</code> verwenden.
2	CWE-401	Memory Leak	Ressourcenleck	Im <code>else</code> -Zweig wird Speicher für <code>data</code> allokiert, aber nie freigegeben, was zu einem Memory Leak führt.	Sicherstellen, dass alle allokierten Ressourcen auch wieder freigegeben werden oder stattdessen Smart Pointer verwenden.

Tabelle 52: Analyse sicherheitsrelevanter Speicherfehler TC10_B Sonnet 3.7

Korrigierter Code:

Hier ist der korrigierte Code, der die identifizierten Sicherheitsprobleme behebt:

```

1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>
7 #include <locale>
8 #include <memory> // Für Smart Pointer

```

```
9
10 #define TEST_OUTPUT(x) do { \
11     std::wstringstream wss; \
12     wss << x; \
13     std::wcout << "\t" << wss.str() << std::endl; \
14 } while (0)
15
16 #else
17 #define TEST_OUTPUT(x)
18 #endif
19
20 class TwoIntsClass
21 {
22 public:
23     int intOne;
24     int intTwo;
25 };
26
27 int helper()
28 {
29     return (rand() % 2);
30 }
31
32 void test()
33 {
34     // Verwendung von Smart Pointer statt Raw Pointer
35     std::unique_ptr<TwoIntsClass> data;
36
37     if (helper())
38     {
39         data = std::make_unique<TwoIntsClass>();
40         data->intOne = 1;
41         data->intTwo = 2;
42         // kein manuelles delete notwendig - wird automatisch beim
43         ↪ Verlassen des Scopes durchgeführt
44     }
45     else
46     {
47         data = std::make_unique<TwoIntsClass>();
48         data->intOne = 1;
49         data->intTwo = 2;
50     }
51
52     // Zugriff nur, wenn data gültig ist
53     if (helper() && data)
54     {
55         TEST_OUTPUT(data->intOne);
56     }
```

Listing 57: Korrigierter Codeausschnitt TC10_B Sonnet 3.7

Die Hauptänderungen umfassen:

1. Verwendung von `std::unique_ptr` statt eines Raw Pointers, um automatische Ressourcenfreigabe zu gewährleisten.
2. Hinzufügen einer Null-Prüfung vor dem Zugriff auf das Objekt.
3. Entfernen des manuellen `delete`-Aufrufs, da der Smart Pointer dies automatisch übernimmt.

Diese Änderungen beheben sowohl das Use-after-free-Problem als auch das Memory Leak und machen den Code deutlich robuster gegen Speicherverwaltungsfehler.

A.11.4 Gemini 3.7 Antwort TC10_B

Als Sicherheitsexperte mit Spezialisierung auf C/C++-Codeanalyse habe ich den bereitgestellten Code auf sicherheitsrelevante Speicherverwaltungsfehler untersucht und behebe diese im Folgenden.

Analyse

Der Code enthält einen potenziellen Use-after-free-Fehler. Wenn `helper()` in der ersten `if`-Bedingung wahr ist, wird Speicher für `data` allokiert, beschrieben und anschließend freigegeben. Wenn `helper()` in der zweiten `if`-Bedingung wahr ist, wird auf `data` zugegriffen, nachdem der Speicher bereits freigegeben wurde, was zu einem Use-after-free-Fehler führt.

Empfehlung

Um den Use-after-free-Fehler zu beheben, sollte sichergestellt werden, dass auf `data` nicht zugegriffen wird, wenn es freigegeben wurde. Dies kann durch Setzen von `data = NULL` nach dem `delete data;` Aufruf erreicht werden und durch Überprüfen von `data` auf `NULL` vor dem Zugriff darauf.

Korrigierter Code

```
1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>
7 #include <locale>
8
9 #define TEST_OUTPUT(x) do { \
10     std::wstringstream wss; \
11     wss << x; \
12     std::wcout << "\t" << wss.str() << std::endl; \
13 } while (0)
14
```

```
15 #else
16 #define TEST_OUTPUT(x)
17 #endif
18
19 class TwoIntsClass
20 {
21 public:
22     int intOne;
23     int intTwo;
24 };
25
26 int helper()
27 {
28     return (rand() % 2);
29 }
30
31 void test()
32 {
33     TwoIntsClass* data;
34
35     data = NULL;
36     if (helper())
37     {
38         data = new TwoIntsClass;
39         data->intOne = 1;
40         data->intTwo = 2;
41         delete data;
42         data = NULL;
43     }
44     else
45     {
46         data = new TwoIntsClass;
47         data->intOne = 1;
48         data->intTwo = 2;
49     }
50     if (helper() && data != NULL)
51     {
52         TEST_OUTPUT(data->intOne);
53     }
54     else
55     {
56
57     }
58     if(data != NULL)
59     {
60         delete data;
61     }
62 }
```

Listing 58: Korrigierter Codeausschnitt TC10_B Gemini 2.5 Pro

Tabelle der Schwachstellen:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-416	Use-after-free	Speicherverwaltungsfehler	Zugriff auf ein Objekt, nachdem es freigegeben wurde.	Setzen Sie den Pointer auf NULL nach dem Freigeben des Speichers und überprüfen Sie den Pointer vor dem Zugriff.

Tabelle 53: Analyse sicherheitsrelevanter Speicherfehler TC10_B Gemini 2.5 Pro

A.12 Testfall *TC11_B*

A.12.1 Prompt

Du bist ein Sicherheitsexperte mit Spezialisierung auf C/C++-Codeanalyse. Dein Ziel ist es, sicherheitsrelevante Speicherverwaltungsfehler zu erkennen und zu beheben. Analysiere den folgenden C/C++-Code ausschließlich im Hinblick auf sicherheitskritische Speicherfehler wie Buffer Overflows, Use-after-free oder Memory Leaks und behebe diese auch gleich falls der Code solche Fehler hat.

Erzeuge eine Tabelle in Latex mit den folgenden Spalten:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
-----	--------	------------------------	-----------	--------------	------------

Analysiere ausschließlich sicherheitsrelevante Aspekte, keine Stil- oder Architekturfragen.

```
```cpp
#define ENABLE_TEST_OUTPUT 1

#if ENABLE_TEST_OUTPUT
#include <sstream>
#include <iostream>
#include <codecvt>
#include <locale>

#define TEST_OUTPUT(x) do { \
 std::wstringstream wss; \
 wss << x; \
 std::wcout << "\t" << wss.str() << std::endl; \
} while (0)

#else
#define TEST_OUTPUT(x)
#endif

size_t helper()
{
 _CrtMemState state;
 _CrtMemCheckpoint(&state);
 return state.lSizes[_NORMAL_BLOCK];
}

class Helper_Object_base
{
public:
 virtual void action(wchar_t* data) = 0;
 virtual ~ Helper_Object_base() {}
};
```

```
class Helper_Object_concrete : public Helper_Object_base
{
public:
 void action(wchar_t* data) override
 {
 (void)data;
 }
};

void test() {
 TEST_OUTPUT(L"A nice string!\n");
 size_t num1 = helper();
 std::cout << "A nice number: " << num1 << " number" << std::endl;
 for (int i = 0; i < LOOP_COUNT; ++i)
 {
 wchar_t* data = NULL;
 data = (wchar_t*)realloc(data, 100 * sizeof(wchar_t));
 if (data == NULL) { exit(-1); }
 wcscpy(data, L"A String");
 TEST_OUTPUT(data);
 Helper_Object_base* baseObject = new Helper_Object_concrete();
 baseObject->action(data);
 delete baseObject;
 size_t tmp = helper();
 std::cout << "A nice number: " << i << ": " << tmp << " number" << std::endl;
 }
 size_t num2 = helper();
 std::cout << "A nice numbere: " << num1 << " number" << std::endl;
 std::cout << "Another nice number: " << num2 << " number" << std::endl;
 std::cout << "Another nice number: " << (num2 - num1) << " number" << std::endl;
}
...
```

### A.12.2 GPT-4o Antwort TC11\_B

Hier ist die sicherheitsrelevante Analyse des Codes mit Fokus auf Speicherverwaltungsfehler wie Buffer Overflow, Use-after-free und Memory Leak in LaTeX-Tabellenform:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-401	Memory Leak	Heap-Speicherleck	Innerhalb der Schleife wird mit <code>realloc</code> Speicher allokiert, dieser aber nie mit <code>free</code> freigegeben. Da dies in jeder Iteration geschieht, führt das zu einem kumulativen Speicherleck.	Füge am Ende jeder Schleifeniteration <code>free(data);</code> hinzu, um den zuvor allokierten Speicher korrekt freizugeben.

Tabelle 54: Analyse sicherheitsrelevanter Speicherfehler TC11\_B ChatGPT (GPT-4o)

#### Vorschlag zur Korrektur (Ausschnitt):

```
1 free(data);
```

Listing 59: Korrigierter Codeausschnitt TC11\_B ChatGPT 4o

### A.12.3 Sonnet 3.7 Antwort TC11\_B

Ich habe den C++-Code auf sicherheitsrelevante Speicherverwaltungsfehler analysiert und mehrere Probleme identifiziert. Hier ist meine Analyse:

#### Sicherheitsanalyse:

Der Code enthält mehrere sicherheitskritische Speicherverwaltungsfehler, die zu unvorhersehbarem Verhalten, Abstürzen oder sogar zu Sicherheitslücken führen können.

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-401	Memory Leak	Ressourcenleck	In der Schleife wird der Speicher für <code>data</code> in jedem Durchlauf neu allokiert, aber nie freigegeben, was zu einem Speicherleck führt.	Nach der Verwendung von <code>data</code> mit <code>free(data)</code> freigeben oder besser einen <code>std::wstring</code> verwenden.
2	CWE-476	NULL Pointer Dereference	Dereferenzierung	Die Variable <code>data</code> wird zuerst auf <code>NULL</code> gesetzt und dann mit <code>realloc</code> verwendet, was bei <code>realloc(NULL, size)</code> wie <code>malloc</code> funktioniert, aber unnötig ist.	Direkt <code>malloc</code> verwenden oder <code>data</code> nicht explizit auf <code>NULL</code> setzen, wenn gleich danach <code>realloc</code> folgt.
3	CWE-121	Stack-based Buffer Overflow	Pufferüberlauf	Bei <code>wscpy(data, LÄ String")</code> gibt es keine Längenkontrolle, was bei längeren Strings zu einem Pufferüberlauf führen könnte.	<code>wscncpy</code> verwenden oder besser <code>std::wstring</code> zur sicheren Stringverarbeitung.
4	CWE-20	Improper Input Validation	Fehlende Validierung	Die Konstante <code>LOOP_COUNT</code> wird verwendet, aber nicht definiert, was zu unvorhersehbarem Verhalten führen kann.	<code>LOOP_COUNT</code> vor der Verwendung definieren oder auf sichere Initialisierung achten.
5	CWE-457	Use of Uninitialized Variable	Uninitialisierte Variable	Die Funktion <code>_CrtMemCheckpoint</code> wird verwendet, aber <code>_NORMAL_BLOCK</code> ist nicht definiert.	Notwendige Header für Speicherdiagnostik einbinden oder alternative Methoden zur Speicherüberwachung verwenden.

Tabelle 55: Analyse sicherheitsrelevanter Speicherfehler TC11\_B Sonnet 3.7

## Korrigierter Code:

Hier ist der korrigierte Code mit behobenen Sicherheitsproblemen:

```

1 #define ENABLE_TEST_OUTPUT 1
2 #define LOOP_COUNT 10 // Definition der fehlenden Konstante
3
4 #if ENABLE_TEST_OUTPUT
5 #include <sstream>
6 #include <iostream>
7 #include <codecvt>
8 #include <locale>
9
10 #define TEST_OUTPUT(x) do { \
11 std::wstringstream wss; \
12 wss << x; \
13 std::wcout << "\t" << wss.str() << std::endl; \
14 } while (0)
15
16 #else
17 #define TEST_OUTPUT(x)
18 #endif
19
20 // Für _CrtMemCheckpoint und _NORMAL_BLOCK
21 #ifdef _MSC_VER
22 #include <crtDBG.h>

```

```
23 #else
24 // Alternative für nicht-Windows-Plattformen
25 typedef struct {
26 size_t lSizes[5];
27 } _CrtMemState;
28
29 void _CrtMemCheckpoint(_CrtMemState* state) {
30 // Dummy-Implementierung
31 for (int i = 0; i < 5; i++) {
32 state->lSizes[i] = 0;
33 }
34 }
35 #define _NORMAL_BLOCK 1
36 #endif
37
38 size_t helper()
39 {
40 _CrtMemState state;
41 _CrtMemCheckpoint(&state);
42 return state.lSizes[_NORMAL_BLOCK];
43 }
44
45 class Helper_Object_base
46 {
47 public:
48 virtual void action(wchar_t* data) = 0;
49 virtual ~Helper_Object_base() {}
50 };
51
52 class Helper_Object_concrete : public Helper_Object_base
53 {
54 public:
55 void action(wchar_t* data) override
56 {
57 (void)data;
58 }
59 };
60
61 void test() {
62 TEST_OUTPUT(L"A nice string!\n");
63 size_t num1 = helper();
64 std::cout << "A nice number: " << num1 << " number" << std::
 ↵ endl;
65 for (int i = 0; i < LOOP_COUNT; ++i)
66 {
67 // Direkt malloc verwenden statt NULL + realloc
68 wchar_t* data = (wchar_t*)malloc(100 * sizeof(wchar_t));
69 if (data == NULL) { exit(-1); }
70
71 // Sichere Variante mit wcsncpy statt wcsncpy
72 wcsncpy(data, L"A String", 99);
```

```
73 data[99] = L'\0'; // Sicherstellen, dass der String null-
 ↪ terminiert ist
74
75 TEST_OUTPUT(data);
76 Helper_Object_base* baseObject = new Helper_Object_concrete
 ↪ ();
77 baseObject->action(data);
78 delete baseObject;
79
80 // Speicher für data freigeben, um Memory Leak zu vermeiden
81 free(data);
82
83 size_t tmp = helper();
84 std::cout << "A nice number: " << i << ": " << tmp << "
 ↪ number" << std::endl;
85 }
86 size_t num2 = helper();
87 std::cout << "A nice numbere: " << num1 << " number" << std::
 ↪ endl;
88 std::cout << "Another nice number: " << num2 << " number" <<
 ↪ std::endl;
89 std::cout << "Another nice number: " << (num2 - num1) << "
 ↪ number" << std::endl;
90 }
```

Listing 60: Korrigierter Codeausschnitt TC11\_B Sonnet 3.7

## Zusätzliche Empfehlungen

Für eine noch sicherere Implementierung würde ich folgende Änderungen empfehlen:

1. **Verwendung von Smart Pointern:** Statt manueller Speicherverwaltung mit `new/delete` sollten Smart Pointer wie `std::unique_ptr` verwendet werden.
2. **Verwendung von STL-Containern:** Statt C-Style-Arrays mit manueller Speicherverwaltung sollten STL-Container wie `std::wstring` verwendet werden.
3. **Fehlerbehandlung verbessern:** Statt bei Fehlern mit `exit(-1)` zu beenden, sollten Ausnahmen geworfen oder Fehlercodes zurückgegeben werden.
4. **Plattformunabhängige Speicherdiagnostik:** Die Verwendung von Windows-spezifischen Funktionen wie `_CrtMemCheckpoint` sollte durch plattformunabhängige Alternativen ersetzt werden.

Diese Änderungen würden den Code nicht nur sicherer, sondern auch wartbarer und robuster machen.

### A.12.4 Gemini 3.7 Antwort TC11\_B

Hier ist eine Analyse des C/C++-Codes im Hinblick auf sicherheitsrelevante Speicherwaltungsfehler, zusammen mit Korrekturen und Empfehlungen, dargestellt in einer LaTeX-Tabelle.

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-415	Double Free	Speicherfehler	Der Code gibt den durch <b>data</b> belegten Speicher nicht frei, was zu einem Speicherleck führt.	Verwende Smart Pointers oder rufe <b>free(data)</b> auf, nachdem <b>data</b> nicht mehr benötigt wird.
2	CWE-415	Memory Leak	Speicherfehler	Der Code gibt den durch <b>data</b> belegten Speicher nicht frei, was zu einem Speicherleck führt.	Stelle sicher, dass der mit <b>realloc</b> allozierte Speicher auch wieder freigegeben wird, um Speicherlecks zu vermeiden. Am Ende der Schleife <b>free(data)</b> aufrufen.

Tabelle 56: Analyse sicherheitsrelevanter Speicherfehler TC11\_B Gemini 2.5 Pro

## Erläuterungen und Verbesserungen

### 1. Speicherleck

- **CWE-ID:** CWE-415
- **Name der Schwachstelle:** Speicherleck
- **Fehlerart:** Speicherfehler
- **Beschreibung:** In der Schleife wird mit **realloc** Speicher für **data** reserviert, aber dieser Speicher wird nie freigegeben. Dies führt mit jeder Iteration zu einem Speicherleck.
- **Empfehlung:** Stelle sicher, dass der mit **realloc** allozierte Speicher auch wieder freigegeben wird, um Speicherlecks zu vermeiden. Am Ende der Schleife **free(data)** aufrufen.

### 2. Fehlende Speicherfreigabe

- **CWE-ID:** CWE-415
- **Name der Schwachstelle:** Double Free
- **Fehlerart:** Speicherfehler
- **Beschreibung:** Der Code gibt den durch **data** belegten Speicher nicht frei, was zu einem Speicherleck führt.
- **Empfehlung:** Verwende Smart Pointer oder rufe **free(data)** auf, nachdem **data** nicht mehr benötigt wird.

## Angepasster Code mit Fehlerbehebungen

Hier ist der verbesserte Code mit Behebung der identifizierten Speicherlecks:

```

1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>

```



```
7 #include <locale>
8
9 #define TEST_OUTPUT(x) do { \
10 std::wstringstream wss; \
11 wss << x; \
12 std::wcout << "\t" << wss.str() << std::endl; \
13 } while (0)
14
15 #else
16 #define TEST_OUTPUT(x)
17 #endif
18
19 size_t helper()
20 {
21 _CrtMemState state;
22 _CrtMemCheckpoint(&state);
23 return state.lSizes[_NORMAL_BLOCK];
24 }
25
26 class Helper_Object_base
27 {
28 public:
29 virtual void action(wchar_t* data) = 0;
30 virtual ~ Helper_Object_base() {}
31 };
32
33 class Helper_Object_concrete : public Helper_Object_base
34 {
35 public:
36 void action(wchar_t* data) override
37 {
38 (void)data;
39 }
40 };
41
42 void test() {
43 TEST_OUTPUT(L"A nice string!\n");
44 size_t num1 = helper();
45 std::cout << "A nice number: " << num1 << " number" << std::
 ↵ endl;
46 for (int i = 0; i < LOOP_COUNT; ++i)
47 {
48 wchar_t* data = NULL;
49 data = (wchar_t*)realloc(data, 100 * sizeof(wchar_t));
50 if (data == NULL) { exit(-1); }
51 wcsncpy(data, L"A String");
52 TEST_OUTPUT(data);
53 Helper_Object_base* baseObject = new Helper_Object_concrete
 ↵ ();
54 baseObject->action(data);
55 delete baseObject;
```

```
56 size_t tmp = helper();
57 std::cout << "A nice number: " << i << ": " << tmp << "
 ↪ number" << std::endl;
58 free(data); // Speicher freigeben
59 }
60 size_t num2 = helper();
61 std::cout << "A nice numbere: " << num1 << " number" << std::
 ↪ endl;
62 std::cout << "Another nice number: " << num2 << " number" <<
 ↪ std::endl;
63 std::cout << "Another nice number: " << (num2 - num1) << "
 ↪ number" << std::endl;
64 }
```

Listing 61: Korrigierter Codeausschnitt TC11\_B Gemini 2.5 Pro

### Zusätzliche Empfehlungen

- **Verwendung von Smart Pointern:** Um Speicherlecks und doppelte Freigaben zu vermeiden, sollten Sie Smart Pointer wie `std::unique_ptr` oder `std::shared_ptr` verwenden.
- **Compiler-Diagnose aktivieren:** Aktivieren Sie Compiler-Flags, um gefährliche Muster zu erkennen. C++-Compiler bieten verschiedene Flags und Optionen, die die Sicherheit Ihrer Anwendung verbessern können.
- **Speicherfehler-Erkennungstools:** Verwenden Sie Tools wie Parasoft Insure++ zur Laufzeit, um Speicherfehler zu erkennen.
- **Regelmäßige Speicherzustandsprüfung:** Verwenden Sie `_CrtMemCheckpoint`, um den Speicherzustand zu bestimmten Zeitpunkten zu überprüfen und Unterschiede zu analysieren.
- **Sichere Alternativen nutzen:** Verwenden Sie `std::vector` anstelle von rohen Arrays für dynamische Arrays und `std::string` für sichere String-Manipulationen.

Durch die Behebung dieser Speicherverwaltungsfehler und die Anwendung der zusätzlichen Empfehlungen kann die Sicherheit und Stabilität des Codes erheblich verbessert werden.

## A.13 Testfall *TC12\_B*

### A.13.1 Prompt

Du bist ein Sicherheitsexperte mit Spezialisierung auf C/C++-Codeanalyse. Dein Ziel ist es, sicherheitsrelevante Speicherverwaltungsfehler zu erkennen und zu beheben. Analysiere den folgenden C/C++-Code ausschließlich im Hinblick auf sicherheitskritische Speicherfehler wie Buffer Overflows, Use-after-free oder Memory Leaks und behebe diese auch gleich falls der Code solche Fehler hat.

Erzeuge eine Tabelle in Latex mit den folgenden Spalten:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
-----	--------	------------------------	-----------	--------------	------------

Analysiere ausschließlich sicherheitsrelevante Aspekte, keine Stil- oder Architekturfragen.

```
```cpp
#define ENABLE_TEST_OUTPUT 1

#if ENABLE_TEST_OUTPUT
#include <sstream>
#include <iostream>
#include <codecvt>
#include <locale>

#define TEST_OUTPUT(x) do { \
    std::wstringstream wss; \
    wss << x; \
    std::wcout << "\t" << wss.str() << std::endl; \
} while (0)

#else
#define TEST_OUTPUT(x)
#endif

#define LOOP_COUNT 100

void helper(std::map<int, char*> dataMap)
{
    char* data = dataMap[2];
    ;
}

void test() {
    TEST_OUTPUT(L"A nice string!\n");
    size_t num1 = helper();
    std::cout << "A nice number: " << num1 << " number" << std::endl;
    for (int i = 0; i < LOOP_COUNT; ++i)
```

```
{
char* data;
std::map<int, char*> dataMap;
data = NULL;
data = (char*)calloc(100, sizeof(char));
strcpy(data, "A String");
TEST_OUTPUT(data);
dataMap[0] = data;
dataMap[1] = data;
dataMap[2] = data;
helper(dataMap);
}

size_t num2 = helper();
std::cout << "A nice numbere: " << num1 << " number" << std::endl;
std::cout << "Another nice number: " << num2 << " number" << std::endl;
std::cout << "Another nice number: " << (num2 - num1) << " number" << std::endl;
}
...
```

A.13.2 GPT-4o Antwort TC12_B

Hier ist die sicherheitsrelevante Analyse deines C++-Codes in Bezug auf Speicherverwaltungsfehler mit Fokus auf Buffer Overflows, Use-after-free und Memory Leaks. Die Analyse umfasst identifizierte Schwachstellen inklusive Beschreibung und Empfehlung:

Analyse (LaTeX-Tabelle):

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-401	Memory Leak	Ressourcenleck	In der Schleife wird in jedem Durchlauf Speicher mit <code>calloc</code> allokiert, aber nie freigegeben. Dies führt zu einem kontinuierlichen Speicherverlust.	Füge am Ende jedes Schleifendurchlaufs <code>free(data);</code> hinzu, um den allokierten Speicher wieder freizugeben.
2	CWE-252	Unchecked Return Value	Fehlerhafte Speicheroperation	Die Rückgabe von <code>calloc</code> wird nicht auf <code>NULL</code> überprüft. Im Falle eines Fehlers kann <code>strcpy</code> auf einen ungültigen Zeiger zugreifen.	Überprüfe, ob der Rückgabewert von <code>calloc</code> ungleich <code>NULL</code> ist, bevor <code>strcpy</code> aufgerufen wird.
3	CWE-120	Buffer Overflow	Schreibzugriff außerhalb der Grenzen	Falls <code>strcpy</code> auf eine zu kurze Zielpufferlänge trifft (z. B. bei Veränderung der Quellzeichenkette), kann ein Buffer Overflow entstehen.	Ersetze <code>strcpy</code> durch <code>strncpy</code> und stelle sicher, dass die Länge nicht überschritten wird.
4	CWE-685	Falsche Argumentenzahl beim Funktionsaufruf	Undefiniertes Verhalten	Der Funktionsaufruf <code>size_t num1 = helper();</code> und <code>helper(dataMap);</code> ist inkonsistent. <code>helper()</code> erwartet ein Argument vom Typ <code>std::map<int, char*></code> , wird aber mehrmals ohne Argument aufgerufen.	Entferne die falschen Aufrufe von <code>helper()</code> ohne Argument oder implementiere eine passende Überladung.

Tabelle 57: Analyse sicherheitsrelevanter Speicherfehler TC12_B ChatGPT-4o

Vorschlag zur Behebung der Schwachstellen:

```

1 void helper(std::map<int, char*> dataMap)
2 {
3     char* data = dataMap[2];
4     // Beispielhafter Zugriff, reale Nutzung ggf. ergänzen
5     TEST_OUTPUT(data);
6 }
7
8 void test() {
9     TEST_OUTPUT(L"A nice string!\n");
10
11     for (int i = 0; i < LOOP_COUNT; ++i)
12     {
13         char* data = (char*)calloc(100, sizeof(char));
14         if (data == NULL) {
15             std::cerr << "Memory allocation failed\n";
16             continue;

```

```

17     }
18
19     strncpy(data, "A String", 99);
20     data[99] = '\0'; // Null-Terminator absichern
21
22     TEST_OUTPUT(data);
23
24     std::map<int, char*> dataMap;
25     dataMap[0] = data;
26     dataMap[1] = data;
27     dataMap[2] = data;
28
29     helper(dataMap);
30
31     free(data);
32 }
33
34 // Falsche Funktionsaufrufe entfernt
35 }

```

Listing 62: Korrigierter Codeausschnitt TC12_B ChatGPT

Wenn du möchtest, kann ich dir auch ein Diff oder ein vollständiges Vorher/Nachher anzeigen. Sag einfach Bescheid.

A.13.3 Sonnet 3.7 Antwort TC12_B

Als Sicherheitsexperte für C/C++-Codeanalyse habe ich den bereitgestellten Code auf sicherheitsrelevante Speicherverwaltungsfehler untersucht. Ich habe mehrere kritische Probleme identifiziert, die ich im Folgenden detailliert analysiere und behebe.

Sicherheitsanalyse:

Der Code enthält mehrere schwerwiegende Speicherverwaltungsfehler, die zu Sicherheitslücken führen können:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-401	Memory Leak	Ressourcenleck	In der <code>for</code> -Schleife wird Speicher mit <code>calloc()</code> allokiert, aber nie freigegeben, was zu einem Speicherleck pro Iteration führt.	Speicher mit <code>free()</code> freigeben, bevor der Gültigkeitsbereich verlassen wird, oder Smart Pointer verwenden.
2	CWE-685	Funktionsprototyp-Fehler	Funktionsdefinition	Die Funktion <code>helper()</code> wird mit Rückgabotyp <code>size_t</code> verwendet, aber als <code>void</code> definiert.	Funktionsdeklaration und -definition konsistent gestalten.
3	CWE-476	NULL Pointer Dereference	Dereferenzierung	In <code>helper()</code> wird <code>dataMap</code> ohne Prüfung auf NULL verwendet.	Null-Prüfung vor Verwendung des Pointers einführen.
4	CWE-119	Unsichere String-funktion	Buffer Overflow	Die Verwendung von <code>strcpy()</code> ohne Längenbegrenzung kann zu Pufferüberläufen führen.	<code>strncpy()</code> oder besser noch <code>std::string</code> verwenden.

Tabelle 58: Analyse sicherheitsrelevanter Speicherfehler TC12_B Sonnet 3.7

Korrigierte Code:

Hier ist der korrigierte Code mit Behebung aller identifizierten Sicherheitsprobleme:

```
1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>
7 #include <locale>
8
9 #define TEST_OUTPUT(x) do { \
10     std::wstringstream wss; \
11     wss << x; \
12     std::wcout << "\t" << wss.str() << std::endl; \
13 } while (0)
14
15 #else
16 #define TEST_OUTPUT(x)
17 #endif
18
19 #include <map>
20 #include <cstring>
21 #include <cstdlib>
22 #include <string>
23
24 #define LOOP_COUNT 100
25
26 // Korrigierte Funktionssignatur mit Rückgabotyp
27 size_t helper(const std::map<int, char*>& dataMap)
28 {
29     // Null-Prüfung hinzugefügt
30     if (dataMap.find(2) != dataMap.end() && dataMap.at(2) !=
31         ↪ nullptr) {
32         char* data = dataMap.at(2);
33         // Beispielrückgabewert basierend auf Stringlänge
34         return strlen(data);
35     }
36     return 0;
37 }
38
39 void test() {
40     TEST_OUTPUT(L"A nice string!\n");
41
42     // Initialisierung mit 0, da helper() beim ersten Aufruf keine
43     ↪ gültigen Daten hat
44     size_t num1 = 0;
45     std::cout << "A nice number: " << num1 << " number" << std::
46         ↪ endl;
47
48     std::map<int, char*> dataMap;
```

```
46     char* data = nullptr;
47
48     for (int i = 0; i < LOOP_COUNT; ++i)
49     {
50         // Speicher nur einmal allokieren und wiederverwenden
51         if (data == nullptr) {
52             data = static_cast<char*>(calloc(100, sizeof(char)));
53             if (data == nullptr) {
54                 std::cerr << "Speicherallokation fehlgeschlagen" <<
55                     ↪ std::endl;
56                 return;
57             }
58
59             // Sicheres String-Kopieren mit Längenbegrenzung
60             strncpy(data, "A String", 99);
61             data[99] = '\0'; // Sicherstellen, dass der String null-
62                 ↪ terminiert ist
63
64             TEST_OUTPUT(data);
65
66             dataMap[0] = data;
67             dataMap[1] = data;
68             dataMap[2] = data;
69
70             num1 = helper(dataMap);
71         }
72
73         size_t num2 = helper(dataMap);
74         std::cout << "A nice number: " << num1 << " number" << std::
75             ↪ endl;
76         std::cout << "Another nice number: " << num2 << " number" <<
77             ↪ std::endl;
78         std::cout << "Another nice number: " << (num2 - num1) << "
79             ↪ number" << std::endl;
80
81         // Speicher freigeben
82         free(data);
83     }
```

Listing 63: Korrigierter Codeausschnitt TC12_B Sonnet

Erläuterung der Korrekturen

1. **Memory Leak behoben:** Der allokierte Speicher wird nun am Ende der Funktion freigegeben. Außerdem wird der Speicher nur einmal allokiert und wiederverwendet, anstatt in jeder Schleifeniteration neu zu allokieren.
2. **Funktionsprototyp korrigiert:** Die `helper()`-Funktion hat nun einen konsistenten Rückgabetyt (`size_t`) und gibt die Länge des Strings zurück.

3. **NULL Pointer Dereference verhindert:** Es wurde eine Prüfung hinzugefügt, ob der Schlüssel in der Map existiert und ob der Wert nicht NULL ist.
4. **Buffer Overflow verhindert:** `strcpy()` wurde durch `strncpy()` mit Längenbegrenzung ersetzt und eine explizite Null-Terminierung hinzugefügt.
5. **Weitere Verbesserungen:**
 - Die Funktion `helper()` erhält die Map als Referenz, um unnötiges Kopieren zu vermeiden.
 - Fehlerbehandlung bei Speicherallokation hinzugefügt.
 - Notwendige Header-Dateien hinzugefügt.

Diese Änderungen beheben alle identifizierten Sicherheitsprobleme und verbessern die Robustheit des Codes erheblich.

A.13.4 Gemini 3.7 Antwort TC12_B

Als Sicherheitsexperte für C/C++-Code werde ich den bereitgestellten Code auf Speicherwaltungsfehler analysieren.

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-401	Memory Leak	Speicherleck	Der durch <code>calloc</code> allokierte Speicher innerhalb der Schleife wird nie freigegeben, was zu einem Speicherleck führt.	Verwenden Sie <code>free(data)</code> , um den Speicher am Ende jeder Schleifeniteration freizugeben.
2	CWE-457	Uninitialized Variable	Logikfehler	Die Funktion <code>helper()</code> wird in Zeile 27 und 33 aufgerufen, obwohl diese eine <code>std::map</code> als Parameter erwartet. Dies führt zu einem Kompilierungsfehler.	Passen Sie den Funktionsaufruf von <code>helper()</code> an, sodass ein gültiges <code>std::map</code> -Objekt übergeben wird.

Tabelle 59: Analyse sicherheitsrelevanter Speicherfehler TC12_B Gemini 2.5 Pro

Hier ist der korrigierte Code:

```

1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>
7 #include <locale>
8 #include <map>
9
10 #define TEST_OUTPUT(x) do { \
11     std::wstringstream wss; \
12     wss << x; \
13     std::wcout << "\t" << wss.str() << std::endl; \
14 } while (0)
15

```

```
16 #else
17 #define TEST_OUTPUT(x)
18 #endif
19
20 #define LOOP_COUNT 100
21
22 void helper(std::map<int, char*> dataMap)
23 {
24     char* data = dataMap[2];
25     ;
26 }
27
28 void test() {
29     TEST_OUTPUT(L"A nice string!\n");
30     std::map<int, char*> dummyMap;
31     //size_t num1 = helper(); //Falscher Funktionsaufruf
32     for (int i = 0; i < LOOP_COUNT; ++i)
33     {
34         char* data;
35         std::map<int, char*> dataMap;
36         data = NULL;
37         data = (char*)calloc(100, sizeof(char));
38         strcpy(data, "A String");
39         TEST_OUTPUT(data);
40         dataMap[0] = data;
41         dataMap[1] = data;
42         dataMap[2] = data;
43         helper(dataMap);
44         free(data); // Speicher freigeben
45     }
46     //size_t num2 = helper(); //Falscher Funktionsaufruf
47     //std::cout << "A nice numbere: " << num1 << " number" << std::
48         ↪ endl;
49     //std::cout << "Another nice number: " << num2 << " number" <<
50         ↪ std::endl;
51     //std::cout << "Another nice number: " << (num2 - num1) << "
52         ↪ number" << std::endl;
53 }
```

Listing 64: Korrigierter Codeausschnitt TC12_B Gemini

A.14 Testfall *TC13_B*

A.14.1 Prompt

Du bist ein Sicherheitsexperte mit Spezialisierung auf C/C++-Codeanalyse. Dein Ziel ist es, sicherheitsrelevante Speicherverwaltungsfehler zu erkennen und zu beheben. Analysiere den folgenden C/C++-Code ausschließlich im Hinblick auf sicherheitskritische Speicherfehler wie Buffer Overflows, Use-after-free oder Memory Leaks und behebe diese auch gleich falls der Code solche Fehler hat.

Erzeuge eine Tabelle in Latex mit den folgenden Spalten:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
-----	--------	------------------------	-----------	--------------	------------

Analysiere ausschließlich sicherheitsrelevante Aspekte, keine Stil- oder Architekturfragen.

```
```cpp
#define ENABLE_TEST_OUTPUT 1

#if ENABLE_TEST_OUTPUT
#include <sstream>
#include <iostream>
#include <codecvt>
#include <locale>

#define TEST_OUTPUT(x) do { \
 std::wstringstream wss; \
 wss << x; \
 std::wcout << "\t" << wss.str() << std::endl; \
} while (0)

#else
#define TEST_OUTPUT(x)
#endif

static int flag = 1;

static void helper(char* data)
{
 if (flag)
 {
 ;
 }
}

void test()
{
 char* data;
```

```
data = NULL;
data = new char[100];
strcpy(data, "A String");
TEST_OUTPUT(data);
flag = 1; /* true */
helper(data);
}
...
```

### A.14.2 GPT-4o Antwort TC13\_B

Hier ist die sicherheitsrelevante Analyse des gegebenen C++-Codes. Es wurde ein klassisches Speicherleck identifiziert, da der mit `new` allozierte Speicher nie wieder freigegeben wird.

#### Sicherheitsanalyse (LaTeX-Tabelle)

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-401	Memory Leak	Speicherleck	Der mit <code>new[]</code> allozierte Speicher wird nicht freigegeben. Die Variable <code>data</code> zeigt auf einen Heap-Puffer, der nach der Verwendung nicht mit <code>delete[]</code> freigegeben wird, was zu einem Speicherleck führt.	Am Ende der Funktion <code>test()</code> sollte <code>delete[] data;</code> eingefügt werden, um den Speicher ordnungsgemäß freizugeben.

Tabelle 60: Analyse sicherheitsrelevanter Speicherfehler TC13\_B ChatGPT-4o

#### Fixierter Code:

```

1 void test()
2 {
3 char* data;
4 data = NULL;
5 data = new char[100];
6 strcpy(data, "A String");
7 TEST_OUTPUT(data);
8 flag = 1; /* true */
9 helper(data);
10 delete[] data; // Speicher korrekt freigeben
11 }

```

Listing 65: Korrigierter Codeausschnitt TC13\_B ChatGpt-4o

Wenn du eine zusätzliche Prüfung auf `NULL` vor dem `delete[]` möchtest (obwohl `delete[] NULL;` erlaubt ist), kann das ergänzt werden – aber in diesem Fall ist es nicht notwendig.

### A.14.3 Sonnet 3.7 Antwort TC13\_B

Als Sicherheitsexperte mit Spezialisierung auf C/C++-Codeanalyse habe ich den vorliegenden Code auf sicherheitskritische Speicherfehler untersucht. Hier ist meine Analyse: Der Code weist mehrere sicherheitsrelevante Speicherverwaltungsfehler auf, die ich im Folgenden detailliert erläutern werde.

#### Sicherheitskritische Speicherfehler

Der analysierte Code enthält folgende Schwachstellen:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-401	Memory Leak	Ressourcenleck	Der in der Funktion <code>test()</code> mit <code>new char</code> allokierter Speicher wird nicht freigegeben, was zu einem Speicherleck führt.	Speicher mit <code>delete[]</code> freigeben, bevor die Funktion beendet wird, oder Smart Pointer verwenden.
2	CWE-120	Buffer Overflow	Pufferüberlauf	Die Verwendung von <code>strcpy()</code> ohne Größenprüfung kann zu einem Pufferüberlauf führen, wenn der Quellstring länger als 100 Zeichen ist.	<code>strncpy()</code> oder besser noch <code>std::string</code> oder <code>std::array</code> verwenden.
3	CWE-476	NULL Pointer Dereference	Nullzeiger-Dereferenzierung	Obwohl <code>data</code> zunächst auf <code>NULL</code> gesetzt wird, erfolgt später eine Neuweisung. Dennoch fehlt eine Prüfung, ob die Speicherallokation erfolgreich war.	Nach der Speicherallokation prüfen, ob <code>data != NULL</code> ist, bevor darauf zugegriffen wird.

Tabelle 61: Analyse sicherheitsrelevanter Speicherfehler TC13\_B Sonnet 3.7

## Korrigierter Code

Hier ist der korrigierte Code mit Behebung der identifizierten Sicherheitsprobleme:

```

1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>
7 #include <locale>
8 #include <cstring>
9
10 #define TEST_OUTPUT(x) do { \
11 std::wstringstream wss; \
12 wss << x; \
13 std::wcout << "\t" << wss.str() << std::endl; \
14 } while (0)
15
16 #else
17 #define TEST_OUTPUT(x)
18 #endif
19
20 static int flag = 1;
21
22 static void helper(char* data)
23 {
24 if (flag)
25 {
26 ;
27 }
28 }
29
30 void test()

```

```
31 {
32 char* data = nullptr;
33
34 try {
35 data = new char[100];
36
37 // Sicherere Alternative zu strcpy
38 strncpy(data, "A String", 99);
39 data[99] = '\0'; // Sicherstellen, dass der String null-
40 ↳ terminiert ist
41
42 TEST_OUTPUT(data);
43 flag = 1; /* true */
44 helper(data);
45 }
46 catch (std::bad_alloc& e) {
47 // Fehlerbehandlung bei fehlgeschlagener Speicherallokation
48 std::cerr << "Speicherallokation fehlgeschlagen: " << e.
49 ↳ what() << std::endl;
50 return;
51 }
52
53 // Speicher freigeben, um Memory Leak zu vermeiden
54 delete[] data;
55 data = nullptr;
56 }
```

Listing 66: Korrigierter Codeausschnitt TC13\_B Sonnet 3.7

## Erläuterung der Korrekturen

### Memory Leak (CWE-401):

- Der allokierte Speicher wird nun mit `delete[] data` freigegeben, bevor die Funktion beendet wird.
- Nach dem Löschen wird der Zeiger auf `nullptr` gesetzt, um Use-after-free-Fehler zu vermeiden.

### Buffer Overflow (CWE-120):

- `strcpy()` wurde durch `strncpy()` ersetzt, wobei die maximale Kopierlänge auf 99 Zeichen begrenzt ist.
- Das letzte Zeichen wird explizit auf `'\0'` gesetzt, um sicherzustellen, dass der String nullterminiert ist.

### NULL Pointer Dereference (CWE-476):

- Ein Try-Catch-Block wurde hinzugefügt, um Fehler bei der Speicherallokation abzufangen.
- Bei einer fehlgeschlagenen Allokation wird eine entsprechende Fehlermeldung ausgegeben.

Alternativ könnte man auch moderne C++-Konstrukte wie `std::string` oder `std::vector` verwenden, um diese Speicherverwaltungsprobleme vollständig zu vermeiden.

#### A.14.4 Gemini 3.7 Antwort TC13\_B

Als Sicherheitsexperte für C/C++-Code werde ich den bereitgestellten Code auf sicherheitsrelevante Speicherverwaltungsfehler analysieren und eine Tabelle mit den gefundenen Schwachstellen erstellen.

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-120	Buffer Overflow	Speicherverwaltungsfehler	Die Funktion <code>strcpy</code> kopiert den String <code>Ä String</code> in den Puffer <code>data</code> , der mit einer Größe von 100 Bytes allokiert wurde. Wenn der kopierte String größer als 99 Bytes ist (plus Nullterminator), kommt es zu einem Buffer Overflow.	Verwendung von <code>strncpy</code> oder <code>strcpy_s</code> zur Begrenzung der Anzahl kopierter Bytes.
2	CWE-415	Double Free	Speicherverwaltungsfehler	Der allokierte Speicher von <code>data</code> wird niemals freigegeben, was zu einem Memory Leak führt, wenn die Funktion <code>test</code> mehrmals aufgerufen wird.	Verwendung von <code>delete[] data;</code> um den Speicher freizugeben.

Tabelle 62: Analyse sicherheitsrelevanter Speicherfehler TC13\_B Gemini 2.5 Pro

### Erklärung der Schwachstellen und Empfehlungen

#### 1. CWE-120 Buffer Overflow

- **Beschreibung:** Die Funktion `strcpy` ist anfällig für Buffer Overflows, da sie die Größe des Zielpuffers nicht überprüft. Im gegebenen Code wird ein Puffer `data` mit einer Größe von 100 Bytes allokiert, und dann wird `strcpy` verwendet, um den String `Ä String` in diesen Puffer zu kopieren. Wenn der kopierte String (einschließlich des Nullterminators) größer als 100 Bytes wäre, würde `strcpy` über das Ende des Puffers hinausschreiben, was zu einem Buffer Overflow führt.
- **Empfehlung:** Verwenden Sie sicherere Alternativen wie `strncpy` oder `strcpy_s`, die eine maximale Anzahl von zu kopierenden Bytes angeben. Dadurch wird verhindert, dass über das Ende des Puffers hinausgeschrieben wird.

#### 2. CWE-415 Doppeltes Freigeben

- **Beschreibung:** Der durch `new char` allokierte Speicher für `data` wird nicht durch `delete[]` freigegeben. Dies führt zu einem Speicherleck.
- **Empfehlung:** Stellen Sie sicher, dass jeder mit `new` allokierte Speicher mit `delete` freigegeben wird, um Speicherlecks zu vermeiden. Im Falle von Arrays, die mit `new[]` allokiert wurden, muss `delete[]` verwendet werden.



## Korrigierter Code

Um die oben genannten Schwachstellen zu beheben, kann der Code wie folgt geändert werden:

```
1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>
7 #include <locale>
8
9 #define TEST_OUTPUT(x) do { \
10 std::wstringstream wss; \
11 wss << x; \
12 std::wcout << "\t" << wss.str() << std::endl; \
13 } while (0)
14
15 #else
16 #define TEST_OUTPUT(x)
17 #endif
18
19 #include <cstring> // für strncpy
20
21 static int flag = 1;
22
23 static void helper(char* data)
24 {
25 if (flag)
26 {
27 ;
28 }
29 }
30
31 void test()
32 {
33 char* data;
34 data = NULL;
35 data = new char[100];
36 strncpy(data, "A String", 99); // Buffer Overflow verhindern
37 data[99] = '\0'; // Nullterminierung sicherstellen
38 TEST_OUTPUT(data);
39 flag = 1; /* true */
40 helper(data);
41 delete[] data; // Speicher freigeben
42 data = NULL;
43 }
```

Listing 67: Korrigierter Codeausschnitt TC13\_B Gemini 3.7

## Änderungen im Detail

- **strncpy** anstelle von **strcpy**: Die Funktion `strncpy(data, "Ä String", 99)` kopiert maximal 99 Zeichen von "Ä String" in den Puffer `data`. Dies verhindert einen Buffer Overflow, da sichergestellt ist, dass nicht mehr als 99 Zeichen in den 100 Byte großen Puffer geschrieben werden.
- **Nullterminierung**: `data[99] = '\0'`; stellt sicher, dass der Puffer immer mit einem Nullzeichen abgeschlossen wird, auch wenn der kopierte String kürzer als 99 Zeichen ist.
- **Speicher freigeben**: `delete[] data`; gibt den zuvor allokierten Speicher frei, um Speicherlecks zu verhindern.
- `data = NULL`; Setzt den Pointer auf NULL, nachdem der Speicher freigegeben wurde.

## A.15 Testfall *TC14\_B*

### A.15.1 Prompt

Du bist ein Sicherheitsexperte mit Spezialisierung auf C/C++-Codeanalyse. Dein Ziel ist es, sicherheitsrelevante Speicherverwaltungsfehler zu erkennen und zu beheben. Analysiere den folgenden C/C++-Code ausschließlich im Hinblick auf sicherheitskritische Speicherfehler wie Buffer Overflows, Use-after-free oder Memory Leaks und behebe diese auch gleich falls der Code solche Fehler hat.

Erzeuge eine Tabelle in Latex mit den folgenden Spalten:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
-----	--------	------------------------	-----------	--------------	------------

Analysiere ausschließlich sicherheitsrelevante Aspekte, keine Stil- oder Architekturfragen.

```
```cpp
#define ENABLE_TEST_OUTPUT 1

#if ENABLE_TEST_OUTPUT
#include <sstream>
#include <iostream>
#include <codecvt>
#include <locale>

#define TEST_OUTPUT(x) do { \
    std::wstringstream wss; \
    wss << x; \
    std::wcout << "\t" << wss.str() << std::endl; \
} while (0)

#else
#define TEST_OUTPUT(x)
#endif

class TwoIntsClass
{
public:
    int intOne;
    int intTwo;
};

static const int flag1 = 1;
static const int flag2 = 0;

void test()
{
    TwoIntsClass* data;
```

```
data = NULL;
if (flag1)
{
    data = new TwoIntsClass[100];
    data[0].intOne = 0;
    data[0].intTwo = 0;
    TEST_OUTPUT(data[0].intOne);
    TEST_OUTPUT(data[0].intTwo);
}
if (flag1)
{
    ;
}
}
...
```

A.15.2 GPT-4o Antwort TC14_B

Hier ist die sicherheitsrelevante Analyse des gegebenen Codes in Bezug auf Speicherwaltungsfehler. Der Code enthält einen klassischen Memory Leak, da der mit `new[]` allokierte Speicher nicht freigegeben wird.

Analyse in Latex-Tabelle:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-401	Memory Leak	Ressourcenleck	Der mit <code>new[]</code> allokierte Speicher für das <code>TwoIntsClass</code> -Array wird nicht freigegeben. Dadurch entsteht ein Speicherleck, das bei häufiger Ausführung zu Ressourcenerschöpfung führen kann.	Speicher durch <code>delete[]</code> freigeben: <code>delete[] data;</code> nach der Nutzung einfügen. Alternativ Smart Pointer wie <code>std::unique_ptr</code> verwenden.

Tabelle 63: Analyse sicherheitsrelevanter Speicherfehler TC14_B ChatGPT-4o

Fixvorschlag (sicherheitsrelevant):

```

1 void test()
2 {
3     TwoIntsClass* data = nullptr;
4     if (flag1)
5     {
6         data = new TwoIntsClass[100];
7         data[0].intOne = 0;
8         data[0].intTwo = 0;
9         TEST_OUTPUT(data[0].intOne);
10        TEST_OUTPUT(data[0].intTwo);
11    }
12    if (flag1)
13    {
14        delete[] data; // Speicher korrekt freigeben
15    }
16 }
```

Listing 68: Korrigierter Codeausschnitt TC14_B ChatGPT-4o

Optional: moderne Lösung mit `std::unique_ptr<T[]>`:

```

1 #include <memory>
```

```

2 void test()
3 {
4     std::unique_ptr<TwoIntsClass []> data;
5     if (flag1)
6     {
7         data = std::make_unique<TwoIntsClass []>(100);
8         data[0].intOne = 0;
9         data[0].intTwo = 0;
10        TEST_OUTPUT(data[0].intOne);
11        TEST_OUTPUT(data[0].intTwo);
12    }
13    // Speicher wird automatisch freigegeben
14 }

```

Listing 69: Korrigierter Codeausschnitt TC14_B ChatGPT-4o mit Smart Pointer

A.15.3 Sonnet 3.7 Antwort TC14_B

Sicherheitsanalyse des C++-Codes

Ich habe den bereitgestellten C++-Code auf sicherheitsrelevante Speicherverwaltungsfehler analysiert. Bei der Untersuchung wurde ein kritischer Speicherfehler identifiziert, der zu Ressourcenlecks führen kann.

Identifizierte Schwachstellen

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-401	Memory Leak	Ressourcenleck	In der Funktion <code>test()</code> wird ein Array mit <code>new TwoIntsClass</code> allokiert, aber nie freigegeben. Da <code>flag1</code> auf 1 gesetzt ist, wird der Speicher immer allokiert, aber nirgends mit <code>delete[]</code> wieder freigegeben.	Den allokierten Speicher am Ende der Funktion mit <code>delete[]</code> freigeben oder Smart Pointer wie <code>std::vector</code> oder <code>std::unique_ptr</code> verwenden.

Tabelle 64: Analyse sicherheitsrelevanter Speicherfehler TC14_B Sonnet 3.7

Korrigierter Code

Hier ist der korrigierte Code, der das identifizierte Memory Leak behebt:

```

1     #define ENABLE_TEST_OUTPUT 1
2
3     #if ENABLE_TEST_OUTPUT
4     #include <sstream>
5     #include <iostream>
6     #include <codecvt>
7     #include <locale>
8
9     #define TEST_OUTPUT(x) do { \
10         std::wstringstream wss; \
11         wss << x; \

```

```
12     std::wcout << "\t" << wss.str() << std::endl; \
13 } while (0)
14
15 #else
16 #define TEST_OUTPUT(x)
17 #endif
18
19 class TwoIntsClass
20 {
21 public:
22     int intOne;
23     int intTwo;
24 };
25
26 static const int flag1 = 1;
27 static const int flag2 = 0;
28
29 void test()
30 {
31     TwoIntsClass* data;
32     data = NULL;
33     if (flag1)
34     {
35         data = new TwoIntsClass[100];
36         data[0].intOne = 0;
37         data[0].intTwo = 0;
38         TEST_OUTPUT(data[0].intOne);
39         TEST_OUTPUT(data[0].intTwo);
40     }
41     if (flag2)
42     {
43         ;
44     }
45
46     // Speicher freigeben, wenn er allokiert wurde
47     if (data != NULL)
48     {
49         delete[] data;
50         data = NULL;
51     }
52 }
```

Listing 70: Korrigierter Codeausschnitt TC14_B Sonnet 3.7

Alternative Lösung mit Smart Pointer:

Eine modernere und sicherere Lösung wäre die Verwendung von Smart Pointern oder Container-Klassen:

```
1 #include <memory>
2 #include <vector>
3
4 // ... (Rest des Codes bleibt gleich)
```

```
5
6 void test()
7 {
8     // Variante 1: Mit std::unique_ptr
9     if (flag1)
10    {
11        std::unique_ptr<TwoIntsClass[]> data(new TwoIntsClass[100])
12        ↪ ;
13        data[0].intOne = 0;
14        data[0].intTwo = 0;
15        TEST_OUTPUT(data[0].intOne);
16        TEST_OUTPUT(data[0].intTwo);
17    }
18
19    // ODER Variante 2: Mit std::vector (noch besser)
20    if (flag1)
21    {
22        std::vector<TwoIntsClass> data(100);
23        data[0].intOne = 0;
24        data[0].intTwo = 0;
25        TEST_OUTPUT(data[0].intOne);
26        TEST_OUTPUT(data[0].intTwo);
27    }
28
29    // Keine manuelle Speicherfreigabe nötig - erfolgt automatisch
30 }
```

Listing 71: Korrigierter Codeausschnitt TC14_B Sonnet 3.7 mit Smart Pointer

Die Verwendung von `std::vector` oder `std::unique_ptr` ist besonders empfehlenswert, da diese den Speicher automatisch freigeben, wenn sie außerhalb ihres Gültigkeitsbereichs gelangen, was die Wahrscheinlichkeit von Memory Leaks deutlich reduziert.

A.15.4 Gemini 3.7 Antwort TC14_B

Als Sicherheitsexperte mit Fokus auf C/C++ Codeanalyse, identifiziere und behebe ich speicherbezogene Sicherheitslücken im bereitgestellten Code. Ich liefere eine Tabelle im LaTeX-Format, die sicherheitsrelevante Aspekte, CWE-IDs, Schwachstellennamen, Fehlertypen, Beschreibungen und Empfehlungen zur Behebung der gefundenen Probleme enthält.

Hier ist die Analyse des Codes:

```
1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>
7 #include <locale>
8
9 #define TEST_OUTPUT(x) do { \
10     std::wstringstream wss; \
```



```
11     wss << x; \  
12     std::wcout << "\t" << wss.str() << std::endl; \  
13 } while (0)  
14  
15 #else  
16 #define TEST_OUTPUT(x)  
17 #endif  
18  
19 class TwoIntsClass  
20 {  
21 public:  
22     int intOne;  
23     int intTwo;  
24 };  
25  
26 static const int flag1 = 1;  
27 static const int flag2 = 0;  
28  
29 void test()  
30 {  
31     TwoIntsClass* data;  
32     data = NULL;  
33     if (flag1)  
34     {  
35         data = new TwoIntsClass[100];  
36         data[0].intOne = 0;  
37         data[0].intTwo = 0;  
38         TEST_OUTPUT(data[0].intOne);  
39         TEST_OUTPUT(data[0].intTwo);  
40     }  
41     if (flag2)  
42     {  
43         ;  
44     }  
45 }
```

Listing 72: Gemini 2.5 Pro hat den eingegebenen Code redundant nochmals ausgegeben

Analyse und Behebung von Sicherheitslücken:

Der Code enthält ein Memory Leak, da der allokierte Speicher für data nicht freigegeben wird.

Hier ist die korrigierte Version des Codes:

```
1 #define ENABLE_TEST_OUTPUT 1  
2  
3 #if ENABLE_TEST_OUTPUT  
4 #include <sstream>  
5 #include <iostream>  
6 #include <codecvt>  
7 #include <locale>  
8  
9 #define TEST_OUTPUT(x) do { \  
10     wss << x; \  
11     std::wcout << "\t" << wss.str() << std::endl; \  
12 } while (0)  
13  
14 #else  
15 #define TEST_OUTPUT(x)  
16 #endif  
17  
18 class TwoIntsClass  
19 {  
20 public:  
21     int intOne;  
22     int intTwo;  
23 };  
24  
25 static const int flag1 = 1;  
26 static const int flag2 = 0;  
27  
28 void test()  
29 {  
30     TwoIntsClass* data;  
31     data = NULL;  
32     if (flag1)  
33     {  
34         data = new TwoIntsClass[100];  
35         data[0].intOne = 0;  
36         data[0].intTwo = 0;  
37         TEST_OUTPUT(data[0].intOne);  
38         TEST_OUTPUT(data[0].intTwo);  
39     }  
40     if (flag2)  
41     {  
42         ;  
43     }  
44     delete data;  
45 }
```

```
10     std::wstringstream wss; \  
11     wss << x; \  
12     std::wcout << "\t" << wss.str() << std::endl; \  
13 } while (0)  
14  
15 #else  
16 #define TEST_OUTPUT(x)  
17 #endif  
18  
19 class TwoIntsClass  
20 {  
21 public:  
22     int intOne;  
23     int intTwo;  
24 };  
25  
26 static const int flag1 = 1;  
27 static const int flag2 = 0;  
28  
29 void test()  
30 {  
31     TwoIntsClass* data;  
32     data = NULL;  
33     if (flag1)  
34     {  
35         data = new TwoIntsClass[100];  
36         data[0].intOne = 0;  
37         data[0].intTwo = 0;  
38         TEST_OUTPUT(data[0].intOne);  
39         TEST_OUTPUT(data[0].intTwo);  
40         delete[] data; // Speicher freigeben  
41         data = NULL;  
42     }  
43     if (flag1)  
44     {  
45         ;  
46     }  
47 }
```

Listing 73: Korrigierter Codeausschnitt TC14_B Gemini 3.7

LaTeX Tabelle der Sicherheitsanalyse:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-401	Memory Leak	Speicherleck	Der durch 'new' allokierte Speicher wird nicht mit 'delete[]' freigegeben.	Verwenden Sie 'delete[] data;', um den Speicher freizugeben, nachdem er nicht mehr benötigt wird.

Tabelle 65: Analyse sicherheitsrelevanter Speicherfehler TC14_B Gemini 2.5 Pro

Diese Tabelle fasst die identifizierte Sicherheitslücke zusammen und gibt eine Empfehlung zu ihrer Behebung.

A.16 Testfall *TC15_B*

A.16.1 Prompt

Du bist ein Sicherheitsexperte mit Spezialisierung auf C/C++-Codeanalyse. Dein Ziel ist es, sicherheitsrelevante Speicherverwaltungsfehler zu erkennen und zu beheben. Analysiere den folgenden C/C++-Code ausschließlich im Hinblick auf sicherheitskritische Speicherfehler wie Buffer Overflows, Use-after-free oder Memory Leaks und behebe diese auch gleich falls der Code solche Fehler hat.

Erzeuge eine Tabelle in Latex mit den folgenden Spalten:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
-----	--------	------------------------	-----------	--------------	------------

Analysiere ausschließlich sicherheitsrelevante Aspekte, keine Stil- oder Architekturfragen.

```
```cpp
#define ENABLE_TEST_OUTPUT 1

#if ENABLE_TEST_OUTPUT
#include <sstream>
#include <iostream>
#include <codecvt>
#include <locale>

#define TEST_OUTPUT(x) do { \
 std::wstringstream wss; \
 wss << x; \
 std::wcout << "\t" << wss.str() << std::endl; \
} while (0)

#else
#define TEST_OUTPUT(x)
#endif

class HelperBase
{
public:
 HelperBase()
 {
 TEST_OUTPUT("Constructor: HelperBase");
 }

 ~HelperBase()
 {
 TEST_OUTPUT("Destructor : HelperBase");
 }
};
```

```
class HelperDerived : public HelperBase
{
public:
 HelperDerived(const char* name)
 {
 TEST_OUTPUT("Constructor: HelperDerived");
 if (name)
 {
 this->name = new char[strlen(name) + 1];
 strcpy(this->name, name);
 TEST_OUTPUT(this->name);
 }
 else
 {
 this->name = new char[1];
 *(this->name) = '\\0';
 }
 }

 ~HelperDerived()
 {
 TEST_OUTPUT("Destructor : HelperDerived");
 delete[] name;
 }

 HelperDerived(HelperDerived& derivedClassObject)
 {
 this->name = new char[strlen(derivedClassObject.name) + 1];
 strcpy(this->name, derivedClassObject.name);
 }

 HelperDerived& operator=(const HelperDerived& derivedClassObject)
 {
 if (&derivedClassObject != this)
 {
 this->name = new char[strlen(derivedClassObject.name) + 1];
 strcpy(this->name, derivedClassObject.name);
 }
 return *this;
 }

private:
 char* name;
};

void test()
{
```

```
 HelperBase* baseClassObject = new HelperDerived("BadClass");
 delete baseClassObject;
}

...
```

### **A.16.2 GPT-4o Antwort TC15\_B**

Hier ist die sicherheitstechnische Analyse des gegebenen C++-Codes. Der Fokus liegt auf Speicherverwaltungsfehlern wie Memory Leaks, Use-after-free und Buffer Overflows. Der Code enthält eine kritische Stelle, bei der eine Basisklassen-Pointer ohne virtuellen Destruktor verwendet wird, was zu undefiniertem Verhalten führen kann.

**Analyse: und Fix in LaTeX-Tabelle**

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-416	Use After Free / Undefiniertes Verhalten	Fehlerhafte Objektzerstörung	In der Funktion <code>test()</code> wird ein Objekt vom Typ <code>HelperDerived</code> über einen Zeiger auf die Basisklasse <code>HelperBase</code> gelöscht. Da <code>HelperBase</code> keinen virtuellen Destruktor hat, wird nur der Destruktor von <code>HelperBase</code> aufgerufen, nicht aber der von <code>HelperDerived</code> . Dies führt zu undefiniertem Verhalten und potenziellen Speicherlecks.	Füge einen virtuellen Destruktor in <code>HelperBase</code> ein: <code>virtual ~HelperBase() {}</code> . Nur so wird bei polymorpher Verwendung der richtige Destruktor aufgerufen.
2	CWE-415	Double Allocation / Speicherleck (Copy/Assignment)	Memory Leak	Im Zuweisungsoperator <code>operator=</code> von <code>HelperDerived</code> wird zuvor allozierter Speicher von <code>name</code> nicht freigegeben, bevor neuer Speicher alloziert wird. Dies führt bei mehrfacher Zuweisung zu einem Speicherleck.	Vor der erneuten Allokation sollte vorhandener Speicher freigegeben werden: <code>delete[] name;</code> vor dem <code>new</code> . Zudem sollte der Operator Selbstzuweisung absichern und korrektes Deep Copy-Verhalten sicherstellen.

Tabelle 66: Analyse sicherheitsrelevanter Speicherfehler TC15\_B ChatGPT-4o

### Empfohlene Fixes im Code:

Virtuellen Destruktor hinzufügen:

```
1 virtual ~HelperBase()
```



```
2 {
3 TEST_OUTPUT("Destructor : HelperBase");
4 }
```

Listing 74: Korrigierter Codeausschnitt mit virtuellem Destruktor TC15\_B ChatGPT-4o  
Fix für den Copy Assignment Operator:

```
1 HelperDerived& operator=(const HelperDerived& derivedClassObject)
2 {
3 if (&derivedClassObject != this)
4 {
5 delete[] this->name;
6 this->name = new char[strlen(derivedClassObject.name) + 1];
7 strcpy(this->name, derivedClassObject.name);
8 }
9 return *this;
10 }
```

Listing 75: Korrigierter Codeausschnitt für Copy Assignment Operator TC15\_B ChatGPT-4o

Wenn du den gesamten Code mit den Fixes brauchst, sag einfach Bescheid.

### A.16.3 Sonnet 3.7 Antwort TC15\_B

Ich habe den C++-Code auf sicherheitsrelevante Speicherverwaltungsfehler analysiert und mehrere kritische Probleme identifiziert. Hier ist meine detaillierte Analyse:

#### Identifizierte Sicherheitsprobleme:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-401	Memory Leak	Ressourcenleck	Im Zuweisungsoperator wird der alte Speicher nicht freigegeben, bevor neuer zugewiesen wird.	Vorhandenen Speicher mit <code>delete[]</code> freigeben.
2	CWE-416	Use After Free	Speicherzugriffsfehler	Im Kopierkonstruktor und Zuweisungsoperator fehlt die Tiefenkopie der Basisklasse.	Basiskonstruktor korrekt aufrufen und Speicher korrekt duplizieren.
3	CWE-665	Improper Initialization	Initialisierungsfehler	Im Kopierkonstruktor wird <code>name</code> nicht initialisiert, wenn <code>derivedClassObject.name</code> NULL ist.	Null-Prüfung hinzufügen und ggf. mit leerem String initialisieren.
4	CWE-762	Mismatched Memory Management Routines	Speicherverwaltungsfehler	Der virtuelle Destruktor fehlt in der Basisklasse.	Virtuellen Destruktor in <code>HelperBase</code> hinzufügen, um korrektes Löschen über Basisklassenzeiger zu ermöglichen.

Tabelle 67: Analyse sicherheitsrelevanter Speicherfehler TC15\_B Sonnet 3.7

#### Detaillierte Erklärung und Korrekturen:

Im Zuweisungsoperator wird der bereits zugewiesene Speicher für `name` nicht freigegeben, bevor neuer Speicher zugewiesen wird. Dies führt zu einem Speicherleck.

```
1 HelperDerived& operator=(const HelperDerived& derivedClassObject)
2 {
3 if (&derivedClassObject != this)
4 {
5 delete[] name; // Hier fehlte die Freigabe des alten
6 ↪ Speichers
7 this->name = new char[strlen(derivedClassObject.name) + 1];
8 strcpy(this->name, derivedClassObject.name);
9 }
10 return *this;
11 }
```

Listing 76: Korrektur des Zuweisungsoperators TC15\_B Sonnet 3.7

Die Basisklasse HelperBase hat keinen virtuellen Destruktor. Wenn ein Zeiger auf die Basisklasse verwendet wird, um ein Objekt der abgeleiteten Klasse zu löschen (wie in der test()-Funktion), wird nur der Destruktor der Basisklasse aufgerufen, nicht aber der der abgeleiteten Klasse. Dies führt zu einem Memory Leak, da der Speicher für name nicht freigegeben wird.

```
1 class HelperBase
2 {
3 public:
4 HelperBase()
5 {
6 TEST_OUTPUT("Constructor: HelperBase");
7 }
8
9 virtual ~HelperBase() // Hier muss virtual hinzugefügt werden
10 {
11 TEST_OUTPUT("Destructor : HelperBase");
12 }
13 };
```

Listing 77: Hinzufügen eines virtuellen Destruktors TC15\_B Sonnet 3.7

Der Kopierkonstruktor ruft nicht den Konstruktor der Basisklasse auf und prüft nicht auf NULL-Pointer.

```
1 HelperDerived(HelperDerived& derivedClassObject) : HelperBase() //
2 ↪ Basisklassenkonstruktor aufrufen
3 {
4 if (derivedClassObject.name) {
5 this->name = new char[strlen(derivedClassObject.name) + 1];
6 strcpy(this->name, derivedClassObject.name);
7 } else {
8 this->name = new char[1];
9 *(this->name) = '\\0';
10 }
11 }
```

Listing 78: Korrektur des Kopierkonstruktors TC15\_B Sonnet 3.7

Es gibt keine Überprüfung, ob die Speicherzuweisung mit new erfolgreich war. Bei Speichermangel könnte dies zu undefiniertem Verhalten führen.

### Korrigierter Code:

Hier ist der vollständig korrigierte Code:

```
1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>
7 #include <locale>
8 #include <cstring> // Für strlen und strcpy hinzugefügt
9
10 #define TEST_OUTPUT(x) do { \
11 std::wstringstream wss; \
12 wss << x; \
13 std::wcout << "\t" << wss.str() << std::endl; \
14 } while (0)
15
16 #else
17 #define TEST_OUTPUT(x)
18 #endif
19
20 class HelperBase
21 {
22 public:
23 HelperBase()
24 {
25 TEST_OUTPUT("Constructor: HelperBase");
26 }
27
28 virtual ~HelperBase() // Virtueller Destruktor hinzugefügt
29 {
30 TEST_OUTPUT("Destructor : HelperBase");
31 }
32 };
33
34 class HelperDerived : public HelperBase
35 {
36 public:
37 HelperDerived(const char* name)
38 {
39 TEST_OUTPUT("Constructor: HelperDerived");
40 if (name)
41 {
42 this->name = new char[strlen(name) + 1];
43 strcpy(this->name, name);
44 TEST_OUTPUT(this->name);
45 }
46 else
```

```
47 {
48 this->name = new char[1];
49 *(this->name) = '\\0';
50 }
51 }
52
53 ~HelperDerived()
54 {
55 TEST_OUTPUT("Destructor : HelperDerived");
56 delete[] name;
57 }
58
59 HelperDerived(const HelperDerived& derivedClassObject) :
60 ⇨ HelperBase() // Basisklassenkonstruktor aufrufen
61 {
62 if (derivedClassObject.name)
63 {
64 this->name = new char[strlen(derivedClassObject.name) +
65 ⇨ 1];
66 strcpy(this->name, derivedClassObject.name);
67 }
68 else
69 {
70 this->name = new char[1];
71 *(this->name) = '\\0';
72 }
73 }
74
75 HelperDerived& operator=(const HelperDerived&
76 ⇨ derivedClassObject)
77 {
78 if (&derivedClassObject != this)
79 {
80 delete[] name; // Alten Speicher freigeben
81
82 if (derivedClassObject.name)
83 {
84 this->name = new char[strlen(derivedClassObject.
85 ⇨ name) + 1];
86 strcpy(this->name, derivedClassObject.name);
87 }
88 else
89 {
90 this->name = new char[1];
91 *(this->name) = '\\0';
92 }
93 }
94 return *this;
95 }
96
97 private:
```

```
94 char* name;
95 };
96
97 void test()
98 {
99 HelperBase* baseClassObject = new HelperDerived("BadClass");
100 delete baseClassObject; // Jetzt wird auch der Destruktor von
 ↪ HelperDerived aufgerufen
101 }
```

Listing 79: Vollständig korrigierter Code TC15\_B Sonnet 3.7

### Zusätzliche Empfehlungen

1. Verwenden Sie moderne C++-Konstrukte wie `std::string` anstelle von rohen Zeigern, um Speicherverwaltungsprobleme zu vermeiden.
2. Implementieren Sie die Rule of Three/Five/Zero vollständig.
3. Erwägen Sie die Verwendung von Smart Pointern wie `std::unique_ptr` für eine automatische Ressourcenverwaltung.
4. Fügen Sie Fehlerbehandlung für Speicherzuweisungen hinzu.

#### A.16.4 Gemini 3.7 Antwort TC15\_B

Als Sicherheitsexperte, der sich auf die Analyse von C/C++-Code spezialisiert hat, konzentriere ich mich auf sicherheitsrelevante Speicherverwaltungsfehler im bereitgestellten Code.

Hier ist die LaTeX-Tabelle, die die identifizierten Schwachstellen und Empfehlungen zusammenfasst:

Nr.	CWE-ID	Name der Schwachstelle	Fehlerart	Beschreibung	Empfehlung
1	CWE-415	Double Free	Speicherfehler	In der Funktion <code>test()</code> wird ein Objekt der Klasse <code>HelperDerived</code> mit <code>new</code> erstellt und einem <code>HelperBase*</code> -Zeiger zugewiesen. Beim Aufruf von <code>delete baseClassObject</code> wird nur der Destruktor von <code>HelperBase</code> , aber nicht der von <code>HelperDerived</code> aufgerufen. Dies kann zu einem Double Free führen, wenn der Destruktor der abgeleiteten Klasse nicht korrekt freigibt.	Verwenden Sie einen <code>virtual</code> Destruktor in der Basisklasse, um sicherzustellen, dass beim Löschen über den Basisklassenzeiger auch der Destruktor der abgeleiteten Klasse ausgeführt wird.
2	CWE-762	Missing Release of Memory after Effective Lifetime	Speicherleck	Im Zuweisungsoperator <code>HelperDerived&amp; operator=(const HelperDerived&amp; derivedClassObject)</code> und im Copy-Konstruktor <code>HelperDerived(const HelperDerived&amp; derivedClassObject)</code> wird neuer Speicher für <code>this-&gt;name</code> allokiert, ohne den zuvor allokierten Speicher freizugeben. Dies führt zu einem Speicherleck, wenn <code>name</code> bereits gesetzt war.	Vor der Zuweisung von neuem Speicher sollte der bisherige Speicher mit <code>delete[] this-&gt;name</code> freigegeben werden. Alternativ sollten <code>std::string</code> oder Smart Pointer verwendet werden.
3	CWE-121	Stack-based Buffer Overflow	Pufferüberlauf	Im Copy-Konstruktor und im Zuweisungsoperator wird Speicher für <code>this-&gt;name</code> basierend auf der Länge von <code>derivedClassObject.name</code> allokiert. Wird <code>strcpy</code> zum Kopieren verwendet, kann es bei ungewöhnlich langen Strings zu einem Pufferüberlauf kommen.	Verwenden Sie sichere Alternativen wie <code>strncpy</code> oder besser <code>std::string</code> . Überprüfen Sie stets die Länge des Quellpuffers vor der Speicherzuweisung und dem Kopieren.

Tabelle 68: Analyse sicherheitsrelevanter Speicherfehler TC15\_B Gemini 2.5 Pro

Der korrigierte Code sieht wie folgt aus:

```

1 #define ENABLE_TEST_OUTPUT 1
2
3 #if ENABLE_TEST_OUTPUT
4 #include <sstream>
5 #include <iostream>
6 #include <codecvt>
7 #include <locale>
8
9 #define TEST_OUTPUT(x) do { \
10 std::wstringstream wss; \
11 wss << x; \
12 std::wcout << "\t" << wss.str() << std::endl; \
13 } while (0)

```

```
14
15 #else
16 #define TEST_OUTPUT(x)
17 #endif
18
19 #include <cstring>
20
21 class HelperBase
22 {
23 public:
24 HelperBase()
25 {
26 TEST_OUTPUT("Constructor: HelperBase");
27 }
28
29 virtual ~HelperBase()
30 {
31 TEST_OUTPUT("Destructor : HelperBase");
32 }
33 };
34
35 class HelperDerived : public HelperBase
36 {
37 public:
38 HelperDerived(const char* name)
39 {
40 TEST_OUTPUT("Constructor: HelperDerived");
41 if (name)
42 {
43 this->name = new char[strlen(name) + 1];
44 strcpy_s(this->name, strlen(name) + 1, name);
45 TEST_OUTPUT(this->name);
46 }
47 else
48 {
49 this->name = new char[1];
50 *(this->name) = '\\0';
51 }
52 }
53
54 ~HelperDerived()
55 {
56 TEST_OUTPUT("Destructor : HelperDerived");
57 delete[] name;
58 }
59
60 HelperDerived(HelperDerived& derivedClassObject)
61 {
62 this->name = new char[strlen(derivedClassObject.name) + 1];
63 strcpy_s(this->name, strlen(derivedClassObject.name) + 1,
 ↪ derivedClassObject.name);
```

```
64 }
65
66 HelperDerived& operator=(const HelperDerived&
67 ↪ derivedClassObject)
68 {
69 if (&derivedClassObject != this)
70 {
71 delete[] this->name;
72 this->name = new char[strlen(derivedClassObject.name) +
73 ↪ 1];
74 strcpy_s(this->name, strlen(derivedClassObject.name) +
75 ↪ 1, derivedClassObject.name);
76 }
77 return *this;
78 }
79
80 private:
81 char* name;
82 };
83
84 void test()
85 {
86 HelperBase* baseClassObject = new HelperDerived("BadClass");
87 delete baseClassObject;
88 }
```

Listing 80: Korrigierter Code TC15\_B Gemini 2.5 Pro



## B Clang Output

### B.1 Testcase TC1\_B

```
1 KEINE ERKENNUNG SEITENS CLANG --> LEER
```

### B.2 Testcase TC2\_B

```
1 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:76:5: warning: Call to function 'strcpy' is
 ↳ insecure as it does not provide bounding of the memory buffer.
 ↳ Replace unbounded copy functions with analogous functions that
 ↳ support length arguments such as 'strncpy'. CWE-119
 ↳ [clang-analyzer-security.insecureAPI strcpy]
2 76 | strcpy(buf, str); // potential FLAW depending on the
 ↳ size of str
 | ~~~~~
3
4 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:76:5: note: Call to function 'strcpy' is
 ↳ insecure as it does not provide bounding of the memory buffer.
 ↳ Replace unbounded copy functions with analogous functions that
 ↳ support length arguments such as 'strncpy'. CWE-119
5 76 | strcpy(buf, str); // potential FLAW depending on the
 ↳ size of str
 | ~~~~~
6
7 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:76:5: warning: 'strcpy' is deprecated: This
 ↳ function or variable may be unsafe. Consider using strcpy_s
 ↳ instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS.
 ↳ See online help for details.
 ↳ [clang-diagnostic-deprecated-declarations]
8 76 | strcpy(buf, str); // potential FLAW depending on the
 ↳ size of str
 | ^
9
10 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\string.h:130:1: note: 'strcpy' has been explicitly marked
 ↳ deprecated here
11 130 | __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1(
 | ^
12
13 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\corecrt.h:835:5: note: expanded from macro
 ↳ '__DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1'
14 835 |
 ↳ __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX(_ReturnType,
 ↳ _ReturnPolicy, _DeclSpec, _FuncName, _FuncName##_s,
 ↳ _DstType, _SalAttributeDst, _DstType, _Dst, _TType1, _TArg1)
 | ^
15
16 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\corecrt.h:1894:17: note: expanded from macro
 ↳ '__DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX'
```

```

17 1894 | _CRT_INSECURE_DEPRECATED(_SecureFuncName)
 ↳ _DeclSpec _ReturnType __cdecl _FuncName(_SalAttributeDst
 ↳ _DstType *_Dst, _TType1 _TArg1);
18 |
19 C:\Program Files\Microsoft Visual Studio\2022\Professional\VC\
 ↳ Tools\MSVC\14.44.35207\include\vcruntime.h:368:55: note:
 ↳ expanded from macro '_CRT_INSECURE_DEPRECATED'
20 368 | #define _CRT_INSECURE_DEPRECATED(_Replacement)
 ↳ _CRT_DEPRECATED_TEXT(\
21 |
22 C:\Program Files\Microsoft Visual Studio\2022\Professional\VC\
 ↳ Tools\MSVC\14.44.35207\include\vcruntime.h:358:47: note:
 ↳ expanded from macro '_CRT_DEPRECATED_TEXT'
23 358 | #define _CRT_DEPRECATED_TEXT(_Text)
 ↳ __declspec(deprecated(_Text))
24 |

```

### B.3 Testcase TC3\_B

```

1 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:111:3: warning: 'wcsncpy' is deprecated: This
 ↳ function or variable may be unsafe. Consider using wcsncpy_s
 ↳ instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS.
 ↳ See online help for details.
 ↳ [clang-diagnostic-deprecated-declarations]
2 111 | wcsncpy(data, source, SIZE - 1);
3 |
4 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\corecrt_wstring.h:200:1: note: 'wcsncpy' has been
 ↳ explicitly marked deprecated here
5 200 | __DEFINE_CPP_OVERLOAD_STANDARD_NFUNC_0_2_EX(
6 |
7 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\corecrt.h:1935:17: note: expanded from macro
 ↳ '__DEFINE_CPP_OVERLOAD_STANDARD_NFUNC_0_2_EX'
8 1935 | _CRT_INSECURE_DEPRECATED(_SecureFuncName)
 ↳ _DeclSpec _ReturnType __cdecl _FuncName(_SalAttributeDst
 ↳ _DstType *_Dst, _TType1 _TArg1, _TType2 _TArg2);
9 |
10 C:\Program Files\Microsoft Visual Studio\2022\Professional\VC\
 ↳ Tools\MSVC\14.44.35207\include\vcruntime.h:368:55: note:
 ↳ expanded from macro '_CRT_INSECURE_DEPRECATED'
11 368 | #define _CRT_INSECURE_DEPRECATED(_Replacement)
 ↳ _CRT_DEPRECATED_TEXT(\
12 |
13 C:\Program Files\Microsoft Visual Studio\2022\Professional\VC\
 ↳ Tools\MSVC\14.44.35207\include\vcruntime.h:358:47: note:
 ↳ expanded from macro '_CRT_DEPRECATED_TEXT'
14 358 | #define _CRT_DEPRECATED_TEXT(_Text)
 ↳ __declspec(deprecated(_Text))

```

```

15 |
16 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:129:2: warning: Argument to 'free()' is
 ↳ offset by -16 bytes from the start of memory allocated by
 ↳ 'malloc()' [clang-analyzer-unix.Malloc]
17 129 | free(data);
18 | ^~~~~~
19 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:122:13: note: Assuming 'dataBuffer' is not
 ↳ equal to NULL
20 122 | if (dataBuffer == NULL) { exit(-1); }
21 | ~~~~~
22 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:122:9: note: Taking false branch
23 122 | if (dataBuffer == NULL) { exit(-1); }
24 | ^
25 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:129:2: note: Argument to 'free()' is offset
 ↳ by -16 bytes from the start of memory allocated by 'malloc()'
26 129 | free(data);
27 | ^~~~~~

```

## B.4 Testcase TC4\_B

```

1 KEINE ERKENNUNG SEITENS CLANG --> LEER

```

## B.5 Testcase TC5\_B

```

1 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:219:2: warning: Value stored to 'data' is
 ↳ never read [clang-analyzer-deadcode.DeadStores]
2 219 | data = dataBadBuffer;
3 | ^~~~~~
4 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:219:2: note: Value stored to 'data' is never
 ↳ read
5 219 | data = dataBadBuffer;
6 | ^~~~~~
7 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:224:29: warning: Storage provided to
 ↳ placement new is only 4 bytes, whereas the allocated type
 ↳ requires 8 bytes [clang-analyzer-cplusplus.PlacementNew]
8 224 | TwoIntsClass* classTwo = new(data)
 ↳ TwoIntsClass;
9 | ~~~~~
10 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:215:2: note: 'data' declared without an
 ↳ initial value

```

```

11 215 | char* data;
12 | ~~~~~
13 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:216:2: note: 'dataRef' initialized here
14 216 | char*& dataRef = data;
15 | ~~~~~
16 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:217:2: note: 'dataBadBuffer' initialized here
17 217 | char dataBadBuffer[sizeof(OneIntClass)];
18 | ~~~~~
19 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:219:2: note: Value assigned to 'data'
20 219 | data = dataBadBuffer;
21 | ~~~~~
22 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:221:3: note: 'data' initialized here
23 221 | char* data = dataRef;
24 | ~~~~~
25 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:224:29: note: Storage provided to placement
 ↳ new is only 4 bytes, whereas the allocated type requires 8
 ↳ bytes
26 224 | TwoIntsClass* classTwo = new(data)
 ↳ TwoIntsClass;
27 | ~~~~~
28 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:242:2: warning: Value stored to 'data' is
 ↳ never read [clang-analyzer-deadcode.DeadStores]
29 242 | data = dataGoodBuffer;
30 | ~~~~~
31 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:242:2: note: Value stored to 'data' is never
 ↳ read
32 242 | data = dataGoodBuffer;
33 | ~~~~~
34 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:246:29: warning: Storage type is aligned to 1
 ↳ bytes but allocated type is aligned to 4 bytes
 ↳ [clang-analyzer-cplusplus.PlacementNew]
35 246 | TwoIntsClass* classTwo = new(data)
 ↳ TwoIntsClass;
36 | ~~~~~
37 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:238:2: note: 'data' declared without an
 ↳ initial value
38 238 | char* data;
39 | ~~~~~
40 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:239:2: note: 'dataRef' initialized here

```

```

41 239 | char*& dataRef = data;
42 | ~~~~~
43 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:240:2: note: 'dataGoodBuffer' initialized here
44 240 | char dataGoodBuffer[sizeof(TwoIntsClass)];
45 | ~~~~~
46 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:242:2: note: Value assigned to 'data'
47 242 | data = dataGoodBuffer;
48 | ~~~~~
49 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:244:3: note: 'data' initialized here
50 244 | char* data = dataRef;
51 | ~~~~~
52 C:\Users\z0041eeu\Desktop\Testumgebung\src\buffer_overflow\
 ↳ buffer_tests.cpp:246:29: note: Storage type is aligned to 1
 ↳ bytes but allocated type is aligned to 4 bytes
53 246 | TwoIntsClass* classTwo = new(data)
 ↳ TwoIntsClass;
54 | ~~~~~

```

## B.6 Testcase TC6\_B

```

1 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:120:9: warning: Use of memory after it is
 ↳ freed [clang-analyzer-unix.Malloc]
2 120 | TEST_OUTPUT(data);
3 | ~
4 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ ../helper/test_output.h:11:5: note: expanded from macro
 ↳ 'TEST_OUTPUT'
5 11 | wss << x; \
6 | ~ ~
7 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:108:9: note: GLOBAL_CONST_FIVE is equal to
 ↳ 5
8 108 | if (GLOBAL_CONST_FIVE == 5)
9 | ~
10 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:4:27: note: expanded from macro
 ↳ 'GLOBAL_CONST_FIVE'
11 4 | #define GLOBAL_CONST_FIVE 5
12 | ~
13 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:108:5: note: Taking true branch
14 108 | if (GLOBAL_CONST_FIVE == 5)
15 | ~
16 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:110:26: note: Memory is allocated

```

```

17 110 | data = (wchar_t*)malloc(100 * sizeof(wchar_t));
18 | ~~~~~
19 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:111:13: note: Assuming 'data' is not equal
 ↳ to NULL
20 111 | if (data == NULL) { exit(-1); }
21 | ~~~~~
22 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:111:9: note: Taking false branch
23 111 | if (data == NULL) { exit(-1); }
24 | ^
25 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:115:9: note: Memory is released
26 115 | free(data);
27 | ~~~~~
28 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:117:9: note: GLOBAL_CONST_FIVE is equal to
 ↳ 5
29 117 | if (GLOBAL_CONST_FIVE == 5)
30 | ^
31 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:4:27: note: expanded from macro
 ↳ 'GLOBAL_CONST_FIVE'
32 4 | #define GLOBAL_CONST_FIVE 5
33 | ^
34 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:117:5: note: Taking true branch
35 117 | if (GLOBAL_CONST_FIVE == 5)
36 | ^
37 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:120:9: note: Use of memory after it is
 ↳ freed
38 120 | TEST_OUTPUT(data);
39 | ^

```

## B.7 Testcase TC7\_B

```
1 KEINE ERKENNUNG SEITENS CLANG --> LEER
```

## B.8 Testcase TC8\_B

```

1 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:21:13: warning: Call to function 'strcpy'
 ↳ is insecure as it does not provide bounding of the memory
 ↳ buffer. Replace unbounded copy functions with analogous
 ↳ functions that support length arguments such as 'strncpy'.
 ↳ CWE-119 [clang-analyzer-security.insecureAPI.strcpy]
2 21 | strcpy(this->name, name);
3 | ~~~~~

```

```
4 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:21:13: note: Call to function 'strcpy' is
 ↳ insecure as it does not provide bounding of the memory buffer.
 ↳ Replace unbounded copy functions with analogous functions that
 ↳ support length arguments such as 'strncpy'. CWE-119
5 21 | strcpy(this->name, name);
6 | ~~~~~
7 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:21:13: warning: 'strcpy' is deprecated:
 ↳ This function or variable may be unsafe. Consider using
 ↳ strncpy_s instead. To disable deprecation, use
 ↳ _CRT_SECURE_NO_WARNINGS. See online help for details.
 ↳ [clang-diagnostic-deprecated-declarations]
8 21 | strcpy(this->name, name);
9 | ^
10 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\string.h:130:1: note: 'strcpy' has been explicitly marked
 ↳ deprecated here
11 130 | __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1(
12 | ^
13 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\corecrt.h:835:5: note: expanded from macro
 ↳ '__DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1'
14 835 |
 ↳ __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX(_ReturnType,
 ↳ _ReturnPolicy, _DeclSpec, _FuncName, _FuncName##_s,
 ↳ _DstType, _SalAttributeDst, _DstType, _Dst, _TType1, _TArg1)
15 | ^
16 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\corecrt.h:1894:17: note: expanded from macro
 ↳ '__DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX'
17 1894 | _CRT_INSECURE_DEPRECATED(_SecureFuncName)
 ↳ _DeclSpec _ReturnType __cdecl _FuncName(_SalAttributeDst
 ↳ _DstType *_Dst, _TType1 _TArg1);
18 | ^
19 C:\Program Files\Microsoft Visual Studio\2022\Professional\VC\
 ↳ Tools\MSVC\14.44.35207\include\vcruntime.h:368:55: note:
 ↳ expanded from macro '_CRT_INSECURE_DEPRECATED'
20 368 | #define _CRT_INSECURE_DEPRECATED(_Replacement)
 ↳ _CRT_DEPRECATED_TEXT(\br/>21 | ^
22 C:\Program Files\Microsoft Visual Studio\2022\Professional\VC\
 ↳ Tools\MSVC\14.44.35207\include\vcruntime.h:358:47: note:
 ↳ expanded from macro '_CRT_DEPRECATED_TEXT'
23 358 | #define _CRT_DEPRECATED_TEXT(_Text)
 ↳ __declspec(deprecated(_Text))
24 | ^
25 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:39:9: warning: Call to function 'strcpy'
 ↳ is insecure as it does not provide bounding of the memory
 ↳ buffer. Replace unbounded copy functions with analogous
```



```

 ↪ functions that support length arguments such as 'strncpy'.
 ↪ CWE-119 [clang-analyzer-security.insecureAPI.strcpy]
26 39 | strcpy(this->name, badClassObject.name);
27 | ~~~~~
28 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↪ use_after_tests.cpp:39:9: note: Call to function 'strcpy' is
 ↪ insecure as it does not provide bounding of the memory buffer.
 ↪ Replace unbounded copy functions with analogous functions that
 ↪ support length arguments such as 'strncpy'. CWE-119
29 39 | strcpy(this->name, badClassObject.name);
30 | ~~~~~
31 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↪ use_after_tests.cpp:39:9: warning: 'strcpy' is deprecated:
 ↪ This function or variable may be unsafe. Consider using
 ↪ strcpy_s instead. To disable deprecation, use
 ↪ _CRT_SECURE_NO_WARNINGS. See online help for details.
 ↪ [clang-diagnostic-deprecated-declarations]
32 39 | strcpy(this->name, badClassObject.name);
33 | ~
34 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↪ ucrt\string.h:130:1: note: 'strcpy' has been explicitly marked
 ↪ deprecated here
35 130 | __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1(
36 | ~
37 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↪ ucrt\corecrt.h:835:5: note: expanded from macro
 ↪ ' __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1 '
38 835 |
 ↪ __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX(_ReturnType,
 ↪ _ReturnPolicy, _DeclSpec, _FuncName, _FuncName##_s,
 ↪ _DstType, _SalAttributeDst, _DstType, _Dst, _TType1, _TArg1)
39 | ~
40 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↪ ucrt\corecrt.h:1894:17: note: expanded from macro
 ↪ ' __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX '
41 1894 | _CRT_INSECURE_DEPRECATED(_SecureFuncName)
 ↪ _DeclSpec _ReturnType __cdecl _FuncName(_SalAttributeDst
 ↪ _DstType *_Dst, _TType1 _TArg1);
42 | ~
43 C:\Program Files\Microsoft Visual Studio\2022\Professional\
 ↪ VC\Tools\MSVC\14.44.35207\include\vcruntime.h:368:55: note:
 ↪ expanded from macro '_CRT_INSECURE_DEPRECATED'
44 368 | #define _CRT_INSECURE_DEPRECATED(_Replacement)
 ↪ _CRT_DEPRECATED_TEXT(
45 | ~
46 C:\Program Files\Microsoft Visual Studio\2022\Professional\
 ↪ VC\Tools\MSVC\14.44.35207\include\vcruntime.h:358:47: note:
 ↪ expanded from macro '_CRT_DEPRECATED_TEXT'
47 358 | #define _CRT_DEPRECATED_TEXT(_Text)
 ↪ __declspec(deprecated(_Text))
48 | ~

```



```

49 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:46:31: warning: Use of memory after it is
 ↳ freed [clang-analyzer-cplusplus.NewDelete]
50 46 | this->name = new char[strlen(badClassObject.name)
 ↳ + 1];
51 | ~~~~~
52 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:42:5: note: Assuming badClassObject ==
 ↳ *this
53 42 | BadClass& operator=(const BadClass& badClassObject)
54 | ~~~~~
55 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:45:9: note: Memory is released
56 45 | delete[] this->name;
57 | ~~~~~
58 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:46:31: note: Use of memory after it is
 ↳ freed
59 46 | this->name = new char[strlen(badClassObject.name)
 ↳ + 1];
60 | ~~~~~
61 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:47:9: warning: Call to function 'strcpy'
 ↳ is insecure as it does not provide bounding of the memory
 ↳ buffer. Replace unbounded copy functions with analogous
 ↳ functions that support length arguments such as 'strncpy'.
 ↳ CWE-119 [clang-analyzer-security.insecureAPI.strcpy]
62 47 | strcpy(this->name, badClassObject.name); /* FLAW -
 ↳ if this is a self-assignment,
63 | ~~~~~
64 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:47:9: note: Call to function 'strcpy' is
 ↳ insecure as it does not provide bounding of the memory buffer.
 ↳ Replace unbounded copy functions with analogous functions that
 ↳ support length arguments such as 'strncpy'. CWE-119
65 47 | strcpy(this->name, badClassObject.name); /* FLAW -
 ↳ if this is a self-assignment,
66 | ~~~~~

```

## B.9 Testcase TC9\_B

```
1 KEINE ERKENNUNG SEITENS CLANG --> LEER
```

## B.10 Testcase TC10\_B

```

1 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:284:9: note: Assuming the condition is true
2 284 | if (globalReturnsTrueOrFalse())
3 | ~~~~~

```

```
4 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:284:5: note: Taking true branch
5 284 | if (globalReturnsTrueOrFalse())
6 | ^
7 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:286:16: note: Memory is allocated
8 286 | data = new TwoIntsClass;
9 | ~~~~~
10 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:290:9: note: Memory is released
11 290 | delete data;
12 | ~~~~~
13 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:299:9: note: Assuming the condition is true
14 299 | if (globalReturnsTrueOrFalse())
15 | ~~~~~
16 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:299:5: note: Taking true branch
17 299 | if (globalReturnsTrueOrFalse())
18 | ^
19 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:302:21: note: Use of memory after it is
 ↳ freed
20 302 | TEST_OUTPUT(data->intOne);
21 | ^
22 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ ../helper/test_output.h:11:12: note: expanded from macro
 ↳ 'TEST_OUTPUT'
23 11 | wss << x; \
24 | ^
25 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:312:1: warning: Potential leak of memory
 ↳ pointed to by 'data' [clang-analyzer-cplusplus.NewDeleteLeaks]
26 312 | }
27 | ^
28 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:284:9: note: Assuming the condition is
 ↳ false
29 284 | if (globalReturnsTrueOrFalse())
30 | ~~~~~
31 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:284:5: note: Taking false branch
32 284 | if (globalReturnsTrueOrFalse())
33 | ^
34 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:294:16: note: Memory is allocated
35 294 | data = new TwoIntsClass;
36 | ~~~~~
37 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:299:9: note: Assuming the condition is
 ↳ false
```

```

38 299 | if (globalReturnsTrueOrFalse())
39 | ~~~~~
40 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:299:5: note: Taking false branch
41 299 | if (globalReturnsTrueOrFalse())
42 | ~
43 C:\Users\z0041eeu\Desktop\Testumgebung\src\use_after_free\
 ↳ use_after_tests.cpp:312:1: note: Potential leak of memory
 ↳ pointed to by 'data'
44 312 | }
45 | ~

```

## B.11 Testcase TC11\_B

```

1 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↳ memory_leak_tests.cpp:31:9: warning: 'wcscpy' is deprecated:
 ↳ This function or variable may be unsafe. Consider using
 ↳ wcscpy_s instead. To disable deprecation, use
 ↳ _CRT_SECURE_NO_WARNINGS. See online help for details.
 ↳ [clang-diagnostic-deprecated-declarations]
2 31 | wcscpy(data, L"A String");
3 | ~
4 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\corecrt_wstring.h:119:1: note: 'wcscpy' has been
 ↳ explicitly marked deprecated here
5 119 | __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1(
6 | ~
7 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\corecrt.h:835:5: note: expanded from macro
 ↳ '__DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1'
8 835 |
 ↳ __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX(_ReturnType,
 ↳ _ReturnPolicy, _DeclSpec, _FuncName, _FuncName##_s,
 ↳ _DstType, _SalAttributeDst, _DstType, _Dst, _TType1, _TArg1)
9 | ~
10 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\corecrt.h:1894:17: note: expanded from macro
 ↳ '__DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX'
11 1894 | _CRT_INSECURE_DEPRECATED(_SecureFuncName)
 ↳ _DeclSpec _ReturnType __cdecl _FuncName(_SalAttributeDst
 ↳ _DstType *_Dst, _TType1 _TArg1);
12 | ~
13 C:\Program Files\Microsoft Visual Studio\2022\Professional\VC\
 ↳ Tools\MSVC\14.44.35207\include\vcruntime.h:368:55: note:
 ↳ expanded from macro '_CRT_INSECURE_DEPRECATED'
14 368 | #define _CRT_INSECURE_DEPRECATED(_Replacement)
 ↳ _CRT_DEPRECATED_TEXT(
15 |
16 C:\Program Files\Microsoft Visual Studio\2022\Professional\VC\
 ↳ Tools\MSVC\14.44.35207\include\vcruntime.h:358:47: note:

```

```

17 ↪ expanded from macro '_CRT_DEPRECATED_TEXT'
18 358 | #define _CRT_DEPRECATED_TEXT(_Text)
19 ↪ __declspec(deprecated(_Text))
20 |
21 ~
22 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
23 ↪ memory_leak_tests.cpp:54:9: warning: 'wcscpy' is deprecated:
24 ↪ This function or variable may be unsafe. Consider using
25 ↪ wcscpy_s instead. To disable deprecation, use
26 ↪ _CRT_SECURE_NO_WARNINGS. See online help for details.
27 ↪ [clang-diagnostic-deprecated-declarations]
28 54 | wcscpy(data, L"Safe String");
29 | ~
30 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
31 ↪ ucrt\corecrt_wstring.h:119:1: note: 'wcscpy' has been
32 ↪ explicitly marked deprecated here
33 119 | __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1(
34 | ~
35 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
36 ↪ ucrt\corecrt.h:835:5: note: expanded from macro
37 ↪ '__DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1'
38 835 |
39 ↪ __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX(_ReturnType,
40 ↪ _ReturnPolicy, _DeclSpec, _FuncName, _FuncName##_s,
41 ↪ _DstType, _SalAttributeDst, _DstType, _Dst, _TType1, _TArg1)
42 | ~
43 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
44 ↪ ucrt\corecrt.h:1894:17: note: expanded from macro
45 ↪ '__DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX'
46 1894 | _CRT_INSECURE_DEPRECATED(_SecureFuncName)
47 ↪ _DeclSpec _ReturnType __cdecl _FuncName(_SalAttributeDst
48 ↪ _DstType *_Dst, _TType1 _TArg1);
49 | ~
50 C:\Program Files\Microsoft Visual Studio\2022\Professional\
51 ↪ VC\Tools\MSVC\14.44.35207\include\vcruntime.h:368:55: note:
52 ↪ expanded from macro '_CRT_INSECURE_DEPRECATED'
53 368 | #define _CRT_INSECURE_DEPRECATED(_Replacement)
54 ↪ _CRT_DEPRECATED_TEXT(\
55 |
56 ~
57 C:\Program Files\Microsoft Visual Studio\2022\Professional\
58 ↪ VC\Tools\MSVC\14.44.35207\include\vcruntime.h:358:47: note:
59 ↪ expanded from macro '_CRT_DEPRECATED_TEXT'
60 358 | #define _CRT_DEPRECATED_TEXT(_Text)
61 ↪ __declspec(deprecated(_Text))
62 |
63 ~

```

## B.12 Testcase TC12\_B

```

1 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↪ memory_leak_tests.cpp:71:11: warning: Value stored to 'data'

```

```

 ↪ during its initialization is never read
 ↪ [clang-analyzer-deadcode.DeadStores]
2 71 | char* data = dataMap[2];
3 | ~~~~ ~~~~~~
4 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↪ memory_leak_tests.cpp:71:11: note: Value stored to 'data'
 ↪ during its initialization is never read
5 71 | char* data = dataMap[2];
6 | ~~~~ ~~~~~~
7 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↪ memory_leak_tests.cpp:94:9: warning: Call to function 'strcpy'
 ↪ is insecure as it does not provide bounding of the memory
 ↪ buffer. Replace unbounded copy functions with analogous
 ↪ functions that support length arguments such as 'strncpy'.
 ↪ CWE-119 [clang-analyzer-security.insecureAPI.strcpy]
8 94 | strcpy(data, "A String");
9 | ~~~~~
10 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↪ memory_leak_tests.cpp:94:9: note: Call to function 'strcpy' is
 ↪ insecure as it does not provide bounding of the memory buffer.
 ↪ Replace unbounded copy functions with analogous functions that
 ↪ support length arguments such as 'strncpy'. CWE-119
11 94 | strcpy(data, "A String");
12 | ~~~~~
13 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↪ memory_leak_tests.cpp:94:9: warning: 'strcpy' is deprecated:
 ↪ This function or variable may be unsafe. Consider using
 ↪ strcpy_s instead. To disable deprecation, use
 ↪ _CRT_SECURE_NO_WARNINGS. See online help for details.
 ↪ [clang-diagnostic-deprecated-declarations]
14 94 | strcpy(data, "A String");
15 | ~~~~~
16 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↪ ucrt\string.h:130:1: note: 'strcpy' has been explicitly marked
 ↪ deprecated here
17 130 | __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1(
18 | ~
19 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↪ ucrt\corecrt.h:835:5: note: expanded from macro
 ↪ ' __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1 '
20 835 |
 ↪ __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX(_ReturnType,
 ↪ _ReturnPolicy, _DeclSpec, _FuncName, _FuncName##_s,
 ↪ _DstType, _SalAttributeDst, _DstType, _Dst, _TType1, _TArg1)
21 | ~
22 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↪ ucrt\corecrt.h:1894:17: note: expanded from macro
 ↪ ' __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX '
23 1894 | _CRT_INSECURE_DEPRECATED(_SecureFuncName)
 ↪ _DeclSpec _ReturnType __cdecl _FuncName(_SalAttributeDst
 ↪ _DstType *_Dst, _TType1 _TArg1);

```

```

24 |
25 C:\Program Files\Microsoft Visual Studio\2022\Professional\
 ↳ VC\Tools\MSVC\14.44.35207\include\vcruntime.h:368:55: note:
 ↳ expanded from macro '_CRT_INSECURE_DEPRECATED'
26 368 | #define _CRT_INSECURE_DEPRECATED(_Replacement)
 ↳ _CRT_DEPRECATED_TEXT(
27 |
28 C:\Program Files\Microsoft Visual Studio\2022\Professional\
 ↳ VC\Tools\MSVC\14.44.35207\include\vcruntime.h:358:47: note:
 ↳ expanded from macro '_CRT_DEPRECATED_TEXT'
29 358 | #define _CRT_DEPRECATED_TEXT(_Text)
 ↳ __declspec(deprecated(_Text))
30 |
31 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↳ memory_leak_tests.cpp:120:9: warning: Call to function
 ↳ 'strcpy' is insecure as it does not provide bounding of the
 ↳ memory buffer. Replace unbounded copy functions with analogous
 ↳ functions that support length arguments such as 'strncpy'.
 ↳ CWE-119 [clang-analyzer-security.insecureAPI.strcpy]
32 120 | strcpy(data, "A String");
 ↳ ~~~~~
33 |
34 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↳ memory_leak_tests.cpp:120:9: note: Call to function 'strcpy'
 ↳ is insecure as it does not provide bounding of the memory
 ↳ buffer. Replace unbounded copy functions with analogous
 ↳ functions that support length arguments such as 'strncpy'.
 ↳ CWE-119
35 120 | strcpy(data, "A String");
 ↳ ~~~~~
36 |
37 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak
 ↳ \memory_leak_tests.cpp:120:9: warning: 'strcpy' is deprecated:
 ↳ This function or variable may be unsafe. Consider using
 ↳ strcpy_s instead. To disable deprecation, use
 ↳ _CRT_SECURE_NO_WARNINGS. See online help for details.
 ↳ [clang-diagnostic-deprecated-declarations]
38 120 | strcpy(data, "A String");
 ↳ ~
39 |
40 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\string.h:130:1: note: 'strcpy' has been explicitly marked
 ↳ deprecated here
41 130 | __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1(
42 | ~
43 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\corecrt.h:835:5: note: expanded from macro
 ↳ '__DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1'
44 835 |
 ↳ __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX(_ReturnType,
 ↳ _ReturnPolicy, _DeclSpec, _FuncName, _FuncName##_s,
 ↳ _DstType, _SalAttributeDst, _DstType, _Dst, _TType1, _TArg1)
45 | ~

```

```

46 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\corecrt.h:1894:17: note: expanded from macro
 ↳ ' _CRT_INSECURE_DEPRECATED(_SecureFuncName)
47 1894 | _CRT_INSECURE_DEPRECATED(_SecureFuncName)
 ↳ _DeclSpec _ReturnType __cdecl _FuncName(_SalAttributeDst
 ↳ _DstType *_Dst, _TType1 _TArg1);
48 | ^
49 C:\Program Files\Microsoft Visual Studio\2022\Professional\
 ↳ VC\Tools\MSVC\14.44.35207\include\vcruntime.h:368:55: note:
 ↳ expanded from macro ' _CRT_INSECURE_DEPRECATED '
50 368 | #define _CRT_INSECURE_DEPRECATED(_Replacement)
 ↳ _CRT_DEPRECATED_TEXT(\
51 | ^
52 C:\Program Files\Microsoft Visual Studio\2022\Professional\
 ↳ VC\Tools\MSVC\14.44.35207\include\vcruntime.h:358:47: note:
 ↳ expanded from macro ' _CRT_DEPRECATED_TEXT '
53 358 | #define _CRT_DEPRECATED_TEXT(_Text)
 ↳ __declspec(deprecated(_Text))

```

## B.13 Testcase TC13\_B

```

1 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↳ memory_leak_tests.cpp:155:5: warning: Call to function
 ↳ 'strcpy' is insecure as it does not provide bounding of
 ↳ the memory buffer. Replace unbounded copy functions with
 ↳ analogous functions that support length arguments such
 ↳ as 'strncpy'. CWE-119
 ↳ [clang-analyzer-security.insecureAPI.strcpy]
2 155 | strcpy(data, "A String");
3 | ~~~~~
4 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↳ memory_leak_tests.cpp:155:5: note: Call to function 'strcpy'
 ↳ is insecure as it does not provide bounding of the memory
 ↳ buffer. Replace unbounded copy functions with analogous
 ↳ functions that support length arguments such as 'strncpy'.
 ↳ CWE-119
5 155 | strcpy(data, "A String");
6 | ~~~~~
7 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↳ memory_leak_tests.cpp:155:5: warning: 'strcpy' is deprecated:
 ↳ This function or variable may be unsafe. Consider using
 ↳ strcpy_s instead. To disable deprecation, use
 ↳ _CRT_SECURE_NO_WARNINGS. See online help for details.
 ↳ [clang-diagnostic-deprecated-declarations]
8 155 | strcpy(data, "A String");
9 | ~
10 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\string.h:130:1: note: 'strcpy' has been explicitly marked
 ↳ deprecated here
11 130 | __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1(

```



```

12 | ~
13 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\corecrt.h:835:5: note: expanded from macro
 ↳ '__DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1'
14 835 |
 ↳ __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX(_ReturnType,
 ↳ _ReturnPolicy, _DeclSpec, _FuncName, _FuncName##_s,
 ↳ _DstType, _SalAttributeDst, _DstType, _Dst, _TType1, _TArg1)
15 | ~
16 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\corecrt.h:1894:17: note: expanded from macro
 ↳ '__DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX'
17 1894 | _CRT_INSECURE_DEPRECATED(_SecureFuncName)
 ↳ _DeclSpec _ReturnType __cdecl _FuncName(_SalAttributeDst
 ↳ _DstType *_Dst, _TType1 _TArg1);
18 | ~
19 C:\Program Files\Microsoft Visual Studio\2022\Professional\
 ↳ VC\Tools\MSVC\14.44.35207\include\vcruntime.h:368:55: note:
 ↳ expanded from macro '_CRT_INSECURE_DEPRECATED'
20 368 | #define _CRT_INSECURE_DEPRECATED(_Replacement)
 ↳ _CRT_DEPRECATED_TEXT(\
21 | ~
22 C:\Program Files\Microsoft Visual Studio\2022\Professional\
 ↳ VC\Tools\MSVC\14.44.35207\include\vcruntime.h:358:47: note:
 ↳ expanded from macro '_CRT_DEPRECATED_TEXT'
23 358 | #define _CRT_DEPRECATED_TEXT(_Text)
 ↳ __declspec(deprecated(_Text))
24 | ~
25 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↳ memory_leak_tests.cpp:187:5: warning: Call to function
 ↳ 'strcpy' is insecure as it does not provide bounding of the
 ↳ memory buffer. Replace unbounded copy functions with analogous
 ↳ functions that support length arguments such as 'strncpy'.
 ↳ CWE-119 [clang-analyzer-security.insecureAPI.strcpy]
26 187 | strcpy(data, "A String");
27 | ~~~~~
28 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↳ memory_leak_tests.cpp:187:5: note: Call to function 'strcpy'
 ↳ is insecure as it does not provide bounding of the memory
 ↳ buffer. Replace unbounded copy functions with analogous
 ↳ functions that support length arguments such as 'strncpy'.
 ↳ CWE-119
29 187 | strcpy(data, "A String");
30 | ~~~~~
31 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↳ memory_leak_tests.cpp:187:5: warning: 'strcpy' is deprecated:
 ↳ This function or variable may be unsafe. Consider using
 ↳ strcpy_s instead. To disable deprecation, use
 ↳ _CRT_SECURE_NO_WARNINGS. See online help for details.
 ↳ [clang-diagnostic-deprecated-declarations]
32 187 | strcpy(data, "A String");

```



```

33 | ~
34 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\string.h:130:1: note: 'strcpy' has been explicitly marked
 ↳ deprecated here
35 130 | __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1(
36 | ~
37 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\corecrt.h:835:5: note: expanded from macro
 ↳ '__DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1'
38 835 |
 ↳ __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX(_ReturnType,
 ↳ _ReturnPolicy, _DeclSpec, _FuncName, _FuncName##_s,
 ↳ _DstType, _SalAttributeDst, _DstType, _Dst, _TType1, _TArg1)
39 | ~
40 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\corecrt.h:1894:17: note: expanded from macro
 ↳ '__DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX'
41 1894 | _CRT_INSECURE_DEPRECATED(_SecureFuncName)
 ↳ _DeclSpec _ReturnType __cdecl _FuncName(_SalAttributeDst
 ↳ _DstType *_Dst, _TType1 _TArg1);
42 | ~
43 C:\Program Files\Microsoft Visual Studio\2022\Professional\
 ↳ VC\Tools\MSVC\14.44.35207\include\vcruntime.h:368:55: note:
 ↳ expanded from macro '_CRT_INSECURE_DEPRECATED'
44 368 | #define _CRT_INSECURE_DEPRECATED(_Replacement)
 ↳ _CRT_DEPRECATED_TEXT(\
45 | ~
46 C:\Program Files\Microsoft Visual Studio\2022\Professional\
 ↳ VC\Tools\MSVC\14.44.35207\include\vcruntime.h:358:47: note:
 ↳ expanded from macro '_CRT_DEPRECATED_TEXT'
47 358 | #define _CRT_DEPRECATED_TEXT(_Text)
 ↳ __declspec(deprecated(_Text))
48 | ~
49 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↳ memory_leak_tests.cpp:210:5: warning: Call to function
 ↳ 'strcpy' is insecure as it does not provide bounding of the
 ↳ memory buffer. Replace unbounded copy functions with analogous
 ↳ functions that support length arguments such as 'strncpy'.
 ↳ CWE-119 [clang-analyzer-security.insecureAPI.strcpy]
50 210 | strcpy(data, "A String");
51 | ~~~~~
52 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↳ memory_leak_tests.cpp:210:5: note: Call to function 'strcpy'
 ↳ is insecure as it does not provide bounding of the memory
 ↳ buffer. Replace unbounded copy functions with analogous
 ↳ functions that support length arguments such as 'strncpy'.
 ↳ CWE-119
53 210 | strcpy(data, "A String");
54 | ~~~~~
55 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↳ memory_leak_tests.cpp:210:5: warning: 'strcpy' is deprecated:

```

```

 ↳ This function or variable may be unsafe. Consider using
 ↳ strcpy_s instead. To disable deprecation, use
 ↳ _CRT_SECURE_NO_WARNINGS. See online help for details.
 ↳ [clang-diagnostic-deprecated-declarations]
56 210 | strcpy(data, "A String");
57 | ^
58 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\string.h:130:1: note: 'strcpy' has been explicitly marked
 ↳ deprecated here
59 130 | __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1(
60 | ^
61 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\corecrt.h:835:5: note: expanded from macro
 ↳ '__DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1'
62 835 |
 ↳ __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX(_ReturnType,
 ↳ _ReturnPolicy, _DeclSpec, _FuncName, _FuncName##_s,
 ↳ _DstType, _SalAttributeDst, _DstType, _Dst, _TType1, _TArg1)
63 | ^
64 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↳ ucrt\corecrt.h:1894:17: note: expanded from macro
 ↳ '__DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX'
65 1894 | _CRT_INSECURE_DEPRECATED(_SecureFuncName)
 ↳ _DeclSpec _ReturnType __cdecl _FuncName(_SalAttributeDst
 ↳ _DstType *_Dst, _TType1 _TArg1);
66 | ^
67 C:\Program Files\Microsoft Visual Studio\2022\Professional\VC\
 ↳ Tools\MSVC\14.44.35207\include\vcruntime.h:368:55: note:
 ↳ expanded from macro '_CRT_INSECURE_DEPRECATED'
68 368 | #define _CRT_INSECURE_DEPRECATED(_Replacement)
 ↳ _CRT_DEPRECATED_TEXT(
69 |
70 C:\Program Files\Microsoft Visual Studio\2022\Professional\VC\
 ↳ Tools\MSVC\14.44.35207\include\vcruntime.h:358:47: note:
 ↳ expanded from macro '_CRT_DEPRECATED_TEXT'
71 358 | #define _CRT_DEPRECATED_TEXT(_Text)
 ↳ __declspec(deprecated(_Text))
72 |
73
74 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↳ memory_leak_tests.cpp:234:5: warning: Call to function
 ↳ 'strcpy' is insecure as it does not provide bounding of the
 ↳ memory buffer. Replace unbounded copy functions with analogous
 ↳ functions that support length arguments such as 'strncpy'.
 ↳ CWE-119 [clang-analyzer-security.insecureAPI strcpy]
75 234 | strcpy(data, "A String");
76 | ~~~~~
77 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↳ memory_leak_tests.cpp:234:5: note: Call to function 'strcpy'
 ↳ is insecure as it does not provide bounding of the memory
 ↳ buffer. Replace unbounded copy functions with analogous

```

```

 ↪ functions that support length arguments such as 'strncpy'.
 ↪ CWE-119
78 234 | strcpy(data, "A String");
79 | ~~~~~
80 C:\Users\z0041eeu\Desktop\Testumgebung\src\memory_leak\
 ↪ memory_leak_tests.cpp:234:5: warning: 'strcpy' is deprecated:
 ↪ This function or variable may be unsafe. Consider using
 ↪ strcpy_s instead. To disable deprecation, use
 ↪ _CRT_SECURE_NO_WARNINGS. See online help for details.
 ↪ [clang-diagnostic-deprecated-declarations]
81 234 | strcpy(data, "A String");
82 | ^
83 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↪ ucrt\string.h:130:1: note: 'strcpy' has been explicitly marked
 ↪ deprecated here
84 130 | __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1(
85 | ^
86 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↪ ucrt\corecrt.h:835:5: note: expanded from macro
 ↪ '__DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1'
87 835 |
 ↪ __DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX(_ReturnType,
 ↪ _ReturnPolicy, _DeclSpec, _FuncName, _FuncName##_s,
 ↪ _DstType, _SalAttributeDst, _DstType, _Dst, _TType1, _TArg1)
88 | ^
89 C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\
 ↪ ucrt\corecrt.h:1894:17: note: expanded from macro
 ↪ '__DEFINE_CPP_OVERLOAD_STANDARD_FUNC_0_1_EX'
90 1894 | _CRT_INSECURE_DEPRECATED(_SecureFuncName)
 ↪ _DeclSpec _ReturnType __cdecl _FuncName(_SalAttributeDst
 ↪ _DstType *_Dst, _TType1 _TArg1);
91 | ^
92 C:\Program Files\Microsoft Visual Studio\2022\Professional\
 ↪ VC\Tools\MSVC\14.44.35207\include\vcruntime.h:368:55: note:
 ↪ expanded from macro '_CRT_INSECURE_DEPRECATED'
93 368 | #define _CRT_INSECURE_DEPRECATED(_Replacement)
 ↪ _CRT_DEPRECATED_TEXT(
94 | ^
95 C:\Program Files\Microsoft Visual Studio\2022\Professional\
 ↪ VC\Tools\MSVC\14.44.35207\include\vcruntime.h:358:47: note:
 ↪ expanded from macro '_CRT_DEPRECATED_TEXT'
96 358 | #define _CRT_DEPRECATED_TEXT(_Text)
 ↪ __declspec(deprecated(_Text))

```

## B.14 Testcase TC14\_B

```
1 KEINE ERKENNUNG SEITENS CLANG --> LEER
```

## B.15 Testcase TC15\_B

1 KEINE ERKENNUNG SEITENS CLANG --> LEER
------------------------------------------