MASTER THESIS

DE HAAGSE HOGESCHOOL PRO

TNO

# Q&A Insecure Code Detection

*University Supervisors:*
First: Daniël Meinsma
Second: Jos Griffioen

*TNO Supervisors:*
First: Jelle Nauta
Second: Thomas Rooijakkers

*Author:*
Richard Holleman

February 10, 2025 - Version V2.0

## Acknowledgement

Two years ago, I started the Master's program in Cyber Security Engineering at The Hague University of Applied Sciences Pro. I did this alongside my full-time job as a computer science teacher. Every Friday, I attended classes at The Hague, where I was surrounded by a diverse group of professionals from various fields who were all doing this master in addition to a full-time job. I found this experience very enjoyable and educational. However, pursuing a part-time master's degree while working full-time, raising a family, and maintaining hobbies was, at times, intensive. I knew this in advance, and that's why I didn't start when my children were younger. In April 2024, I began the graduation process and had various ideas for topics. I first had to decide whether or not to graduate with an external company. My employer, the HAN University of Applied Sciences, does not have expertise in cybersecurity at master's level, and I saw the added value of graduating with an external company. One of my lecturers during the master's program worked for TNO, which gave me the idea to explore the possibility of graduating with TNO. I then communicated my ideas to TNO, and one of the ideas matched their current work. The idea had the following keywords: vulnerability warning with AI / programming buddy/coach for programmers regarding cybersecurity / AI advise secure software development. This idea ultimately resulted in the research presented in this thesis. I would like to thank TNO, and in particular Jelle Nauta as my first supervisor and Thomas Rooijakkers as my second supervisor, for the time and energy they invested in this. I would also like to thank my supervisors Daniel Meinsma and Jos Griffioen, who guided me from the university. During my graduation, I conducted an interview with Erik van der Kouwe, a researcher and lecturer at the VU Amsterdam; I would also like to thank him for his time. Finally, my thanks go to Monique Hendriks, my partner, for her support and for proofreading my thesis.

# Contents

# List of Figures

# List of Tables

## Executive Summary

This master's thesis explores the application of Large Language Models (LLMs), such as GEmini, GPT-4, and Claude, for detecting vulnerabilities in C++ code snippets, particularly on online platforms like StackOverflow.com. The goal was to develop a browser extension that can alert students and programmers to unsafe code and provide them with relevant information about identified vulnerabilities.

This research is motivated by the observation that students and programmers often utilize online resources, such as StackOverflow.com, to find programming solutions. However, these sources can also contain unsafe code snippets, which can lead to the unintentional learning and use of insecure coding practices. The emergence of LLMs offers new possibilities to address this problem through automated detection and explanation of vulnerabilities.

The central research question of this thesis is: "How can a web browser for students be enhanced so that unsafe code snippets on Q&A websites are automatically enriched with relevant information about the identified vulnerability using generative AI?" To answer this question, sub-questions are posed regarding the requirements for such a tool, the relevant information about unsafe code, the identification of code snippets, the evaluation method using generative AI, and the presentation of the relevant information to the user.

The Design Science Research (DSR) methodology was employed to design, develop, and evaluate an artifact (the browser extension). This included a literature review on secure programming practices, existing SAST tools, the role of AI in vulnerability detection, and Chrome extensions. Subsequently, quantitative experiments were conducted to measure the effectiveness of different LLMs in detecting vulnerabilities, and qualitative analysis was performed to examine the practical implications.

The results demonstrate that LLMs, particularly GPT-4 in combination with strategy 1, show promise in detecting vulnerabilities in C++ code snippets. However, the effectiveness varies depending on the specific model and the prompts used. The developed browser extension offers functionality for both evaluating LLMs and supporting users in identifying unsafe code. It provides visual feedback on the safety of code snippets on StackOverflow.com and displays detailed information about detected vulnerabilities.

In conclusion, LLMs have the potential to automate and improve the detection of vulnerabilities in code, contributing to safer coding practices. While challenges remain, such as refining prompt strategies and ensuring the accuracy of results, LLMs offer a promising direction for enhancing software security and supporting developers in writing secure code. This thesis contributes to the understanding of the capabilities, limitations, and practical implications of LLMs in the context of software security.

# Chapter 1

# Introduction

## 1.1 Background and Problem Statement

It is well known that computer science students often solve their programming challenges by googling the answer. They then end up on Q&A sites such as StackOverflow and Reddit (Pöial [2021]). These are websites where usually a question about the problem has already been asked and answered by people from the community. If the question has not been asked and answered before, students often ask the question themselves, and it is usually answered within a few hours. Students also obtain code examples from repositories such as GitHub and SourceForge. What all these sources have in common is that the website owner is not responsible for the quality of the code, and therefore, unsafe code examples and snippets are regularly found among them (Bai et al. [2019]). This problem is most prevalent on Q&A sites but certainly also applies to repositories.

Since the end of 2022, generative AIs based on Large Language Models (LLMs), such as ChatGPT, have emerged as a new source for students (Haleem et al. [2022]). Students can present their problem to the chatbot in human-based instructions (prompts), and it will often come up with a working solution. Research has since shown that LLMs generate both safe and unsafe code (Majdinasab et al. [2024]). One reason for this could be that LLMs are trained on code sourced from repositories like GitHub, which may contain unsafe code.

The use of unsafe code snippets by students is a problem because they are taught unsafe code constructs, and later in their careers, they will produce unsafe code that is subsequently put into production. In addition, research shows that even among graduate programmers, writing secure code is an ongoing challenge, and they do not always check code examples that work functionally for unsafe code (Acar et al. [2017]).

Since it became known that LLMs can generate unsafe code, several studies have been conducted with the aim of solving this problem. For example, it has been investigated and shown that through model fine-tuning, the amount of unsafe code can be reduced (He et al. [2024]). Also, AI providers such as GitHub are actively working on improving their models so that no, or at least less, unsafe code is generated (Zhao [2023]).

The fact that there are now LLMs and that they are constantly improving does not mean that students no longer use the other sources. In fact, students often find it difficult to ask the right question to a chatbot and therefore still like to search on Q&A sites or in repositories. In addition, an LLM cannot yet solve every programming problem, especially if you do not know what the problem is. It is therefore likely that these sources will continue to be used in the years to come.

## 1.2 Research Objective

The objective of this research was to find a solution to the problem that students are taught unsafe code constructs because they obtain code snippets from websites such as StackOverflow. It was investigated how this objective can best be achieved using generative AI, without modifying or fine-tuning such a model. The result is a proof-of-concept that can be used by students but also allows for the testing and evaluation of different LLMs. The artefact had to be able to identify C++ code snippets on stackoverflow and assess whether the code is safe or not using generative AI. If it is not safe, relevant information should be displayed. This could include, for example, a safe alternative and a Common Weakness Enumeration (CWE) or Common vulnerability Enumeration (CVE) reference. Regarding the evaluation of different LLMs, a reliable and reproducible test setup had to be developed.

The tool should be usable in education from the start of the propaedeutic phase and should ensure that students learn early on which code snippets on websites such as StackOverflow are safe or unsafe. In addition, it could also help experienced programmers avoid using unsafe snippets from such sources. By making the artefact publicly available, it can contribute to a generally better understanding of secure code and ultimately safer software.

In summary, the research has two objectives: 1) To develop an artefact that can be used by students, 2) To have a test setup to determine the capacity of LLMs regarding vulnerability detection.

Within the scope of the research, there were two important hypotheses that needed to be proven. 1) An LLM is capable of analyzing code and can provide suggestions for improvement, 2) The advantage of using an LLM over traditional methods is that little technology is needed beyond the LLM Application Programming Interface (API). The latter would be a significant advantage over traditional tools and an important motivation for using the developed artefact.

## 1.3 Main Research Question and Sub-questions

The research is guided by the following main research question and sub-questions: “How can a web browser for students be enhanced so that unsafe code snippets on Q&A websites are automatically enriched with relevant information about the identified vulnerability using generative AI?

1. What requirements must such a tool meet to be usable for students?
2. What information regarding unsafe code is relevant?
3. How can a code snippet be identified on a Q&A website?
4. What method can best be used to automatically evaluate the code examples by a generative AI?
5. How can the relevant information be displayed to the student?

During the preliminary research phase, the scope was narrowed to the following, as further described in the Software Requirement Specification (SRS):

1. This research focuses exclusively on the C/C++ programming language; other languages are outside its scope.
2. The scope of this research is limited to a set of 10 to 15 specific vulnerabilities. Other vulnerabilities were not considered.
3. The artefact was developed solely for the Q&A site: StackOverflow.

Based on the main research question and its associated sub-questions, the initial objective of this research also included an investigation into the didactic aspects of the subject. This was primarily intended to be explored as part of sub-questions 1 and 2. However, due to time constraints and a focus on primarily technical feasibility, the didactic aspects received limited attention. Only the section 'literature review' (chapter 3.2) describes the relationship between education and secure programming, and the SRS dedicates limited attention to it.

Furthermore, it is worth noting that a considerable amount of comparable research was conducted during the execution of this study. The section 'contribution' (chapter 2), outlines the added value of this study, and the section 'literature review' (chapter 3.6) discusses the related work.

To answer the main question and realize the artefact, Design Science Research (DSR) was used. DSR is fitting for this research because it focuses on developing and evaluating a practical artefact, in this case a browser extension, to address the real-world problem of vulnerability detection in code snippets. It provides a solid roadmap to design, build, and test the artefact and the use of LLM's for vulnerability detection.

## 1.4 Thesis Structure

The structure of this thesis, based on Design Science Research (DSR) principles, is divided into seven chapters. Each chapter contributes to the development, implementation, and evaluation of the artefact, culminating in a comprehensive discussion and conclusion. The structure is outlined as follows:

**Chapter 1: Introduction** - This chapter provides the foundation for the research, introducing the problem statement, research question, and sub-questions. It outlines the motivation and relevance of the study. Furthermore, it briefly describes the chosen methodology, Design Science Research, and provides an overview of the thesis structure.

**chapter 2: Contribution** - This chapter describes, within the context of comparable research conducted concurrently, the differences and similarities between these studies and the added value and unique aspects of this research.

**Chapter 3: Literature Review** - This chapter presents a comprehensive review of existing literature relevant to the research topic. It explores existing solutions, theoretical frameworks, and empirical studies related to static code analysis, browser extensions and vulnerability detection using AI, etc. The review synthesizes current knowledge, identifies limitations in existing approaches, and positions this research within the broader academic landscape.

**Chapter 4: Research Methodology** - This chapter details the Design Science Research (DSR) methodology adopted for this study. It elaborates on the specific DSR steps followed. The rationale for choosing DSR is presented, and the specific methods and techniques employed within each phase are described in detail.

**Chapter 5: Design and Development** - This chapter documents the design and development process of the artefact, which in this research is a browser extension for vulnerability detection. This chapter describes the requirements that the artefact must meet and what was ultimately realized. Additionally, it elaborates on the architecture and functionality. The functionality is divided into: 1) the functionality during final production, and 2) the functionality during the execution of the experiment.

**Chapter 6: Evaluation** - This chapter outlines the evaluation of the developed artefact and the chosen LLM's. It describes the evaluation methods used, including the data collected, results of the evaluation and the analysis performed. The evaluation assesses the effectiveness and usability of different LLM's and the artefact in general.

**Chapter 7: Conclusion and Discussion** - This chapter presents the concluding remarks of the research and the findings from the evaluation. It summarizes the key findings, discusses the implications of the research, and reflects on the limitations of the study. It also outlines potential directions for future research, building upon the insights gained from this study.

Chaper 8: Reflection - This chapter is a personal reflection on the past graduation period.

# Chapter 2

# Contribution

The use of AI, specifically Large Language Models (LLMs), regarding vulnerability detection is a growing field of research, as made clear in Section 6.3. Notably, all 15 papers discussed in that section were published between May 2024 and November 2024, underscoring the relevance of this research area and, consequently, of the research presented in this thesis. A comparative analysis of the research design of this study with related works reveals numerous similarities, but also significant differences. For instance, most studies, like this thesis, indicate the use of prompt engineering in some form and conclude that how prompts are formulated influences the quality of the response. Several studies report that the 'Chain-of-Thought' (CoT) approach has a positive effect (Sheng et al. [2024a], Mao et al. [2024] ), while other studies indicate the use of certain strategies but do not compare or evaluate these strategies (Li et al. [2024a], Jiang et al. [2024]). This thesis, in contrast, confirms that the 'Chain-of-Thought' (CoT) approach can have a positive effect, but also demonstrates a negative effect based on the relationship between prompt complexity and a simplified assigned output. The precise nature of this relationship needs further investigation.

Furthermore, the studies vary in terms of programming language, dataset, the use of fine-tuning, and integration into practical applications. This research distinguishes itself by focusing on the integration of LLM-based vulnerability detection within a web browser, specifically targeting the identification of insecure code snippets on Q&A websites such as StackOverflow, thereby addressing the issue as close to the source as possible. Other studies primarily focus on experimental evaluations without addressing potential integration. An exception is the work of Ziyang Li (Li et al. [2024a]), which integrates with the Static Code Analysis Tool 'CodeQL'. Another unique aspect of this research, compared to the 15 similar studies, is the specific combination of programming language, LLM, and dataset. While some studies address multiple programming languages, and others focus solely on one, this research, along with four others, concentrates on C++. One of these also utilizes the NIST SARD Juliet 1.3 dataset (Pelofske et al. [2024]).

However, a key difference, and a significant contribution of this research, is the scale of evaluation. While the study conducted by Elijah Pelofske analyzes only 36 test cases, this research examines a substantially larger dataset comprising 1000 distinct test cases, thereby providing a more robust and generalizable assessment.

Another study conducted by Shaznin Sultana (Sultana et al. [2024]) also focuses on C/C++ but employs a fine-tuned LLM, unlike this research, which utilizes a purely pre-trained model. Moreover, the number of test cases used in the study of Shaznin Sultana remains unclear. While it specifies the number of samples used for fine-tuning, it does not specify the number used for final evaluation. In summary, the research of this thesis contributes to the field through its unique combination of elements, its novel integration approach, and the fact that the experiment was conducted with 1000 different samples.

Naturally, these differences in design and execution lead to distinct and unique results compared to other studies, making this a valuable extension to the existing Design Knowledge (DK). Furthermore, it is not only the differences that contribute value. The similarities also serve to corroborate the conclusions drawn from the various studies. The majority of those studies in this field, through various methodologies, underscore the significance of prompt engineering and confirm the potential of LLMs in vulnerability detection. For instance, study conducted by Shaznin Sultana (Sultana et al. [2024]) reports a True Positive (TP) rate (recall) of 87% when employing GPT-4, and the study conducted by Wenpin Hou (Hou and Ji [2024]) demonstrates that their optimal prompt strategy, used in conjunction with GPT-4, outperformed 85% of human participants. Although this research obtained a slightly lower detection rate of approximately 70%, it remains largely consistent with these findings. Crucially, it reinforces the consensus that while LLMs offer a valuable tool for vulnerability detection, they are not without limitations. This study, therefore, contributes to the growing body of Design Knowledge (DK) by confirming the potential, yet acknowledging the need for continued improvement, of LLMs in this area.

# Chapter 3

# Literature Review

## 3.1 Secure programming

The concept of 'secure programming' is known by various names, including 'secure coding,' 'defensive programming,' and 'secure software development.' Within the context of software development, it is essential for the product to function correctly and reliably. The term 'secure' implies that no security issues should be introduced during the development process and is also known as 'software construction security'. It is noteworthy that this term may carry different meanings for different individuals; it can refer to how code constructs are programmed to ensure their inherent security or to the incorporation of security measures into software (Washizaki [2024]). This research primarily addresses the former.

Existing research recognizes the critical role played by secure programming on overall cyber security (Potluri et al. [2023] , Fulton et al. [2021]). Cyber threats can arise from various sources, including software defects, human errors, or weaknesses in organisational processes. Among these, software vulnerabilities remain one of the most prevalent attack vectors (Grover et al. [2016] , Dozono et al. [2024a]). For decades, this issue has primarily been addressed by patching software vulnerabilities post-facto. While this approach was suitable in the early days of the digital revolution, it became increasingly inadequate as connectivity grew with the advent of the internet and, ultimately, the Internet of Things (IoT) (Grover et al. [2016]). Historically, numerous significant cybersecurity incidents have been caused by software vulnerabilities, with notable examples including:

1. Morris Worm (1988): One of the first viruses that exploited buffer overflow vulnerabilities to propagate.

2. Heartbleed (2014): A bug in OpenSSL that exposed sensitive data due to improper memory handling.

3. Equifax Breach (2017): One of the largest data breaches in history, caused by a vulnerability in the Apache Struts framework.

4. Log4Shell (2021): This vulnerability allowed attackers to execute arbitrary code on affected servers by exploiting unsafe input processing.

For a long time, software development teams and individual programmers were perceived as solely responsible for these issues. However, previous studies have shown that businesses, including management and executive teams, often did not emphasize the importance of secure software development and provided insufficient support (Assal and Chiasson [2019] and Tuladhar et al. [2021]). Due to the numerous incidents that have occurred with increasing frequency in recent years, businesses have begun to recognize the importance, a sentiment that is similarly reflected among software developers. A common approach employed by developers in problem-solving involves seeking solutions online, particularly through Q&A sites. This practice also applies to issues related to secure programming. However, earlier research has demonstrated that these sources frequently contain insecure code constructs, thereby not necessarily contributing to safer coding practices (Acar et al. [2017], Lam et al. [2022]).

Given that secure programming has long been a low priority in the professional field, educational institutions have similarly neglected it. This oversight is because educational programs, particularly in higher education (HBO), tailor their curricula to meet the needs of the companies where students will work after their studies. Factors influencing better knowledge of 'secure programming' have been explored in various studies, demonstrating that introducing education on 'secure programming' at an early stage is a significant factor. Insufficient education, alongside negligence, is identified as a significant underlying cause of many cyber-attacks (Buckley et al. [2018]). Prior research also indicates a shift is occurring, with secure programming becoming an increasingly essential competency taught to students across various levels: beginner, intermediate, and expert (Lam et al. [2022]).

While there have been papers and books published regarding secure programming in the past, it has never received as much attention as in recent years. This is evident in the incorporation of 'Secure' programming into the 'Software Development Life Cycle' (SDLC), resulting in the updated concept named 'Secure Software Development Life Cycle' (SSDLC). This addition is recommended in academic literature (Richardson and Thies [2013]) and is also endorsed and implemented by companies, including the reputable cybersecurity firm Snyk (Tal [2024]). Furthermore, there is a growing number of standards addressing secure programming. For instance, the National Institute of Standards and Technology (NIST), a scientific body under the U.S. federal government, published the Secure Software Development Framework (SSDF) in 2022 (Nist [2024b]). In 2024, the subject is also added to the new version of the Software Engineering Body of Knowledge (SWEBOK), published by the IEEE Computer Society (Washizaki [2024]). The implementation of secure programming is evident in the development of programming languages. In recent years, languages have been developed that are designated as ’more secure by design’ than traditional languages such as C/C++. An example of this is Rust, which has ’memory safety’ implemented as a key feature (Fulton et al. [2021]).

## 3.2 Secure Programming in Higher Education

Many studies have highlighted the necessity of secure programming, yet it is also widely reported that recent software engineering graduates, and even lecturers, often lack adequate knowledge on the subject (Mdunyelwa et al. [2021]). Further studies emphasize that secure programming should be a core component of the curriculum (Torbunova et al. [2024]).

Challenges Faced by Students regarding the level of knowledge among students, recent research shows that many students lack the fundamental skills necessary to write secure programs . Even when students have prior knowledge, they often fail to consider software security while programming (Lam et al. [2022]). Research conducted by Lam et al. reveals that one common issue is that students introduce buffer overflows into their code because they are unaware of whether the C programming language performs bounds checking. The same study also indicates that many students lack knowledge about memory management and that their understanding of how pointers and memory are related is insufficiently developed.

Other concerning findings include students frequently ignoring compiler warnings (Lam et al. [2022]) and often copying answers directly from online forums, such as StackOverflow, without critically evaluating or understanding them. This study showed that a portion of students used unsafe or inappropriate code snippets in their projects in this manner. This behavior underscores the need to teach students how to responsibly use online resources while adhering to security principles.

In the general framework of didactics, taxonomies such as ‘Bloom’s Taxonomy’ and the ‘SOLO Taxonomy’ help describe student learning, guide curriculum development, and identify learning progression. While these are not specifically designed for teaching secure programming, they are still applicable. Majed Almansoori et al. (Almansoori et al. [2023]) have developed an extension of the ‘SOLO Taxonomy’ with a focus on developing teaching materials and assessing learning outcomes in secure programming. Research has also explored where and how secure programming can be implemented in education. The following approaches have been identified: 1) the single-course approach; 2) the track approach; and 3) the thread approach (Chung et al. [2014]). The difference between the 'track' and 'thread' approach is that the 'track' implements standalone new courses, while the 'thread' integrates it into existing courses. The latter is considered the most effective didactically and places the least burden on an already full curriculum. Another similar classification used in studies is: 1) concentrated courses; 2) diffusion through curricula; 3) concentration/degree programs (Yuan et al. [2016]).

Numerous studies have attempted to describe or develop supplementary resources that educators can use to create secure programming courses. These studies propose one or more of the following solutions: frameworks, clinics, interventions, exercises like CTF (Capture the Flag), courses, workshops, tools, e-learning, checklists, or patterns. For instance, Bisschop et al. (Bishop et al. [2019]) introduced a 'security clinic' to strengthen secure programming competencies. The strength of this clinic lies in the feedback students receive on their work, as feedback is one of the key didactic learning principles. Another approach with positive characteristics is the use of automated tools for teaching secure programming. An example is 'ESIDE' (Educational Security in the IDE), which resulted from Study 'Comparing Educational Approaches to Secure Programming : Tool vs. TA' (Sedova [2017]). The advantage of such a tool is that it minimally burdens lecturers and the curriculum, while providing students with immediate feedback on their mistakes. However, the downside, as shown by this research, is that students often do not thoroughly read the accompanying information and motivation provided.

## 3.3 Available SAST Tools

It should now be clear that starting to check for insecure code early in the SDLC is essential. White box and black box analyses form two fundamental approaches within software security testing to achieve this (Cruz et al. [2023]). White box analysis provides full access to the source code, allowing programmers and testers to conduct in-depth examinations of the code's structure, logic, and potential vulnerabilities. This method is particularly effective in identifying specific security risks at the code level, such as logical vulnerabilities or insecure programming constructs. In contrast, black box analysis works without access to the source code, focusing instead on the system's input and output. it's often combined with techniques such as fuzzing (Seacord [2013]), where random or semi-random data is fed into the software to test its robustness.

Code is often assessed through manual code review performed by fellow programmers. While manual review benefits from human intuition and expertise, as previously mentioned, programmers generally have limited knowledge in the area of software security, and the manual process inherently overlooks certain issues. Numerous studies argue that using automated code assessment tools is essential and necessary (Charoenwet et al. [2024]). Combining both approaches is often the most effective method, as it merges the thoroughness of human review with the speed and consistency of automated checks. For instance, A.-M. Stanciu et al., in their study, assert that manual review is an indispensable part, as automated tools cannot detect all vulnerabilities (Stanciu and Ciocârlie [2023]). Furthermore, these tools are also used in education, and research shows that they have a positive impact on the learning process of the students and that they themselves are positive about their use. (AlOmar et al. [2023]).

The most basic form of code analysis is performed by a compiler. For instance, a C++ compiler can issue warnings about "uninitialized variables," which, by itself, can result in insecure code (Guß [2020]). Although compilers can play a role in detecting certain issues, their capabilities for security testing are limited. Compilers are primarily designed to translate code into executable binaries, not to conduct extensive security analysis. Some modern compilers, however, have built-in features that can identify some security vulnerabilities, such as buffer overflows or uninitialized variables. Although useful, these features are generally insufficient for comprehensive security analysis and should be supplemented by dedicated tools for thorough assessment.

Across the entire spectrum of White Box and Black Box analysis, various methods can be distinguished. For example, in white box analysis, the method of Static Application Security Testing (SAST) inspects source code for vulnerabilities without executing the program. It provides insights into code structure, quality, and potential security flaws early in the development cycle. At the other end of the spectrum is Dynamic Application Security Testing (DAST) for black box analysis, which evaluates the system during execution and simulates attacks within the context of software security. This method does not require source code access and is purely based on input/output (I/O), which is also the defining aspect of black box analysis. A third method, Interactive Application Security Testing (IAST), combines aspects of SAST and DAST by monitoring the application in real time to detect vulnerabilities. IAST tools provide immediate feedback on potential issues, combining the strengths of both static and dynamic analysis (YaunPan [2019]). A considerable amount of research has been conducted on various individual tools, how they relate to each other (Charoenwet et al. [2024]; Plch [2018]; Guß [2020]; Kaur and Nayyar [2020]; YaunPan [2019]; Li et al. [2024c]), and some studies provide an overall view of available tools (Li et al. [2023]; Fatima et al. [2018]; Cruz et al. [2023]).

In the scope of the research for this master thesis, SAST is the most relevant. Therefore, SAST will be further addressed, and DAST and IAST will be disregarded.

As a specialization within SAST, it is also worthwhile to mention Query-based Static Application Security Testing (Q-SAST), a method that has gained popularity in recent years. Zongjie Li et al.

have conducted research on how this method can detect C/C++ vulnerabilities (Li et al. [2024c]). Despite these techniques and their combinations, SAST has serious limitations, such as high false-positive rates, limited coverage of runtime vulnerabilities, and the inability to always find all vulnerabilities. All these drawbacks are confirmed by the studies mentioned above. For instance, the study "An Empirical Study of Static Analysis Tools for Secure Code Review" shows that at the function level, the worst-performing tool has a 4% detection rate, while the best-performing tool reaches 52% (Charoenwet et al. [2024]). The study also found a high false-positive rate, averaging over 76%.

Studies that provide an overview of available tools reveal a wide range of options. For example, the study “Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java” from 2023 shows that there are already 192 different SAST tools (Li et al. [2023]), and other studies indicate that even this list is incomplete by evaluating tools which are not included in the 192 (Fatima et al. [2018]; Charoenwet et al. [2024]). This list ranges from small research project tools to large tools offered by profitable or open-source organizations. They vary in popularity, effectiveness, focus area, programming language, and underlying technique. According to these studies, the following tools are most commonly used or yield the best results for C++: CodeQL, Clang, Coverity, SonarQube, CSTAT, Parasoft C/C++ test, CppCheck, SonarQube, and PVS-Studio.

Following industry standards, such as the OWASP Top Ten, ISO/IEC 27001, and NIST guidelines, is crucial for SAST tools to ensure security and compliance in development. Tools that meet these standards often offer a more comprehensive and structured approach to identifying vulnerabilities (Guß [2020]). Furthermore, tools that adhere to the CWE classification provide a better reference framework and can better be compared with other SAST tools (Li et al. [2023]).

C++ poses unique security challenges due to its low-level memory access and manual memory allocation. SAST tools specifically tailored to C++, like those mentioned earlier, offer specialized capabilities to detect memory-related vulnerabilities, buffer overflows, and unsafe casting. However, the effectiveness of each tool varies depending on code complexity, developer skill, and tool-specific features. It is clear that no tool offers 100% vulnerability detection coverage. Recent studies shows that Q-SAST analysis is a promising methods in achieving higher coverage (Li et al. [2024c]).

Furthermore, recent advancements in machine learning (ML) and large language models (LLMs) have introduced new possibilities for SAST tools. ML algorithms can enhance SAST by identifying patterns in large datasets, resulting in a reduction of the high false positive rate (Noever [2023]). LLMs, such as OpenAI’s GPT models, offer potential for contextual code understanding and can help developers write more secure code by identifying vulnerabilities during development. Studies conducted after the major advances in LLMs in 2022 (Haleem et al. [2022]) indicate that LLMs, regarding vulnerability detection, can outperform current SAST tools (Noever [2023]; Dozono et al. [2024a]; Hüther et al. [2023]). For example, D. A. Noever reports in his study that he achieved a 90% reduction in vulnerabilities using GPT-4. He also mentioned that the LLM is able to self-audit, suggest fixes and is more precise then traditional SAST tools.

All papers referenced thus far that address SAST tools or have developed a SAST tool as an output describe tools executed in various ways. Some tools operate as standalone external applications for code analysis, others are integrated into an Integrated Development Environment (IDE) through a plugin/extension mechanism, and some function as part of Continuous Integration (CI) systems with version control. None of the tools, operate directly within the browser to detect insecure code on websites, which is the aim of this master thesis.

## 3.4 Chrome Browser Extension

Browser extensions are software modules integrated into web browsers to enhance functionality beyond the browser's core offerings (Mehta [2016]). Common examples range from ad blockers and form auto-fill tools to more complex applications like security tools and collaborative plugins. These extensions can be obtained through online marketplaces or manually installed by users (Akshay Dev and Jevitha [2017]). For all major browsers extensions are available. So also for the Chrome browser. They can be built with web technologies such as HTML, CSS, and JavaScript (Mehta [2016]), Chrome extensions offer a variety of functionalities classified into primary categories:

1. Functional Enhancements: Extensions that introduce new features, such as task managers, improved search functionalities, or advanced navigation tools.
2. Security and Privacy: Extensions designed to enhance online safety, for instance, by blocking malicious websites, managing cookies, or providing encryption services.
3. Interface Customization: Extensions that adjust the graphical user interface (GUI) with themes, icons, or layout changes.
4. Integration with External Services: Extensions that enable connectivity with third-party applications, including cloud services, social media platforms, or productivity suites like Google Workspace.

Examples of widely used extensions are (helloleads [2020]):

1. Evernote Website Clipper: This tool allows users to save parts of web page – 4.000.000 users.
2. LastPass: A password manager for websites and apps – 10.000.000 users.
3. Grammarly: This tool checks the spelling of written text, such as emails, within the browser – 10,000,000 users.
4. Adblock Plus: A tool that blocks advertisement on websites – 10.000.000 users.
5. Google Translate and Dictionary: This tool, developed by Google, enables users to translate web pages – 10.000.000 users.
6. Todoist: This tool offers to-do list functionality – 600.000 users.

While extensions have notable advantages, such as boosting productivity and personalization of browsing experiences, they also introduce risks. Positive aspects include increased workflow efficiency, enhanced security, and a customizable browsing environment, making extensions valuable for professionals, regular users, and developers alike (Mehta [2016]). Extensions play a critical role in digital innovation by allowing developers to add new functionalities without the need to change the browser itself.

However, risks include potential security threats from malicious extensions, which can access personal data or contain harmful software (Akshay Dev and Jevitha [2017]). Moreover, having too many extensions may slow down the browser or cause conflicts with other web applications, highlighting the need for rigorous security protocols and oversight by browser developers and digital marketplaces. Google, for instance, reviews extensions before they are made available in its marketplace (Hsu et al. [2024]). Despite these measures, many extensions remain vulnerable to cyber threats, partially due to the lack of maintenance; approximately 60% of extensions are reportedly unmaintained (Jin et al. [2024]).

The architecture of Chrome's extension mechanism is managed through the 'Google Chrome Extensions API' (Google [2024]). This system includes several components, such as (see figure 3.1):

**The manifest:** A JSON file containing general extension information and managing extension permissions.

**UI Elements:** The extension API has access to different browser UI elements, such as 1) toolbar buttons, 2) browser menus, 3) webpage content, and 4) context menus.

**Background Script**: Also known as the 'Service Worker' runs independently in the background, enabling for instance storage access and external API interactions.

**Content Script:** Provides active functionality on web pages or adjusts visual elements as necessary.

The background and content scripts can communicate through a message/event listener mechanism, allowing seamless interaction between the two.

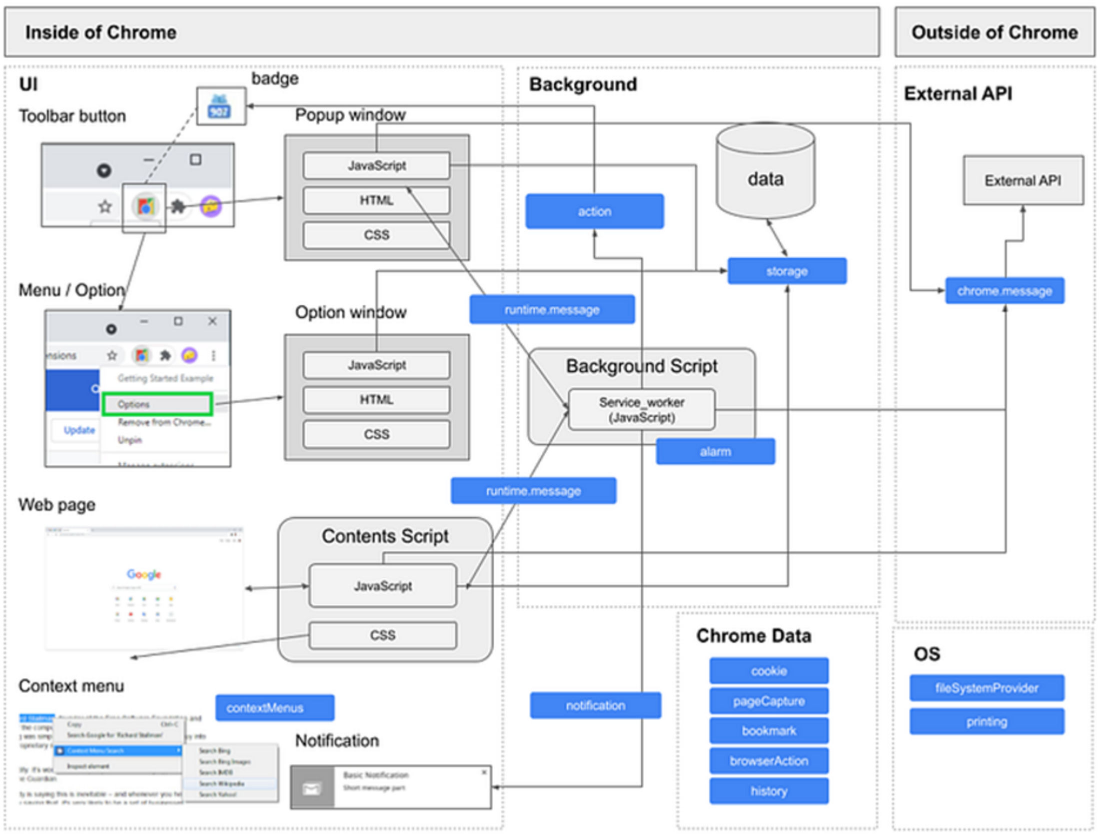


Figure 3.1: chrome API architecture, bron: medium.com

## 3.5 C++ vulnerabilities and categories

### 3.5.1 Sources for Categorizing Vulnerabilities

Different models developed by organizations such as Mitre categorize software vulnerabilities with the aim of gaining better control over the extensive list of potential vulnerabilities. Additionally, various authors of books have attempted to create classifications. While they largely follow the existing models, they also apply their own perspectives and categorizations. The different models that categorize vulnerabilities are:

- **Common Vulnerabilities and Exposures (CVE):** The CVE system (mitre [2025c]) provides a standardized, publicly accessible glossary of known information security vulnerabilities and exposures. According to their website, they currently classify vulnerabilities into 11 categories, including Overflow, Cross-Site Scripting, Memory Corruption, SQL Injection, File Inclusion, and Cross-Site Request Forgery, among others (cvedetails [2024]).

- **Common Weakness Enumeration (CWE):** CWE (mitre [2025d]) is a community-developed list of common software and hardware security weaknesses. It provides a hierarchical structure for organizing weaknesses. CWE offers various "views" tailored to specific needs. The "Software Development" view, relevant to this study, contains 399 weaknesses out of a total of 933 (according to their latest statistics), organized into 40 categories. Within this view, the "Weaknesses in Software Written in C++" view provides a targeted subset relevant to this research.

- **Common Attack Pattern Enumeration and Classification (CAPEC):** CAPEC (mitre [2025a]) complements CWE by providing a catalog of common attack patterns. It offers insights into how attackers exploit specific weaknesses. The CAPEC "Mechanisms of Attack" view is particularly relevant for understanding the practical exploitation of vulnerabilities.

- **OWASP Top 10:** The Open Web Application Security Project (OWASP) Top 10 (Owasp [2025]) provides an overview of the ten most critical security risks for web applications. Although focused on web applications, it offers valuable insights into common vulnerability patterns. The 2017 version, frequently cited, has since been updated with the 2021 version. As the name suggests, it focuses on the most common and critical security weaknesses in web applications.

- **STRIDE:** STRIDE (Microsoft [2009]) is a model for categorizing threats , classifying them as Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege. While useful for threat modeling, particularly during the architectural design phase, STRIDE is considered too abstract for identifying specific coding errors during code reviews.

- **Application Security Verification Standard (ASVS)**: ASVS (Owasp [2024]) provides a comprehensive standard for verifying the security of applications. It serves as a more detailed resource when a formal standard is required.

- ATT&CK: MITRE ATT&CK (mitre [2025b]) is a globally accessible knowledge base of attacker tactics and techniques based on real-world observations. It offers valuable context for understanding how vulnerabilities are exploited in practice.

In addition to these models, the following security-focused books describe categories of vulnerabilities:

- **Security in Computing (5th Edition)** by Pfleeger (Pfleeger [2002]): This book provides a broad overview of computer security concepts, including several chapters relevant to software vulnerabilities. Specific weaknesses discussed include Buffer Overflow, Incomplete Mediation, Time-of-Check to Time-of-Use, Undocumented Access Points, Off-by-One Errors, Integer Overflow, Unterminated Null-Terminated Strings, Parameter Length, Type, and Number Issues, Unsafe Utility Programs, and Race Conditions. It also covers web-specific attacks, browser vulnerabilities, and methods for acquiring user or website data, including SQL Injection and Cross-Site Scripting (XSS).

- **24 Deadly Sins of Software Security** by Howard, LeBlanc, and Viega (Howard et al. [2011]): This book describes 24 common security flaws, including Buffer Overruns, Integer Overflows, Format String Vulnerabilities, Improper Input Validation, Race Conditions, SQL Injection, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), Path Traversal, Weak Cryptography, Insecure Storage, Insufficient Logging and Monitoring, Improper Error Handling, Insecure Direct Object References, Misconfiguration, Use of Magic Numbers, Failure to Use Security Features, Insufficient Authentication and Authorization, Hardcoded Credentials, Insecure APIs, Memory Management Issues, Unsafe Use of APIs, Improper Use of Temporary Files, and Failure to Keep Software Up to Date.

- **The Web Application Hacker's Handbook (2nd Edition)** by Stuttard and Pinto (Stuttard [2011]): This book serves as a comprehensive guide to web application security, addressing various attack vectors and vulnerabilities. Chapter 19 specifically focuses on common source code vulnerabilities, such as Cross-Site Scripting (XSS), SQL Injection, Path Traversal, Arbitrary Redirection, OS Command Injection, Backdoor Passwords, Native Software Bugs, and Source Code Comments. The book also covers platform-specific topics for Java, ASP.NET, PHP, Perl, and JavaScript, including identifying user-supplied data, session interaction, potentially dangerous APIs, and configuration aspects.

It is important to acknowledge that these sources often use different categorization schemes, leading to overlaps and variations in terminology. These categorizations are not mutually exclusive but rather represent different levels of granularity and perspectives on the same underlying issues. CWE addresses this with its hierarchical structure, offering a more detailed and organized view, while CAPEC focuses on the attack patterns used to exploit these weaknesses. OWASP Top 10, though limited in scope, highlights the most common and critical risks for web applications. Therefore, a combination of all these sources is necessary for a comprehensive overview.

### 3.5.2 C/C++ Vulnerabilities

C and C++ are particularly prone to certain types of vulnerabilities due to their low-level nature and manual memory management. According to the TIOBE index, C and C++ ranked 2nd and 4th among the most popular programming languages (Zhang et al. [2021]). Furthermore, among the 839 CWE types, C/C++ account for the most reported types, with C having 80 and C++ 84. In comparison, Java has 73 and PHP 23. This makes C/C++ the languages with the highest number of security issues (Zhang et al. [2021]).
Various sources emphasize the most common and critical vulnerabilities in C++ code. Snyk, a security platform, identifies the following as the top 5 C++ security risks (Team [2022]):

1. **Buffer Overflow:** Occurs when a program attempts to write data beyond the allocated memory buffer.

2. **Integer Overflow and Underflow:** Happens when an arithmetic operation results in a value outside the representable range of the type.

3. **Pointer Initialization:** Issues arising from uninitialized or improperly initialized pointers.

4. **Incorrect Type Conversion:** Problems caused by unsafe or unintended type conversions.

5. **Format String Vulnerability:** Occurs when user-supplied input is directly used in format string functions, potentially allowing attackers to execute arbitrary code.

These vulnerabilities arise from C++'s manual memory management and lack of built-in boundary checks, requiring developers to take extra care in handling memory allocation, pointers, and input validation. Additionally, CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer), CWE-416 (Use After Free), CWE-190 (Integer Overflow or Wraparound), CWE-476 (NULL Pointer Dereference), and CWE-415 (Double Free) are frequently detected vulnerabilities in responses on StackOverflow.

### 3.5.3 Examples of CWEs within Software Development Categories

To provide a more concrete understanding of the CWEs selected for this research, this section presents examples within the context of the Software Development (SD) categories. The selection focuses on vulnerabilities related to pointers and memory management, which are particularly relevant to C++ and the scope of this research. The following CWEs are grouped under their respective top-level categories (mitre [2025d]):

- Improper Resource Shutdown or Release (CWE-404): This category encompasses errors that occur when resources (e.g., memory, files) are not properly released or shut down.
  - CWE-401: Missing Release of Memory after Effective Lifetime: This vulnerability occurs when dynamically allocated memory is not freed after it is no longer needed.
  - CWE-772: Missing Release of Resource After Effective Lifetime: This vulnerability is similar to CWE-401 but is a more general case where a resource, broader than just memory, is not released. This can lead to resource exhaustion and reduced performance.
  - CWE-762: Mismatched Memory Management Routines: This vulnerability arises when there is inconsistent memory management, such as allocating memory with new and releasing it with free. This can lead to heap corruption and unpredictable behavior.
- Incorrect Calculation (CWE-682): This category deals with errors in calculations that can lead to vulnerabilities.
  - CWE-131: Incorrect Calculation of Buffer Size: This vulnerability occurs when the required size for a buffer is calculated incorrectly. This can lead to buffer overflows when more data is written to the buffer than there is space for.
- Pointer Issues (CWE-465): This category includes vulnerabilities related to incorrect use of pointers, such as dereferencing or assignment.
  - CWE-587: Assignment of a Fixed Address to a Pointer: This vulnerability arises when a pointer is assigned to a fixed, hardcoded memory address. This can be dangerous if the address is in a protected or inaccessible memory area.
  - CWE-476: NULL Pointer Dereference: This vulnerability occurs when a program attempts to access a memory location through a pointer that has the value NULL. This usually results in a crash of the application.
- Operation on a Resource after Expiration or Release (CWE-672): This category includes the use of resources after they have been released or their validity has expired.

  - CWE-416: Use After Free: This vulnerability arises when memory is still used after it has been freed. This can lead to unpredictable behavior or can be exploited by an attacker.
  - CWE-415: Double Free: This vulnerability occurs when a program tries to free the same memory block multiple times. This can cause corruption of the heap and potentially lead to code execution.

- Improper Input Validation (CWE-20): This category includes vulnerabilities that arise because external input is not sufficiently checked for validity.
  - CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer: This is an overarching category for vulnerabilities where a program reads or writes outside the boundaries of an allocated memory buffer. This can lead to crashes, data corruption, or code execution.
  - CWE-787: Out-of-bounds Write: This vulnerability occurs when a program writes to a memory location outside the boundaries of an allocated buffer. This can lead to overwriting other data or code.

## 3.6 AI Models in context of Vulnerability Detection

This subsection explores the application of next-generation pretrained AI models, particularly Large Language Models (LLMs), for detecting vulnerabilities in code. It discusses the evolution of LLMs, their capabilities in software development, and recent research examining their effectiveness in identifying security issues.

### 3.6.1 The Evolution and Their Application in Software Development

The field of Natural Language Processing (NLP) has seen significant advancements with the introduction of LLMs. Before 2022, the landscape was considerably different, with limited availability and capability of models for public use. The release of models such as ChatGPT in late 2022 marked a turning point, making powerful LLMs accessible to a wider audience and sparking interest in their application across various domains (Haleem et al. [2022]).

Prior to this development, the application of AI in software development, including security-related tasks, was primarily dominated by more traditional machine learning techniques and static analysis tools. While these tools were useful, they often struggled to handle the complexity and nuances of modern programming languages and the evolving nature of software vulnerabilities (source4). They also lacked the ability to reason about code in a manner akin to a human developer. The following table provides an overview of some notable LLMs, including a selection of pre-2022 models for context:

| **Model** | **Release Date** | **Key Features** |
|---|---|---|
| BERT | 2018 | Bidirectional Encoder Representations from Transformers. Pre-2022, foundational for NLP tasks. |
| RoBERTa | 2019 | A robustly optimized BERT pretraining approach. Pre-2022, improved upon BERT's performance. |
| GPT-2 | 2019 | Pre-2022, demonstrated impressive text generation capabilities. |
| GPT-3 | 2020 | Pre-2022, significantly larger than GPT-2, showed few-shot learning capabilities. |
| InstructGPT-3 | Early 2022 | Pre-2022, Instruction-tuned version of GPT-3. |
| ChatGPT (GPT-3.5) | Late 2022 | Publicly accessible, conversational, based on GPT-3.5. Marked a turning point in LLM availability. |
| GPT-4 | March 2023 | Significant improvement over GPT-3.5, larger model, better reasoning. |
| GPT-4 Turbo | November 2023 | Faster and more cost-effective version of GPT-4. |
| GPT-40 | May 2024 | Latest iteration, enhanced performance, and multi-modal capabilities. |
| Gemini Ultra | Early 2024 | Google's advanced LLM, competitive with GPT-4. |
| Claude 2 | July 2023 | Anthropic's LLM, known for its large context window. |
| Claude 3 Opus | March 2024 | Anthropic's most powerful model, claimed to surpass GPT-4 on some benchmarks. |
| LLaMA-2-70b-chat-hf | July 2023 | Large, powerful model from Meta, open-source. |
| zephyr-7b-alpha | October 2023 | Smaller, efficient model, good for specific tasks. |
| zephyr-7b-beta | November 2023 | Improved version of zephyr-7b-alpha. |
| Mistral-7B-Instruct-v0.1 | September 2023 | High-performing 7B parameter model, instruction-tuned. |
| Turdus | N/A | No specific release date found, likely a smaller, specialized model. |

| **Model** | **Release Date (approx.)** | **Key Features** |
|---|---|---|
| LLaMA-3-70B | April 2024 | Latest iteration of Meta's LLaMA model, 70 billion parameters. |
| LLaMA-3-8B | April 2024 | Smaller version of LLaMA-3, 8 billion parameters. |
| Gemma-7B | February 2024 | Google's open-source model, 7 billion parameters. |
| Mixtral-8x7B | December 2023 | Mixture-of-Experts model from Mistral AI, known for its efficiency. |
| CodeLlama-7B | August 2023 | Meta's code-specific LLM, 7 billion parameters. |
| CodeLlama-13B | August 2023 | Larger version of CodeLlama, 13 billion parameters. |
| CodeGemma-7B | February 2024 | Google's code-specific LLM, built on Gemma, 7 billion parameters. |

The emergence of publicly accessible, powerful LLMs has spurred research into their application for various software development tasks, including code generation (Dozono et al. [2024b], Hou and Ji [2024]), code completion, bug fixing, and vulnerability detection. Within the domain of software security, studies have shown that certain LLMs, particularly larger models such as GPT-4, demonstrate strong performance in code-related tasks. These models often outperform traditional methods and, in specific scenarios, even exceed the capabilities of human programmers (Hou and Ji [2024]).

### 3.6.2 Research on LLM-Based Vulnerability Detection

Since the widespread availability of LLMs, there has been a significant increase in research exploring their potential for vulnerability detection. This trend is reflected in the growing number of papers published on this topic after the initial research proposal for this thesis in April 2024. It is a rapidly evolving field, with new models and techniques continuously emerging. The following list provides a non-exhaustive overview of recent studies in this area:

1. Large Language Models for Secure Code Assessment: A Multi-Language Empirical Study (Dozono et al. [2024b])
2. Comparing large language models and human programmers for generating programming code (Hou and Ji [2024])
3. Detecting Insecure Code with LLMs (Buehler [2024])

4. Harnessing Large Language Models for Software Vulnerability Detection: A Comprehensive Benchmarking Study (Tamberg and Bahsi [2024])

5. Beyond ChatGPT - Enhancing Software Quality Assurance Tasks with Diverse LLMs and Validation Techniques (Widyasari et al. [2024])

6. LLM-Assisted Static Analysis for Detecting Security Vulnerabilities (Li et al. [2024a])

7. Code Vulnerability Detection: A Comparative Analysis of Emerging Large Language Models (Sultana et al. [2024])

8. StagedVulBERT Multi-Granular Vulnerability Detection with a Novel Pre-trained Code Model (Jiang et al. [2024])

9. NAVRepair: NodeType Aware C and CPP Code vulnerability repair (Sheng et al. [2024b])

10. LProtector an LLM-driven Vulnerability Detection (Sheng et al. [2024a])

11. A Qualitative Study on Using ChatGPT for Software Security Perception vs. Practicality (Kholoosi et al. [2024])

12. Towards Effectively Detecting and Explaining Vulnerabilities Using Large Language Models (Mao et al. [2024])

13. Vul-RAG Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG (Du et al. [2024])

14. "Automated Software Vulnerability Static Code Analysis Using Generative Pre-Trained Transformer Models (Pelofske et al. [2024])

15. Language Models can Solve Computer Tasks (Kim et al. [2024])

These studies share a common foundation in their exploration of LLMs for vulnerability detection but differ in their specific approaches, models used, datasets, and evaluation metrics.

**Similarities:**

1. Use of LLMs: All studies leverage LLMs as a core component for vulnerability detection.

2. Focus on Automation: They aim to automate or semi-automate the process of identifying vulnerabilities, reducing the need for manual effort.

3. Evaluation on Datasets: Most studies evaluate their methods on established vulnerability datasets or create new ones for this purpose.

4. Comparison with Baselines: Many studies compare LLM performance with traditional static analysis tools or other existing methods.

**Differences:**

1. LLM Models Used: Studies employ various LLMs, ranging from widely used models like GPT-3.5 and GPT-4 to specialized models such as CodeLlama and CodeGemma, as well as open-source alternatives like Llama 2, Mistral, and Zephyr.

2. Prompting Strategies: Different prompting strategies are explored, including zero-shot, few-shot, Chain-of-Thought (CoT), Tree of Thoughts (ToT), and newer methods such as Recursive Criticism and Improvement (RCI) (Kim et al. [2024]) and detailed security assessment instructions (Tamberg and Bahsi [2024]).

3. Fine-Tuning vs. Prompt Engineering: Some studies focus on fine-tuning LLMs for vulnerability detection ( Sultana et al. [2024], Jiang et al. [2024], Mao et al. [2024]), while others rely primarily on prompt engineering (Dozono et al. [2024b], Hou and Ji [2024], Buehler [2024], Tamberg and Bahsi [2024], Widyasari et al. [2024], Kholoosi et al. [2024], Pelofske et al. [2024], Kim et al. [2024]).

4. Integration with Other Techniques: Certain approaches combine LLMs with other techniques, such as static analysis (Li et al. [2024a]), retrieval-augmented generation (RAG) (Sheng et al. [2024a], Du et al. [2024]), or voting mechanisms (Widyasari et al. [2024]).

5. Granularity of Detection: Some studies target coarse-grained detection (e.g., function level), while others address fine-grained detection (e.g., statement level) (Jiang et al. [2024]).

6. Programming Languages: While many studies focus on C/C++ (Sultana et al. [2024], Jiang et al. [2024], Sheng et al. [2024b], Sheng et al. [2024a], Pelofske et al. [2024]), others explore Python (Buehler [2024]), Java (Tamberg and Bahsi [2024], Li et al. [2024a]), and multiple languages (Dozono et al. [2024b]).

7. Dataset Origins: The datasets used vary, including real-world vulnerabilities (Li et al. [2024a], Kholoosi et al. [2024], Du et al. [2024]), synthetic datasets, and composite datasets from platforms like GitHub (Buehler [2024], Sultana et al. [2024]).

8. Evaluation Metrics: Although common metrics such as accuracy, precision, recall, and F1-score are widely used, some studies introduce new evaluation metrics or methodologies (Kholoosi et al. [2024], Mao et al. [2024], Du et al. [2024] ).

### 3.6.3 Prominent Prompting Strategies: CoT and RCI

Prompt engineering is crucial for maximizing the effectiveness of LLMs in vulnerability detection. A deliberate approach to crafting prompts for an LLM to generate a desired output or outcome for a specific task is referred to as a strategy. Two prominent strategies discussed in the literature are Chain-of-Thought (CoT) and Recursive Criticism and Improvement (RCI).

**Chain-of-Thought (CoT)** CoT prompting encourages the LLM to generate a sequence of intermediate reasoning steps before arriving at the final answer. This approach has been shown to improve the model's ability to perform complex reasoning tasks. In the context of vulnerability detection, CoT can guide the LLM through analyzing code, identifying potential vulnerabilities, and explaining the reasoning behind its assessment (Tamberg and Bahsi [2024], Mao et al. [2024]).

**Recursive Criticism and Improvement (RCI)** RCI is a more recent prompting technique where the LLM is instructed to critically evaluate its own output, identify potential errors or weaknesses, and generate an improved response based on this self-critique (Kim et al. [2024]). This iterative process can enhance the reasoning and problem-solving capabilities of the LLM. RCI has demonstrated promising results in tasks requiring complex reasoning and planning, and its application to vulnerability detection is an emerging area of research. These studies highlight the need for advanced prompting techniques to unlock the full potential of LLMs for complex code analysis tasks.

### 3.6.4 Hallucination

Furthermore, it is noteworthy that LLMs are susceptible to hallucination (Sultana et al. [2024]). This means that an LLM can generate a coherent and grammatically correct response that is, in fact, inaccurate or nonsensical.

# Chapter 4

# Research Methodology

## 4.1 Overview of Design Science Research (DSR)

This research has Design Science Research (DSR) as its primary research methodology. DSR is a proven methodology mainly used in information sciences and engineering research. It focuses on the creation and evaluation of an artefact designed to solve a practical problem (vom Brocke et al. [2020]). In contrast to purely theoretical research, DSR emphasizes the practical application of scientific knowledge, referred to as Design Knowledge (DK), to develop innovative solutions. This method aligns well with the applied scientific character of the HBO Master's program. The components of the artefact include: models, program code, documentation, a defined test setup, and test results. The core principle of DSR lies in its iterative and problem-solving nature, where the artefact is developed, evaluated, and refined through a systematic process. Reflecting the iterative nature of DSR, the prototype was developed and refined through multiple iterations.

### 4.1.1 Justification for the Use of DSR

The choice of DSR for this research is motivated by several factors. Firstly, the research is aimed at investigating and finding a solution to a practical problem: the fact that students use unsafe code snippets from online Q&A platforms and thus learn unsafe techniques. This aligns with DSR's problem-oriented approach, where research is focused on solving an identified problem in practice. Secondly, the research aims to develop an artefact (the browser extension) to help alleviate this problem. DSR provides a suitable framework for analyzing, designing, implementing, and evaluating/testing the artefact. Finally, DSR facilitates the generation of both practical and theoretical contributions. The development and evaluation of the artefact contribute to practical problem-solving, and the insights gained during the research process contribute to the knowledge base.

### 4.1.2 DSR Process Model

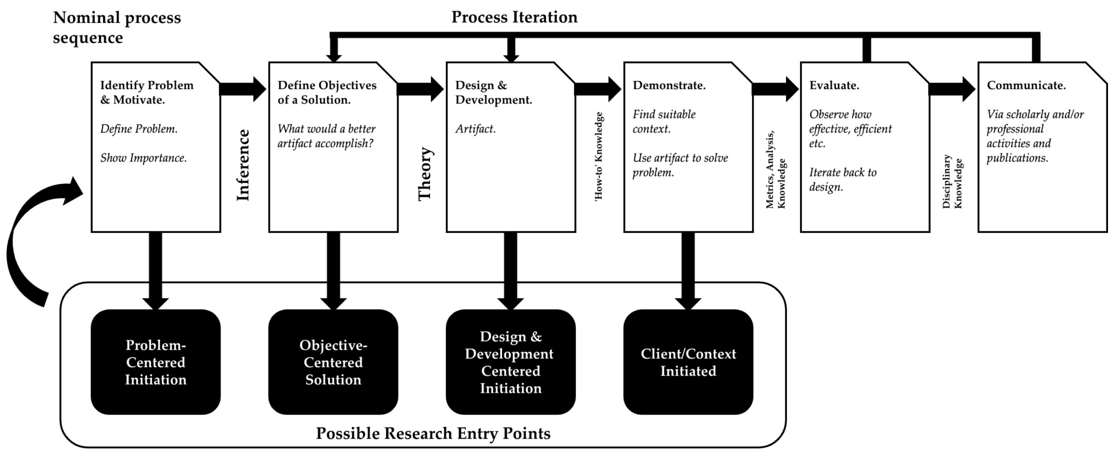

Nominal process sequence
Process Iteration
Identify Problem & Motivate.
Define Problem.
Show Importance.
Inference
Define Objectives of a Solution.
What would a better artifact accomplish?
Theory
Design & Development.
Artifact.
'How-to' Knowledge
Demonstrate.
Find suitable context.
Use artifact to solve problem.
Metrics, Analysis, Knowledge
Evaluate.
Observe how effective, efficient etc.
Iterate back to design.
Disciplinary Knowledge
Communicate.
Via scholarly and/or professional activities and publications.
Problem-Centered Initiation
Objective-Centered Solution
Design & Development Centered Initiation
Client/Context Initiated
Possible Research Entry Points

Figure 4.1: DSR model overview (source: www.researchgate.net)

The research, and therefore also this thesis, is structured according to the six-step process model proposed by Peffers et al. (Peffers et al. [2007]). These 6 steps are visualized in Figure 4.1, and the specific implementation followed during the research will now be explained in further detail.

## 4.2 Research Phases

### 4.2.1 Identify problem and motivate

This phase was completed once the research proposal was approved. The proposal included a formulated research question, an initial description of the research methodology, and a thorough analysis of the problem domain, including a review of relevant literature. This literature study provided an initial understanding of the existing research landscape related to secure programming, AI, and didactics. The majority of this preliminary research is incorporated into the 'Introduction' chapter of this thesis.

### 4.2.2 Define Objectives of a Solution

This phase, also partially addressed in the research proposal, involved defining the objectives for the artefact. Preliminary research regarding to which programming language, which vulnerabilities, and which AI models to use, helped determine the scope and to identify initial requirements for the artefact. These requirements are documented in the Software Requirements Specification (SRS). Additionally, a qualitative semi-structured interview was conducted with a domain expert to gain further insight into the problem regarding didactics and secure programming in C++. Assistant professor Dr. E. van der Kouwe was approached and agreed to participate.

A structured process, established and followed during the literature study, guided the development of the theoretical framework. The steps were as follows:

1. **Subject List Definition:** Based on the research proposal and preliminary research, a conceptual list of subjects was determined. A literature study was then conducted for each subject. The following steps were therefore repeated for each subject.
2. **Keyword Determination:** Keywords for the subject were determined. For the subject 'Secure programming,' for example, these were: secure coding, insecure code, secure programming, programming vulnerabilities, defensive programming, secure coding history, secure coding overview, and secure coding review.
3. **Academic Search Engine Utilization:** Academic search engines, including Google Scholar, IEEE Xplore, ACM Digital Library, and ScienceDirect, were used to search for relevant publications using the identified keywords. The various keywords were also combined and/or restricted with logical operators.
4. **Initial Paper Selection and Related Paper Identification:** The most relevant paper was then used to get an overview of all related papers etc. The website: 'connectedpapers.com' was used for this step.
5. **Paper Selection:** Based on the initial list of papers from step 3 and the result from step 4, a selection was made. The condition was that a paper should not be older than 15 years, and preferably shorter. From 10 to 15 papers, the title, publication year, abstract, keywords, and notes were collected and placed in a single Word document.
6. **Sub-topic Determination and Writing:** Sub-topics were determined and written with a non-official source reference.
7. **Language Review:** A language review was conducted.
8. **Thesis Integration:** The text was placed in a LaTeX based thesis document.
9. **Content Review:** A content review was conducted by a researcher from TNO.
10. **Feedback Processing and Citation Formalization:** Feedback was then processed, and the source was added according to APA style.

### 4.2.3 Design and Development

This phase focuses on the design and development of the artefact. The architecture of the artefact was designed using UML diagrams in the tool Astah (Astah [2024]). A high-fidelity prototype was developed using Figma (Figma [2024]) to visualize the user interface and user experience. The development process followed a three-step iterative prototyping approach. Each iteration involved refining the design, implementing new features, and conducting tests. The first iteration was the basis for a browser extension, the second iteration featured the implementation of the ’website mode,’ and the third iteration was the implementation of the ’dataset mode.’ During the evaluation phase, development was regularly revisited to realize minor improvements and adjustments.

### 4.2.4 Demonstration

This phase involved conducting the experiments and obtaining data for evaluation. Based on the literature study regarding datasets, a longlist of datasets was compiled. This longlist was then theoretically assessed and resulted in a shortlist of 6 datasets. These datasets were all tested and assessed for usability. The result was three datasets, of which 2 ultimately received an implementation. A series of experiments were conducted using these datasets, ensuring that each experiment was performed at least twice to validate the consistency of the results. Various metrics were collected during the experiments.

### 4.2.5 Evaluation

This phase focuses on evaluating the performance of the different LLMs based on the data collected during the demonstration phase. The collected data was interpreted and analyzed, the process was supported by the tool Julius.ai (julius [2024]). The results are visualized using graphs to facilitate understanding and interpretation. Subsequently, conclusions were drawn from the analysis.

### 4.2.6 Communication

The final phase involves communicating the research findings. This is achieved through the writing of this thesis, which documents the entire research process, including the problem statement, literature study, methodology, design and development of the artefact, demonstration, and evaluation. The thesis is written using LaTeX and is based on a LaTeX thesis template developed by Jan Küster called ’All Inclusive Master Thesis Computer Science.’

## 4.3 Project Management Tools

Throughout the research project, Trello (Trello [2024]) was used for task management and progress tracking, while a Gantt chart was used for visualizing the project timeline and communication to stakeholders.

# Chapter 5

# Design and Development

## 5.1 Problem and Goal analysis

As previously described, students often retrieve code snippets from Q&A websites. One of the goals of this research, and therefore the artefact, is to provide static code analysis information as close to the source as possible. In this context, 'close to the source' is at the source website itself and within the browser. Traditional tools need to run on a server or a local PC and lack integration with browsers. Moreover, most tools focus on a limited selection of programming languages, making it impossible to analyze every programming language with a single tool.

Therefore, this research explores whether this process can be simplified using a Large Language Model (LLM). Without requiring the installation of 'Static Code Analysis Tools,' any programming language can be analyzed with a single API call. Another issue with traditional static code analysis tools, that is mentioned in the literature review, is their relatively low detection rate and high false positive rate.

First, an analysis was conducted to determine the requirements the artefact must meet and how it will be used. This analysis is documented in a 'Software Requirement Specification' (SRS) and is part of the deliverables (see appendix .1). Input for the analysis was drawn from personal expertise in C++ programming and didactics, online sources referenced in the 'Literature Review' chapter, and an interview (see Apendix .7) with an expert in secure C/C++ programming and didactics. Key insights from the interview include the following:

1. Providing explanations is essential, not only stating what can be improved but also explaining why.
2. In the context of C/C++ vulnerabilities, the most common errors students make are classic memory issues, improper use of pointers, buffer overflows, and use-after-free errors.

3. If the goal is to teach practical skills, this is best integrated into regular programming courses.
4. Theoretical knowledge is crucial for understanding the nature of the threats.
5. LLMs can significantly save time both in developing tools and during their use.

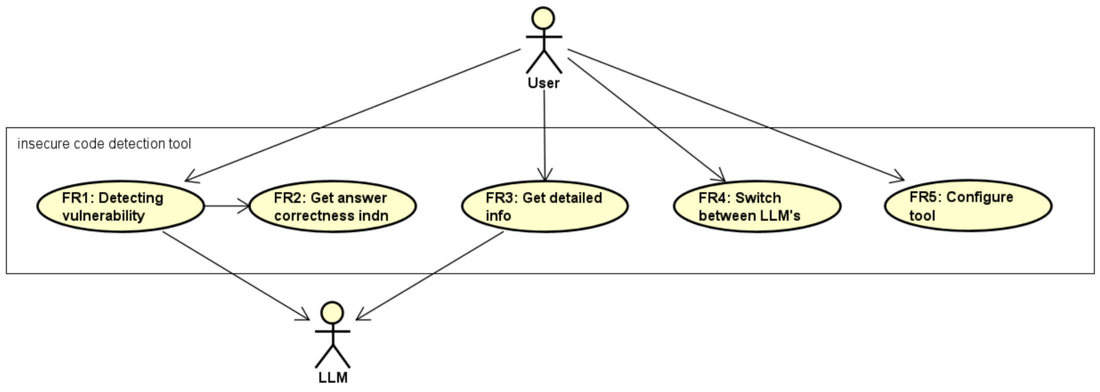

User
insecure code detection tool
FR1: Detecting vulnerability
FR2: Get answer correctness indn
FR3: Get detailed info
FR4: Switch between LLM's
FR5: Configure tool
LLM

Figure 5.1: high level usecases / requirements

As outlined in the SRS, various stakeholders and users have been identified. Primarily, these are students and the researcher conducting this study. The researcher plays a key role because the tool must demonstrate and compare the performance of different LLMs. This is necessary to answer the main research question regarding the relevance of information provided by LLMs. The solution, therefore, has two goals: creating a browser-based artefact capable of detecting vulnerabilities in code snippets and providing a setup for testing and comparing LLM output related to the detection of insecure code. These goals have resulted in the use cases / high-level functional requirements shown in Figure 5.1.

Detecting vulnerabilities (FR1) is separated from displaying detailed information (FR3), as a simple indicator must first be shown for a code snippet to immediately signal whether the code is safe. The user can then choose to access detailed information. This approach prevents the user from being overwhelmed with information directly. Additionally, for the research, it is important to be able to automate the evaluation of how well different LLMs perform, which is not feasible with detailed information.

Furthermore, functionality is required to determine how reliable the response from an LLM is. As noted earlier in the 'Literature Review,' an LLM can hallucinate and will always provide an answer, even if the answer is incorrect. Therefore, it is desirable to inform the user about this and indicate the reliability of the response. The complete list of requirements is categorized according to FURPS and can be reviewed in the SRS. FURPS is an acronym representing a model used to categorize the requirements of software.

## 5.2 Design and Realisation

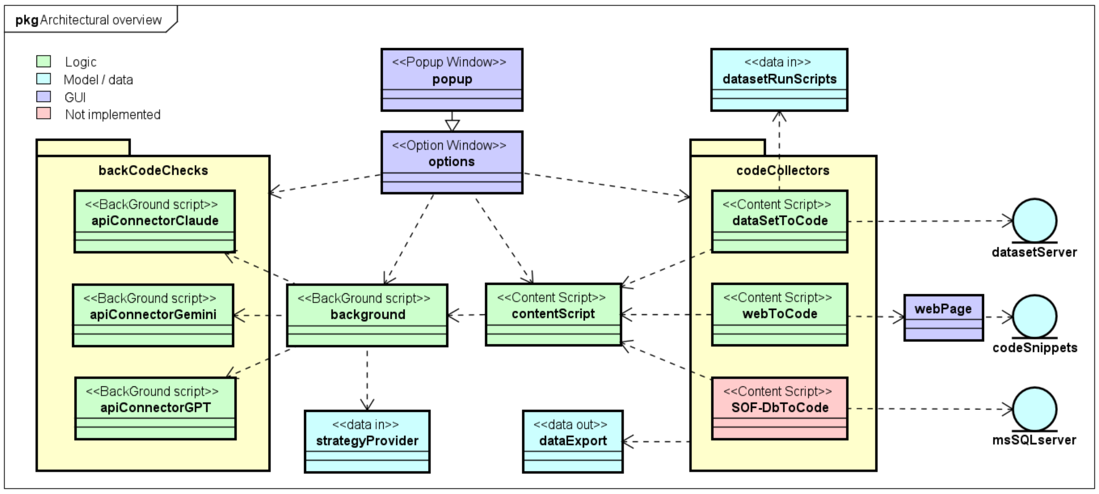

pkg Architectural overview
Logic
Model / data
GUI
Not implemented
<<Popup Window>> popup
<<Option Window>> options
<<data in>> datasetRunScripts
backCodeChecks
<<BackGround script>> apiConnectorClaude
<<BackGround script>> apiConnectorGemini
<<BackGround script>> apiConnectorGPT
<<BackGround script>> background
<<Content Script>> contentScript
codeCollectors
<<Content Script>> dataSetToCode
<<Content Script>> webToCode
<<Content Script>> SOF-DbToCode
webPage
datasetServer
codeSnippets
msSQLserver
<<data in>> strategyProvider
<<data out>> dataExport

Figure 5.2: architectural overview

### 5.2.1 Architectural Overview

The final design is based on the architecture of the ‘browser extension mechanism’ (see Figure 3.1). This is reflected in the architectural overview of the developed artefact (see Figure 5.2). The stereotyping, such as «Content Script», indicates the relationship with the extension mechanism. This mechanism, which includes asynchronous communication between the ‘Content Script’ and ‘Background Script,’ ensures that loading the website does not take significantly longer than usual.

On the right side of the diagram, the ‘codeCollectors’ are modeled, two of which have been implemented. The ‘webToCode’ component extracts code snippets from websites like StackOverflow.com and provides functionality for the end user, in this case the student or software developer in general. The ‘dataSetToCode’ component retrieves code snippets from the dataset server and is intended for conducting experiments during the research. This means that the system has two operating modes: 1) Website Mode: For extracting and analyzing code snippets directly from websites and 2) Dataset Mode: For retrieving and analyzing code snippets from a dataset server for research experiments.

The ‘browser extension mechanism’ is officially based on HTML/CSS in combination with JavaScript. For this project, the decision was made to use TypeScript instead of JavaScript.

TypeScript is an extension of JavaScript and offers several advantages. It provides tools for writing robust and error-free code through static type checking and clear error messages during development. Additionally, TypeScript leverages IntelliSense (code completion tool) more effectively. These benefits make it a better choice than JavaScript.

The codebase is structured with all the recommended TypeScript components, such as Webpack, React, and Jest. Unit tests have been implemented using Jest, though high test coverage will be achieved at a later stage. For processing the data from the experiment, Python is used. Python was chosen due to its simplicity and extensive library support, making it the most widely used language in the data analysis community. For analyzing and visualizing the data, the following Python libraries were used: Matplotlib, Seaborn, and Pandas.

The chosen IDE is VS Code, with the following top-level folder structure:

- dataset_result (experiment output and Python analysis scripts)
- dataset_run_scripts (run scripts for the Megavul dataset)
- dataset_server (web server and datasets)
- public (HTML components and extension manifest)
- src (TypeScript code and unit tests)

GitHub is used for version control. The project is hosted at the following location: github.com/ecna/QAInsecureCodeDetection and is marked as a private repository. Access can be requested.

### 5.2.2 Strategies

As highlighted in the 'Literature Review,' prior research has explored possible prompts to request LLMs to analyze code and suggest improvements. Among these, the following approaches were used:

1. Recursive Criticism and Improvement (RCI) Prompt
2. Chain-of-Thought (CoT) Prompt

Based on the prompts evaluated in various papers, exploratory research was conducted to determine which strategies work well in the context of this study. This process ultimately resulted in four different prompt strategies (see Appendix .3). As explained earlier, a strategy refers to a deliberate approach to crafting prompts for an LLM. Strategies 1, 2, and 3 are designed to enable automated verification of the question and answer. To ensure reliable results, the experiment must yield a large number of samples, which would not be feasible to evaluate manually. For this reason, explicitly requesting the response in a specific JSON format is part of the prompt. The format is as follows:

Listing 5.1: Strategy 1 2 and 3 response

```
{
    "isCodeSecure": false
    "CWEs":[0,0,0]
}
```

When the LLM identifies that a code snippet is insecure, it is asked to provide the relevant CWEs. This question is essentially two different questions in one: 1) Identify the vulnerability, and 2) Classify the vulnerability under the corresponding CWE. This requires the LLM to be familiar with the CWEs and the ability to map a vulnerability to the relevant CWE. Given the research question, this approach has a drawback: the LLM may excel at identifying vulnerabilities but perform poorly at naming the corresponding CWE. This could lead to the incorrect conclusion that the LLM is unsuitable, even if it is effective. However, the advantage of this approach is that since the dataset specifies which CWE is present in a code snippet, the correctness of the response can be verified automatically and that is what is needed for this research. What is considered a correct answer is explained in the Evaluation chapter.

Three strategies were devised to examine the impact of the phrasing of the prompt on the response:

1. Simple and minimal instruction.
2. Detailed instruction with a broad scope.
3. Detailed instruction with a narrow scope.

Strategy 1 provides the LLM minimal instructions, while Strategies 2 and 3 use detailed instructions based on a combination of RCI and CoT. The difference between Strategy 2 and Strategy 3 is that Strategy 2 ask to add potential CWEs , whereas Strategy 3 excludes them.

Strategy 4 on the other hand, is not intended for automated evaluation but serves as the final version where detailed information is requested. It is also based on RCI and CoT. This detailed information consists of the following components:

1. Security Assessment
    (a) Secure/Insecure
    (b) Explanation

2. Identified Vulnerabilities
   (a) Issue 1: Description (CWE-ID)
   (b) Issue 2: Description (CWE-ID)
   (c) ...
3. Suggestions for Improvement
   (a) Proposed solution for Issue 1
   (b) Proposed solution for Issue 2
   (c) ...
4. Final Version of Secure Code

This output is also in JSON format, as it simplifies processing the response in the webpage, though it is not required for automated evaluation. This format, however, enables manual checking of the response's validity.

### 5.2.3 Dataset and Dataset Server

As highlighted in the 'Literature Review,' several datasets contain code snippets labeled with CWE numbers, among other annotations. The MegaVul (Ni et al. [2024]) and SARD (Nist [2024a]) datasets emerged as the most useful, and functionality has been developed to utilize them during the experiment. Both datasets are provided via a dataset web server. This is necessary because the 'browser extension mechanism' does not allow direct access to the hard drive for security reasons. Therefore, the dataset is requested by the 'Content Script' from the 'dataset server' through a REST API.

For the MegaVul dataset, run scripts define which CWE numbers to retrieve and how many samples should be fetched (see Appendix .4). This allows the dataset size for the experiment to be determined dynamically and at runtime.

Listing 5.2: original SARD code

```
void CWE190_Integer_Overflow__char_fscanf_add_05_bad()
{
    char data;
    data = ' ';
    if(staticTrue)
    {
        /* POTENTIAL FLAW: Use a value input from the console */
        fscanf (stdin, "%c", &data);
    }
    if(staticTrue)
    {
        {
            /* POTENTIAL FLAW: Adding 1 to data could cause an overflow */
            char result = data + 1;
            printHexCharLine(result);
        }
    }
}
```

Listing 5.3: abstracted SARD code

```

void lkjhgfdsaz()
{
    char data;
    data = ' ';
    if(staticTrue)
    {
        fscanf (stdin, "%c", &data);
    }
    if(staticTrue)
    {
        char result = data + 1;
        printHexCharLine(result);
    }
}
```

The SARD dataset, however, requires a different approach. This dataset is not immediately usable and had to be manually processed to make it suitable. Each test case (see Listing 4.2) within the SARD dataset includes, among other things, comments describing the CWE, as well as CWE-related information embedded in function and variable names. Additionally, the test cases consist of compilable text files, unlike MegaVul, which already uses a JSON format. By using Python scripts and GitHub Copilot, the SARD dataset was modified so that CWE information is no longer inferable from the code (Listing 4.3).

Both datasets are ultimately converted into the same JSON format:

Listing 5.4: final JSON output

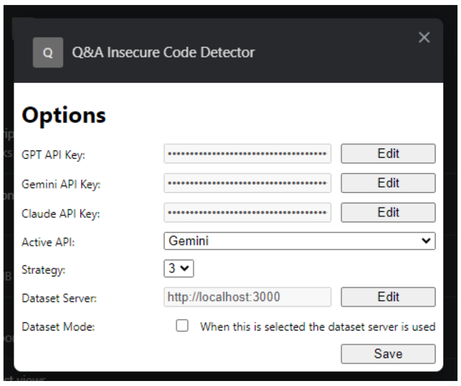
```
[
    {
        "data": {
            "cwe_ids": [],
        },
        "language": "lang-cpp",
        "code": "",
        "result": {}
    }
]
```

- ‘cwe_ids’ contains the vulnerabilities present as CWE numbers.
- ‘code holds’ the vulnerable code itself.
- ‘result’ will store the JSON output of the LLM.

However, the two datasets include different additional information as part of the ‘data’ field in the JSON format (see Appendix .5).

### 5.2.4 General Functionality

Through the popup menu and the options menu, users can switch between ‘website mode’ and ‘dataset mode’. Additionally, the active LLM API can be selected via both menus. Only the LLM API for which an API key is available, can be chosen. API keys can only be entered or modified through the options menu. Furthermore, this menu allows users to select the strategy and adjust the address of the dataset server (see Figure 5.3).



Q&A Insecure Code Detector

Options

GPT API Key: •••••••••••••••••••••••••••••••••••• Edit
Gemini API Key: •••••••••••••••••••••••••••••••••••• Edit
Claude API Key: •••••••••••••••••••••••••••••••••••• Edit
Active API: Gemini
Strategy: 3
Dataset Server: http://localhost:3000 Edit
Dataset Mode: ☐ When this is selected the dataset server is used

Save

Figure 5.3: option menu

### 5.2.5 Operation in Production (Website Mode)

When ‘dataset mode’ is not activated via the menus, the system will operate in ‘website mode’. In that case the system, when browsing to stackoverflow.com, will use the "webToCode" component to extract all code snippets from the website. A code snippet can be identified by its placement within the following HTML tags: <pre><code></code></pre>.

For each code snippet, the system will analyze whether the snippet contains C/C++ code and whether it includes insecure code constructs. This analysis is performed based on the selected prompt strategy and the chosen LLM API (see Figure 5.3). Once all information is gathered, it is processed and displayed on the webpage. Strategy 4 is specifically designed for this scenario.

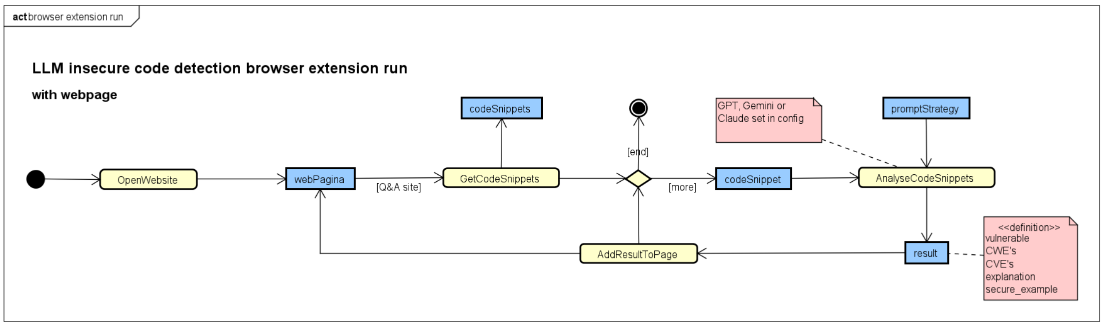

act browser extension run
LLM insecure code detection browser extension run
with webpage
OpenWebsite
webPagina
[Q&A site]
codeSnippets
GetCodeSnippets
[end]
[more]
codeSnippet
GPT, Gemini or Claude set in config
promptStrategy
AnalyseCodeSnippets
AddResultToPage
result
<<definition>> vulnerable CWE's CVE's explanation secure_example

Figure 5.4: website mode

As a result, the website indicates for each code snippet whether it is secure or not. If the LLM does not identify any vulnerabilities, an icon is displayed in the top-right corner of the code block indicating that no vulnerabilities were found. If a vulnerability is detected, a red button labeled "warning" with a warning triangle icon is shown. Clicking this button opens a dialog window containing detailed information about the vulnerabilities, among other details (see Figure 6.8).

### 5.2.6 Operation During Experiment (Dataset Mode)

When 'dataset mode' is selected via the menu, the chosen LLM API and strategy do not matter. In this mode, the system will automatically switch between all APIs and the first 3 strategies. The current implementation requires manual code modification to switch between the MegaVul and SARD datasets. In a future version, this could be made dynamically adjustable via the options menu. Figure 5.5 illustrates the flow for one LLM with one strategy. As shown in the JSON output and the diagram, the final data consists of pre-known information, such as the vulnerabilities present, along with the findings of the LLM.

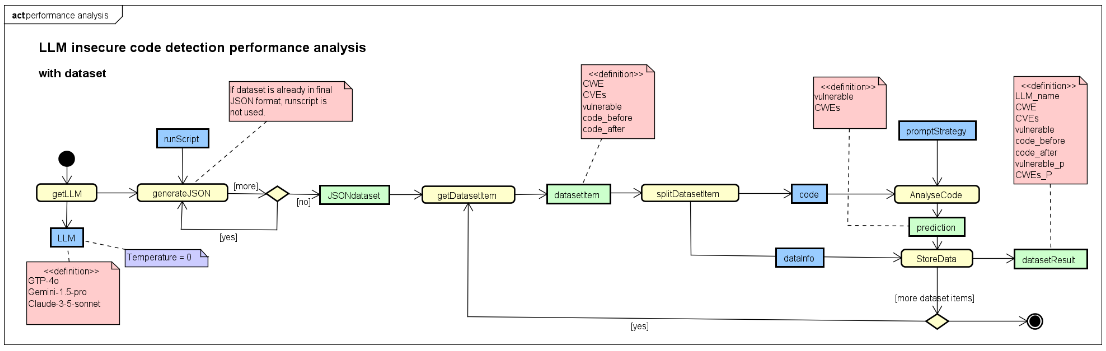

act performance analysis
LLM insecure code detection performance analysis
with dataset
If dataset is already in final JSON format, runscript is not used.
runScript
getLLM
generateJSON
[more]
[no]
[yes]
JSONdataset
getDatasetItem
<<definition>> CWE CVEs vulnerable code_before code_after
datasetItem
splitDatasetItem
code
<<definition>> vulnerable CWEs
promptStrategy
AnalyseCode
prediction
<<definition>> LLM_name CWE CVEs vulnerable code_before code_after vulnerable_p CWEs_P
dataInfo
StoreData
datasetResult
[more dataset items]
[yes]
LLM
Temperature = 0
<<definition>> GTP-4o Gemini-1.5-pro Claude-3-5-sonnet

Figure 5.5: dataset mode

# Chapter 6

# Evaluation

## 6.1 Experiment Setup

An experiment was conducted to evaluate how well the selected LLMs (Gemini, GPT, and Claude) perform in the context of unsafe code detection in C++ and to assess their usability for this task. Additionally, the experiment aimed to determine how the phrasing of the question influences the results. Finally, the study evaluated how this new LLM-based method compares to traditional static code analysis tools.

Prior to the experiment, the usability of various datasets was assessed. It became clear that the Megavul dataset was less suitable for this experiment because it contains large code fragments with numerous vulnerabilities. This is due to the fact that the code originates from repositories such as GitHub. Using this dataset would result in an extensive list of existing and detected CWEs, which would not be practical for this experiment. Therefore, the SARD dataset was chosen instead. This dataset consists of small code snippets specifically designed to represent a single CWE.

In Figure 6.1, the entire experiment is visualized. It begins at step 1, where a dataset of 1000 items is retrieved. This dataset consists of 100 unique samples for each of 10 different vulnerabilities. The final list of CWEs used in the experiment includes: 190, 401, 415, 416, 476, 590, 775, 762, 773, and 789. This list closely aligns with the initially selected CWEs, with minor adjustments made because not every pre-selected CWE has 100 SARD testcases. The dataset of 1000 samples was tested using strategies 1, 2, and 3 on Gemini, GPT-4, and Claude. This resulted in a total of 9,000 tests, which were analyzed and compared. This process of analyzing starts at step 2 in Figure 6.1. An answer is considered correct (True Positive, TP) if the CWE represented by the test case is found by the LLM, and it is also considered correct if the LLM provides multiple CWEs and the correct one is among them. Experiments were also conducted with imposing a limitation by specifying in the strategy that the answer should contain no more than 1, 3, or 5 different CWEs. This was especially necessary with the MegaVul dataset, as it yielded a large list of CWEs. With the SARD dataset, this proved unnecessary, as the LLMs based on this dataset generally did not provide more than 3 CWEs and often not more then 1 or 2.

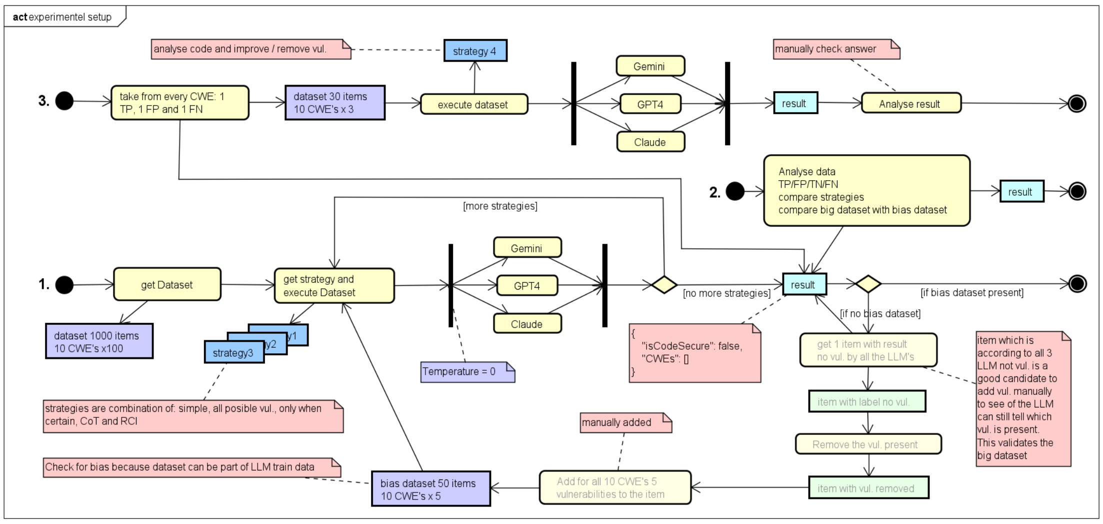

act experimentel setup
analyse code and improve / remove vul.
strategy 4
3.
take from every CWE: 1 TP, 1 FP and 1 FN
dataset 30 items 10 CWE's x 3
execute dataset
Gemini
GPT4
Claude
manually check answer
result
Analyse result
2.
Analyse data
TP/FP/TN/FN
compare strategies
compare big dataset with bias dataset
result
[more strategies]
1.
get Dataset
get strategy and execute Dataset
Gemini
GPT4
Claude
[no more strategies]
result
[if bias dataset present]
[if no bias dataset]
dataset 1000 items 10 CWE's x100
strategy1
strategy2
strategy3
Temperature = 0
{ "isCodeSecure": false, "CWEs": [] }
get 1 item with result no vul. by all the LLM's
item which is according to all 3 LLM not vul. is a good candidate to add vul. manually to see of the LLM can still tell which vul. is present. This validates the big dataset
item with label no vul.
strategies are combination of: simple, all posible vul., only when certain, CoT and RCI
manually added
Remove the vul. present
Check for bias because dataset can be part of LLM train data
bias dataset 50 items 10 CWE's x 5
Add for all 10 CWE's 5 vulnerabilities to the item
item with vul. removed

Figure 6.1: experiment setup

The lower part of Figure 6.1 represents a process that was not carried out due to time constraints. In this planned phase, 3 code snippets would have been manually created for each CWE. The purpose of this additional step was to address a potential bias, as the SARD dataset used in the experiment may have been part of the training process of the LLMs. By creating a subset of manually constructed samples, it would have been possible to verify whether the detection rate (DT) remains consistent. This aspect is now part of potential future work.

Since the LLMs' responses consist only of a list of CWE numbers, it is not possible to determine in detail what went wrong when a response is incorrect. To address this, a new dataset was generated using a Python script. For each CWE, this dataset includes a "True Positive" (TP), "False Positive" (FP), and "False Negative" (FN) variant. This process begins at step 3 in Figure 6.1. The resulting dataset of 30 samples was processed using strategy 4 (detailed information prompt) across all three LLMs. This produced a total of 90 items, which were manually evaluated.

## 6.2 LLM Settings

To ensure the reproducibility of the experiment, which is a crucial aspect of scientific research, several LLM settings were carefully selected. Specifically, fixed model versions were used instead of the “latest” versions. This approach ensures that the same model version can be selected when the experiment is repeated. However, it is important to note that the duration these versions remain available online, depends on the LLM providers. The following model versions were used during the experiment:

- gemini-1.5-pro-002
- gpt-4o-2024-08-06
- claude-3-5-sonnet-20241022

Another key setting for reproducibility is the temperature parameter of the LLM. The temperature controls the randomness of the output: lower values produce more predictable responses, while higher values generate greater variability. For all three models, the temperature was set to 0, ensuring deterministic and consistent results.

Additionally, the maximum number of output tokens was tailored to the requirements of the different strategies. For Strategies 1, 2, and 3, the output token limit was set to 50, forcing concise responses, which aligns with the objectives of these strategies. For Strategy 4, the token limit was increased to 8192 which allow an answer as comprehensive as possible. This limit of 8192 tokens corresponds to the maximum output capacity of the Claude model.

## 6.3 Evaluation of CWE detection

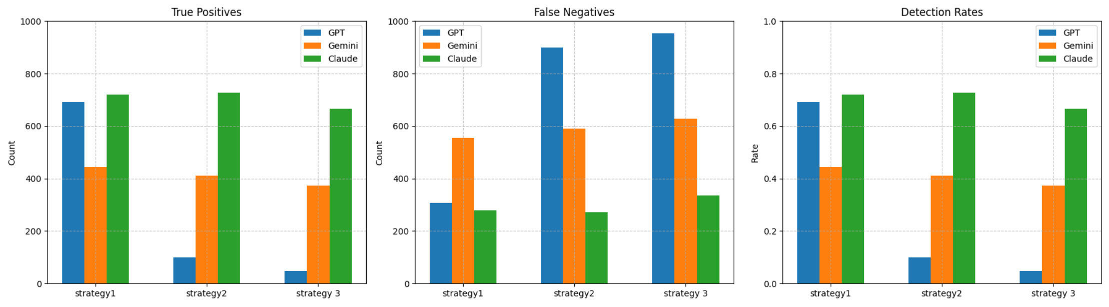


Figure 6.2: strategy comparision

The initial phase of the analysis focuses on evaluating the performance of the different strategies employed. For each strategy, the number of vulnerabilities correctly identified (True Positives, TP) and those missed (False Negatives, FN) is visualized on Figure 6.2. Notably, the simplest strategy (strategy 1), characterized by minimal instructions, generally outperforms strategies with more extensive instructions (strategies 2 and 3). In summary, the results indicate the following:

- Claude consistently outperforms other models across all strategies, demonstrating the highest detection ratios.

- Strategy 1 exhibits the highest average detection ratio (0.619), followed by Strategy 2 (0.412) and Strategy 3 (0.362).
- Claude achieves the highest detection ratio within Strategy 1 (0.727), correctly identifying 727 out of 1000 samples.
- The manner in which the prompt is formulated has the most pronounced impact on the performance of GPT.

Given that strategy 1 clearly yields the best and most consistent results, it will be the subject of further analysis in this section. The complete analysis for all strategies is available in Appendix .6.

An examination of the TP and FN rates per CWE reveals that certain CWEs are detected more effectively than others (see Figure 6.3). For instance, CWE 773 and 775 are rarely detected. While Claude exhibits a higher detection rate for these CWEs compared to the other two models, it also presents a higher FP rate, which may account for this observation. In contrast, CWE 401, 416, and 476 are detected in nearly all instances. Analyzing cases where a FN was recorded for CWE 415, Gemini frequently identified CWE 416 instead. CWE-415 (Double Free) refers to the incorrect deallocation of the same memory block multiple times, whereas CWE-416 (Use After Free) related to the utilization of memory after it has been deallocated.

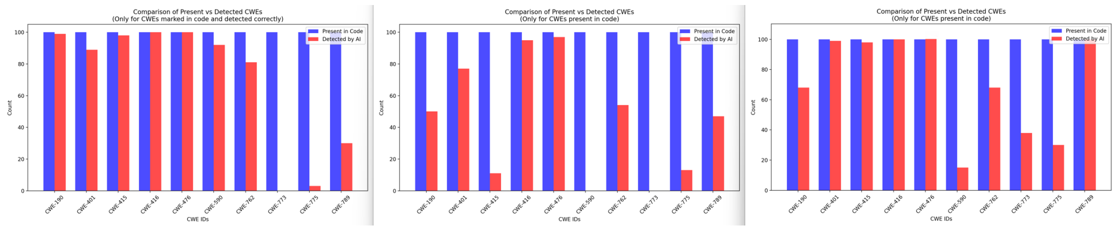


Figure 6.3: TP and FN comparision
From left to right: GPT, Gemini, Claude

The False Positive (FP) rate was also measured (see Figure 6.4), although its utility in drawing definitive conclusions is limited. This limitation arises because a detected CWE may indeed be present within the code snippet, even if it was not the intended target for that specific sample in the evaluation dataset. Nevertheless, the FP rate remains valuable for comparative analysis of the performance between the different models. By taking into account that the measured FP rate represents the most pessimistic scenario, it can be compared with the FP rates of traditional static analysis tools. CWE 415 and CWE 416 exhibit the highest FP rates, but this is likely attributable to their similarity in nature, often leading to their co-detection. Excluding these two CWEs, the FP rate across the selected CWEs is relatively low. Considering the overall balance between TP, FN, and FP rates, GPT demonstrates the most favorable performance.

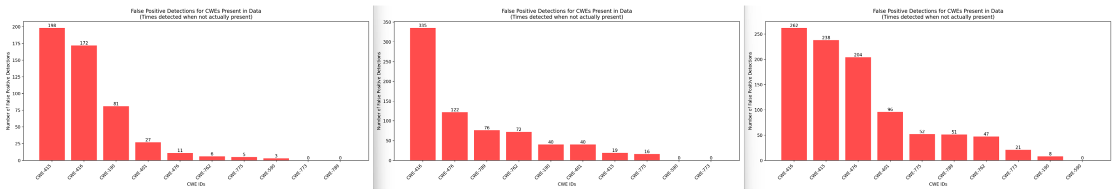


Figure 6.4: FP comparision
From left to right: GPT, Gemini, Claude

Several studies have investigated the True Positive (TP), False Negative (FN), False Positive (FP), and Detection Rate (DR) of various traditional Static Code Analysis Tools. The reliance on varying datasets, CWEs, and tools across these studies complicates benchmarking them against each other. To facilitate a meaningful comparison without retesting the static analysis tools using the dataset and CWEs from this research, studies that also focused on C++ and utilized similar CWEs were selected. Performance metrics from the tools identified as best-performing in two of those studies, one from 2017 (Arusoaie et al. [2017]) and one from 2024 (Li et al. [2024b]), were collected to provide a basis for comparison:

- True Positives: 456
- False Negatives: 544
- Detection Rate: 0.456
- False Positives: 113,33

Comparing these figures with the best-performing LLM in this research, it can be observed that the LLM-based method, achieving a detection rate of 0.727 (from strategy 1, Claude), significantly outperforms the traditional approach, which has a detection rate of 0.456.

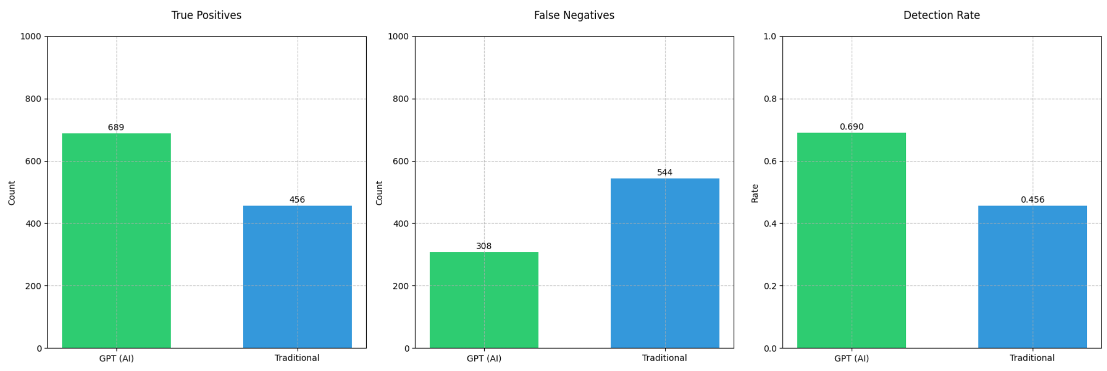


Figure 6.5: TP/FN - AI vs traditional tools

Although the FP rate is less conclusive, as previously discussed, it can still be compared with traditional methods to demonstrate that the LLM-based approach generates fewer FPs (see Figure 6.5).

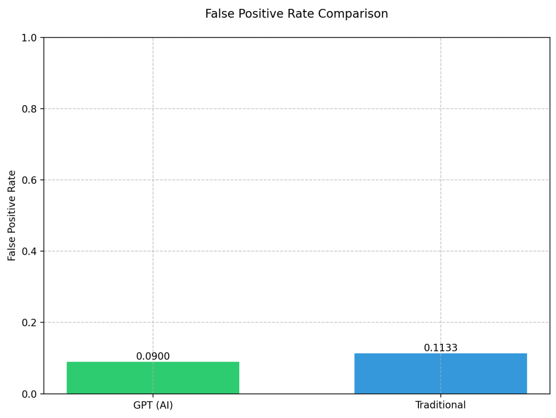


Figure 6.6: FP - AI vs traditional tools

### 6.3.1 Evaluation of Detailed Information

When a TP, FN, and FP are presented for a given CWE using strategy 4, detailed information is obtained. In the following tables, 'actual_cwes' represents the CWE that is actually present in the code, 'predicted_cwes' denotes the CWEs that were detected in the previous experiment with 1000 samples, and 'Vulnerabilities Found' indicates the result of the new assessment based on strategy 4.

The tables presented are programmatically generated by a Python script, utilizing the 'responseGPT.json' file located within the dataset_result/strategy4 directory. This file contains references to specific SARD test cases, including detailed vulnerability information. Analysis of the TP cases reveals that the provided motivations ('Vulnerabilities Found') are consistent with the vulnerability details documented in the corresponding SARD test cases. In the FN cases, the LLM, employing strategy 4, frequently still produces the correct answer. This is observed for CWE-190, CWE-415, and CWE-476. As for the FP cases, while a detected CWE may be present in some instances, it does not correspond to the CWE that the test case was intended to evaluate. However, there are also instances where the LLM should not have detected certain CWEs. For example, the FP for CWE 190 was detected because of the following issue: 'line 54 - Potential integer overflow when calculating memory allocation size.' The related code on line 54 is as follows: `memset(\&service, 0, sizeof(service));`. While it is true that incorrect usage of memset, such as `memset(buffer,0, len * sizeof(int));`, can indeed lead to CWE-190 (Integer Overflow), this is not the case here, and thus CWE-190 is indeed an incorrect detection.

For the CWEs not included in the tables below, no sets of TP, FN, and FP were available.

| Type | TP | FN | FP |
|---|---|---|---|
| Actual CWEs | CWE-190 | CWE-190 | CWE-789 |
| Predicted CWEs | CWE-190 | CWE-20, CWE-120, CWE-676 | CWE-190 |
| Vulnerabilities Found | [line 10] - Potential integer overflow when incrementing a char variable (CWE-190) | [line 66] - Potential integer overflow when incrementing data (CWE-190) | [line 54] - Potential integer overflow when calculating memory allocation size (CWE-190) |

Table 6.1: detailed CWE-190

| Type | TP | FN | FP |
|---|---|---|---|
| Actual CWEs | CWE-401 | CWE-401 | CWE-476 |
| Predicted CWEs | CWE-401, CWE-120 | CWE-252, CWE-476, CWE-758 | CWE-401 |
| Vulnerabilities Found | [line 20] - Memory leak due to overwriting the pointer without freeing the previously allocated memory (CWE-401) | (none) | [line 12-21] - Memory leak due to not freeing allocated memory (CWE-401) |

Table 6.2: detailed CWE-401

| Type | TP | FN | FP |
|---|---|---|---|
| Actual CWEs | CWE-415 | CWE-415 | CWE-401 |
| Predicted CWEs | CWE-415, CWE-416 | CWE-416 | CWE-415 |
| Vulnerabilities Found | [lines 10-12, 28-30] - Double Free Vulnerability (CWE-415) | [line 38] - Double Free: The pointer 'data' is freed twice, once in 'ppoeidn3' and again in 'ppoeidn0'. (CWE-415) | (none) |

Table 6.3: detailed CWE-415

| Type | TP | FN | FP |
|---|---|---|---|
| Actual CWEs | CWE-476 | CWE-476 | CWE-190 |
| Predicted CWEs | CWE-476 | CWE-457 | CWE-476 |
| Vulnerabilities Found | [line 7] - Null pointer dereference (CWE-476) | [line 19] - Null pointer dereference when accessing data[0] (CWE-476) | (none) |

Table 6.4: detailed CWE-476

| Type | TP | FN | FP |
|---|---|---|---|
| **Actual CWEs** | CWE-775 | CWE-775 | CWE-773 |
| **Predicted CWEs** | CWE-404, CWE-775, CWE-20 | CWE-676, CWE-404 | CWE-775 |
| **Vulnerabilities Found** | [line 27] - Potential resource leak due to not closing the file descriptor (CWE-775) | (none) | [Line 29, 36] - Potential resource leak due to improper file handling (CWE-775) |

Table 6.5: detailed CWE-775

Another observation to be made regarding the experiment with strategy 4 is the significantly increased response time of the LLMs in comparison to strategies 1, 2, and 3. The current response time ranges from 1 to 3 seconds, contingent upon the specific model and the code snippet being analyzed. As previously mentioned, this latency does not impact the website's loading time, as the process is executed asynchronously. Nevertheless, this does result in a delay in the availability of detection information. The magnitude of this delay will be further amplified with an increasing number of code snippets on the website.

All source test cases (input dataset) and results (output JSON) can be found in the provided project folder, which originates from the GitHub project page. Appendix 2 shows and explains the folder structure of this project folder.

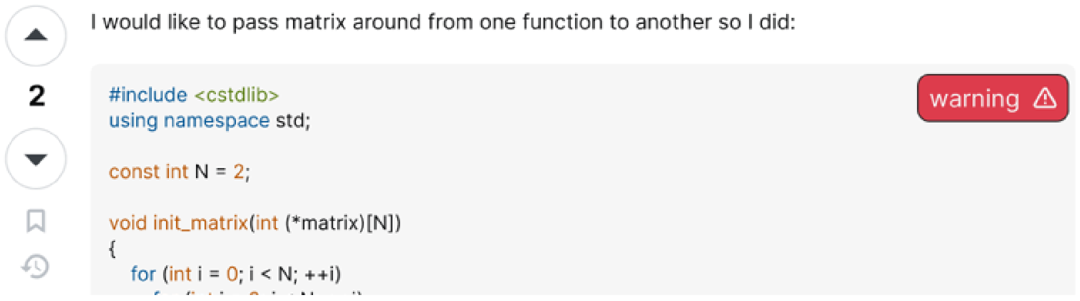
## 6.4 Realized Requirements from the SRS

At the time of writing, nearly all requirements from the Software Requirements Specification (SRS) have been implemented. The following requirement remains outstanding:

U2: As a user, I want to access detailed vulnerability information directly through a clear UI element, without needing to perform manual investigation.

Currently, the detailed information is printed to the console. In the period between the submission of the thesis and the final presentation (thesis defense), this functionality will be implemented. A high-fidelity design has already been created (see Figure 6.7 and 6.8). Furthermore, in addition to the MUST requirements, the following COULD/SHOULD requirements have been implemented:

FR3.2: As a user, I want to get a more secure alternative with a code example proposed.
FR4.2: As a user, I want to have the possibility to switch between the LLM providers.
FR5.5: As a user, I want to set up credentials regarding the used LLM provider API.

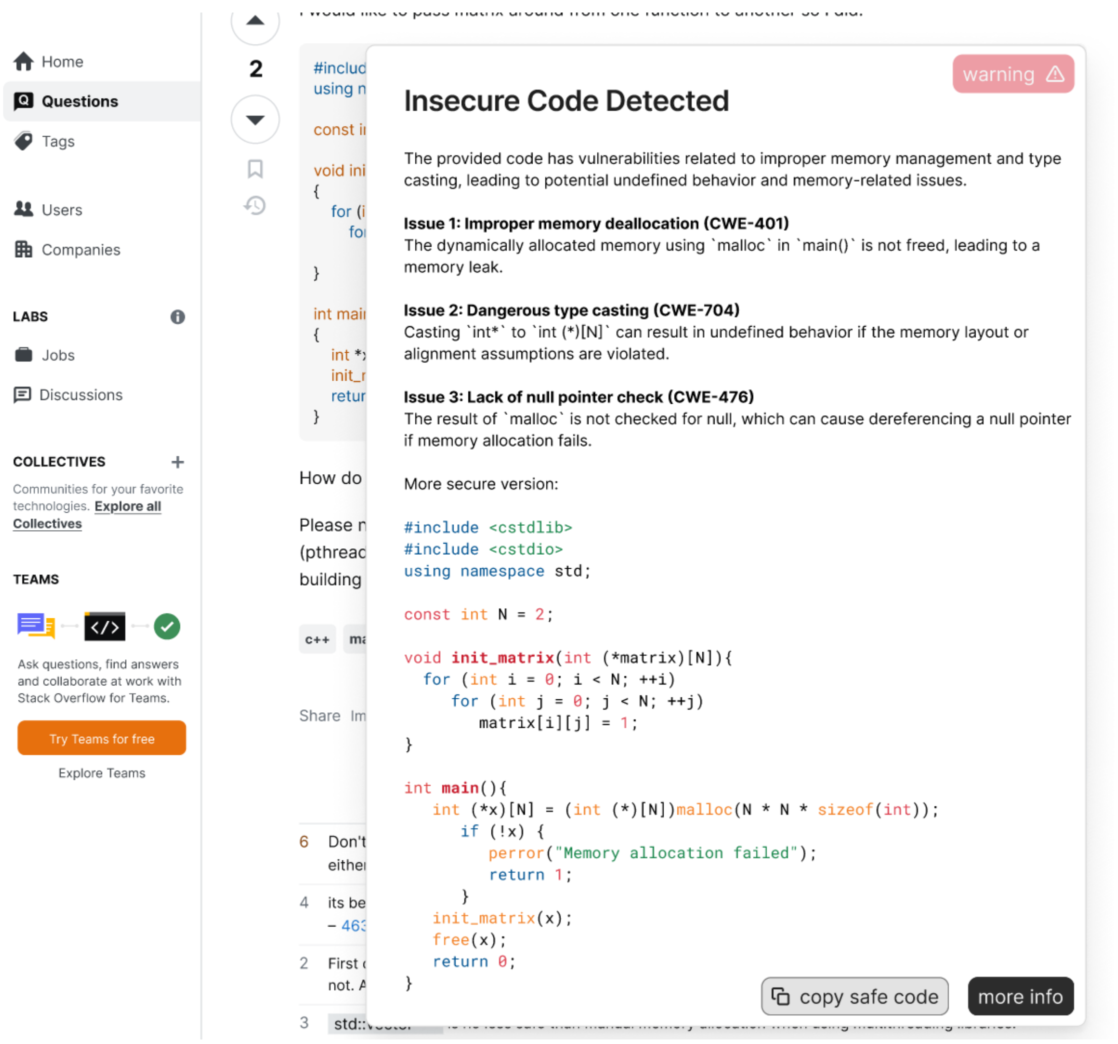

I would like to pass matrix around from one function to another so I did:

```
#include <cstdlib>
using namespace std;

const int N = 2;

void init_matrix(int (*matrix)[N])
{
    for (int i = 0; i < N; ++i)
```

Figure 6.7: warning before the button is clicked


Home
Questions
Tags
Users
Companies
LABS
Jobs
Discussions
COLLECTIVES
Communities for your favorite technologies. Explore all Collectives
TEAMS
Ask questions, find answers and collaborate at work with Stack Overflow for Teams.
Try Teams for free
Explore Teams
warning
copy safe code
more info

## Insecure Code Detected

The provided code has vulnerabilities related to improper memory management and type casting, leading to potential undefined behavior and memory-related issues.

**Issue 1: Improper memory deallocation (CWE-401)**
The dynamically allocated memory using `malloc` in `main()` is not freed, leading to a memory leak.

**Issue 2: Dangerous type casting (CWE-704)**
Casting `int*` to `int (*)[N]` can result in undefined behavior if the memory layout or alignment assumptions are violated.

**Issue 3: Lack of null pointer check (CWE-476)**
The result of `malloc` is not checked for null, which can cause dereferencing a null pointer if memory allocation fails.

More secure version:

```
#include <cstdlib>
#include <cstdio>
using namespace std;

const int N = 2;

void init_matrix(int (*matrix)[N]){
  for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; ++j)
        matrix[i][j] = 1;
}

int main(){
    int (*x)[N] = (int (*)[N])malloc(N * N * sizeof(int));
        if (!x) {
            perror("Memory allocation failed");
            return 1;
        }
    init_matrix(x);
    free(x);
    return 0;
}
```

Figure 6.8: detail info dialog after the button is clicked

# Chapter 7

# Conclusion and Discussion

Based on the evaluation results, it can be concluded that the main research question has been answered. The question was: "How can a web browser for students be enhanced so that unsafe code snippets on Q&A sites are automatically enriched with relevant information about the identified vulnerability using generative AI?" The conclusion will now be structured based on the sub-questions formulated to arrive at the answer to the main question. The detailed information underlying the key points summarized below can be found in the chapters: Literature Review, Design and Development, and Evaluation.

**1. What requirements must such a tool meet to be usable for students?**

This sub-question was addressed during Phase 1 (Identify Problem and Motivate) and Phase 2 (Define Objectives of a Solution) of the DSR process, resulting in the Software Requirements Specification (SRS). The answer to this sub-question can be found in that document. Key requirements identified for students include:

1. The tool must detect the most common C++ errors made by students.
2. It should be user-friendly, with no learning curve required.
3. Detection should occur close to the source without causing additional delays in website loading.

**2. What information regarding unsafe code is relevant?**

This question was primarily answered through a literature review and an expert interview, conducted during Phases 2 and 3 (Design and Development) of the DSR process. Key findings include:

1. Information should be tailored to the user's proficiency level; for instance, vulnerability details for first-year students should differ from those for fourth-year or graduate programmers.

2. Literature and the interview highlighted that students often encounter issues with classic memory problems, improper use of pointers, buffer overflows, use-after-free errors, and boundary checking.

3. Relevant information should not only identify the problem, but also suggest solutions and ideally provide improved alternatives.

4. CWE references were found to be more relevant for cybersecurity management courses. However, they were necessary for this research to facilitate automated experiments.

**3. How can a code snippet be identified on a Q&A website?**

This sub-question was mainly addressed during Phase 3 (Design and Development) and iteratively refined in Phases 4 (Demonstration) and 5 (Evaluation). The answer was formulated based on a literature review, available product analysis, and prototyping. The findings include:

1. Recommended architecture for browser extensions was identified through the literature review and subsequently followed.

2. On Stack Overflow, code snippets are surrounded by the following HTML tags: <code><pre> </pre></code>. This was confirmed through product analysis.

3. The UML designs and prototype resulting from this research demonstrate the implementation of code snippet identification.

**4. What method can best be used to automatically evaluate the code examples using generative AI?**

The literature review on AI revealed many models, but only a few are suitable for detecting vulnerabilities:

1. Models predating 2022 were not suitable for this task.

2. Among post-2022 models, Claude, Gemini, and GPT were identified as the best performers, with GPT deemed particularly effective.

3. Studies have shown that the phrasing of a prompt significantly influences the quality of an AI's response.

Experiments detailed in the Evaluation chapter revealed that GPT-4o combined with Strategy 1 provided the best results.

**5. How can the relevant information be displayed to the student?**

This question was answered based on literature review, available product analysis, and prototyping, resulting in:

1. A Software Requirements Specification (SRS) detailing requirements for displaying information.
2. A high-fidelity wireframe outlining the design.
3. A prototype implementing these findings.

**Conclusions Regarding the Main Research Question**

Given that Q&A sites are web pages accessed through a browser and it is generally preferable to address a problem as close to the source as possible, it was concluded during the preliminary research that a browser extension would be the most suitable approach. The literature study revealed that most static code analysis tools perform code evaluation within the IDE, in version control on the server, or as standalone applications on the programmer's PC. Therefore, a tool that operates directly at the source, on the website, is a welcome addition to the landscape of static code analysis tools.

Based on the literature review and the results of this research, the browser extension mechanism demonstrates the following advantages:

1. Direct detection at the source (Q&A website).
2. Minimal impact on loading times due to asynchronous communication.
3. Browser extensions are relatively easy to develop due to the technologies used and good documentation.
4. A developed extension can be easily deployed to other browsers.
5. Website domains on which the extension should operate are easy to add.

However, the mechanism also exhibits some disadvantages:

1. The extension mechanism does not have a good reputation, as many extensions in the store contain vulnerabilities themselves and are not maintained.
2. Extensions must be distributed through the store, or the browser must be set to 'developer mode.'

The primary motivation for employing LLMs for code assessment was the simplicity with which any piece of code can be inspected using a single API. Previously, without LLMs, evaluating code snippets in the browser would not have been possible without setting up a server running multiple applications. This research confirms that with a single API, and even with a single specific API call, any piece of code in any programming language can be inspected.

The experiment, conducted with 9000 samples, provides a reliable overview and demonstrates that an LLM, as hypothesized, is indeed capable of providing meaningful and relevant information about vulnerabilities in code. Despite the inherent risk of hallucination in LLMs, which may lead to incorrect answers, the detection remains useful. The experiment shows that an LLM detects approximately 70% of vulnerabilities. This is significantly higher than the 45% achieved by traditional tools, according to previously conducted studies. Considering the possibility of hallucination, a warning that the answer may be incorrect is desirable. However, on the other hand, traditional tools do not always provide reliable answers either. The same studies show that the false positive rate (FP) for traditional tools is relatively high, with an average of 11%. Based on the results of this research, it can be stated that with 9%, this is certainly lower. It is likely, as reasoned in the evaluation, that this percentage is even lower, but this is difficult to measure automatically. This would require manual verification of a large number of samples to determine if a detected vulnerability is present in the code, even if it is not the vulnerability the test case was designed for. The manual verification as part of the evaluation, in any case, indicates through sampling that sometimes this is the case.

Furthermore, the evaluation reveals that the phrasing of the prompt significantly influences the results. A simple prompt with minimal instructions (strategy 1) generally yields better results than an extensive instruction based on RoC and RCI (strategies 2 and 3). Claude and Gemini seem to be less affected by this than GPT. However, requesting detailed information (strategy 4), which is also an extensive instruction based on RoC and RCI, appears to produce better results than strategy 2 and 3. A conclusion that could be drawn from this is that allowing an LLM to reason in the background in combination with enforcing a short answer yields a worse result than making the reasoning part of the answer. In other words, there is a negative effect based on the relationship between prompt complexity and a simplified assigned output. However, this requires further investigation.

The use of Large Language Models (LLMs) for vulnerability detection is a rapidly developing field, and this thesis contributes to this area by focusing on a unique approach: integrating LLMs within web browsers to identify insecure code snippets on Q&A platforms like StackOverflow. Aligning with other research in the field, this study confirms the critical importance of prompt engineering, reinforces the potential of LLMs in vulnerability detection, and acknowledges their inherent limitations.

Zooming in on the assessment of individual CWEs, it appears that the specific vulnerability being evaluated makes a considerable difference. For instance, CWE 773 and 775 are almost never detected, while 415 and 416 are almost always detected. Because the metrics for traditional tools derive from prior studies using slightly different sets of CWEs, a future study could compare traditional and AI approaches using a consistent set of CWEs.

Regarding the tool's usability for students, it can be concluded that it is relatively useful, especially compared to traditional tools. However, this needs further practical testing and evaluation with students. Another factor that may play a role and requires further investigation is the cost associated with using the LLM API. Although the costs are relatively low and continue to decrease, a student may not be willing to spend money on this.

In summary, the following conclusions can be drawn:

1. Vulnerability detection using LLMs is viable but has limitations. This confirms the findings of related studies.
2. The performance of LLMs surpasses that of traditional tools.
3. LLMs exhibit a lower FP rate than traditional tools, but the exact difference is difficult to determine.
4. Currently, there are (still) costs associated with the APIs.
5. The phrasing of the prompt influences the result.
6. Simple instructions appear to work better than elaborate instructions.
7. Having an LLM reason in the background seems less effective than making the reasoning part of the output.
8. There is a significant difference in detection rate (DR) between different CWEs.
9. There is a risk of hallucination, leading to potentially incorrect answers.
10. The false positive rate (FP) is difficult to determine automatically.
11. An LLM API call based on strategy 4 takes a significant amount of time (1 to 3 seconds).
12. This study has a unique approach by integrating LLMs within web browsers
13. This study confirms vulnerability detection findings of related work

Based on the research conducted, several questions remain unanswered, and new questions have arisen. Therefore, further research could be conducted on the following topics (future work):

1. Experimenting with programming languages other than C/C++.

2. Experimenting with self-created code snippets containing vulnerabilities to address the bias related to a learned dataset.
3. Investigating whether anything can be said automatically about the reliability of the answer.
4. A comparative experiment between traditional tools and AI based on the same set of CWEs.
5. Investigating why the different prompt versions have so much influence on GPT.
6. Investigating how open-source LLMs perform in the context of vulnerability detection.
7. Investigating whether LLM fine-tuning and RAG can have a positive effect on the result.
8. Conducting an evaluation with end-users and focusing more on didactics.

# Chapter 8

# Reflection

In the beginning, I started without a clear understanding of what the final deliverable should be, and my work was primarily organized with a focus on software development. This approach originated from my current role as an HBO (Higher Professional Education) computer science lecturer and my previous roles as a software developer. However, it didn't feel right, so I consulted the study guide, specifically the section outlining the requirements for the master's thesis. I then adjusted my planning accordingly, aligning it more closely with the scientific nature of the graduation project.

I also noticed that TNO places little emphasis on software development and associated practices, such as requirements specification, designs using UML, continuous integration, unit testing, and so on. TNO is primarily focused on short-term software projects designed to support research. My graduation project was conducted using the Design Science Research (DSR) methodology, and this approach worked well. The phases and sub-questions were well-aligned. However, sub-questions 2 and 3 both focused on the design phase of DSR and turned out to be relatively small in scope. I considered merging them but ultimately decided not to after consultation during a supervision-meeting. Looking back, I might as well have merged them, as there wasn't enough time to delve into details such as how the information could best be presented to the student.

The emphasis on students in the main research question was a deliberate choice, given my background in education and my interest in addressing the didactics of the subject. In hindsight, there was insufficient time to fully explore the didactics. The topic is touched upon in the literature review and is a limited part of the SRS, but the primary focus was on the technical aspects and conducting experiments with LLMs. I believe this was the right choice, considering this is an engineering study rather than an educational one.

Early in the project, the scope was narrowed by selecting one programming language and focusing on 10 vulnerabilities. This was the right decision, but during the experiment, the Megavul dataset turned out to be unsuitable. As a result, I switched to the SARD dataset. However, this dataset didn't contain 100 samples for every CWE, requiring some CWEs to be replaced. Initially, one week was allocated for the experiments, but due to setbacks, this phase ultimately took three weeks. Despite the challenges, the results of the experiments were usable and provided a solid comparative study.

During the literature review, using the website connectedpapers.com proved to be a good choice, as it visually mapped out all related research. The detailed step-by-step approach was also very useful. Unfortunately, the final two chapters could not be reviewed by TNO for content due to time constraints.

The planning was well-monitored and organized using Trello, but as previously mentioned, the project ran over time toward the end. Overall, it has been an interesting graduation project with valuable results.

# Bibliography

Y. Acar, C. Stransky, D. Wermke, C. Weir, M. L. Mazurek, and S. Fahl. Developers need support, too: A survey of security advice for software developers. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 22–26. IEEE, 2017.

P. Akshay Dev and K. Jevitha. Stride based analysis of the chrome browser extensions api. In *Proceedings of the 5th International Conference on Frontiers in Intelligent Computing: Theory and Applications: FICTA 2016, Volume 2*, pages 169–178. Springer, 2017.

M. Almansoori, J. Lam, E. Fang, A. G. Soosai Raj, and R. Chatterjee. Towards finding the missing pieces to teach secure programming skills to students. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 973–979, 2023.

E. A. AlOmar, S. A. AlOmar, and M. W. Mkaouer. On the use of static analysis to engage students with software quality improvement: An experience with pmd. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 179–191. IEEE, 2023.

A. Arusoaie, S. Ciobâca, V. Craciun, D. Gavrilut, and D. Lucanu. A comparison of open-source static analysis tools for vulnerability detection in c/c++ code. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 161–168, 2017. doi: 10.1109/SYNASC.2017.00035.

H. Assal and S. Chiasson. 'think secure from the beginning' a survey with software developers. In *Proceedings of the 2019 CHI conference on human factors in computing systems*, pages 1–13, 2019.

Astah. tool for uml design, 2024. URL https://astah.net/. Accessed: 2025-01-01.

W. Bai, O. Akgul, and M. L. Mazurek. A qualitative investigation of insecure code propagation from online forums. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 34–48. IEEE, 2019.

M. Bishop, M. Dark, L. Futcher, J. Van Niekerk, I. Ngambeki, S. Bose, and M. Zhu. Learning principles and the secure programming clinic. In *Information Security Education. Education* *in Proactive Information Security: 12th IFIP WG 11.8 World Conference, WISE 12, Lisbon, Portugal, June 25–27, 2019, Proceedings 12*, pages 16–29. Springer, 2019.

I. A. Buckley, J. Zalewski, and P. J. Clarke. Introducing a cybersecurity mindset into software engineering undergraduate courses. *International Journal of Advanced Computer Science And Applications*, 9(6), 2018.

M. Buehler. Detecting insecure code with llms, 2024. URL https://towardsdatascience.com/detecting-insecure-code-with-llms-8b8ad923dd98. Accessed: 2025-01-01.

W. Charoenwet, P. Thongtanunam, V.-T. Pham, and C. Treude. An empirical study of static analysis tools for secure code review. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 691–703, 2024.

S. Chung, L. Hansel, Y. Bai, E. Moore, C. Taylor, M. Crosby, R. Heller, V. Popovsky, and B. Endicott-Popovsky. What approaches work best for teaching secure coding practices. In *Proceedings of the 2014 HUIC Education and STEM Conference*, 2014.

D. B. Cruz, J. R. Almeida, and J. L. Oliveira. Open source solutions for vulnerability assessment: A comparative analysis. *IEEE Access*, 11:100234–100255, 2023.

cvedetails. cve details, vulnerabilities by types, 2024. URL cvedetails.com/vulnerabilities-by-types.php. Accessed: 2025-01-01.

K. Dozono, T. E. Gasiba, and A. Stocco. Large language models for secure code assessment: A multi-language empirical study. *arXiv preprint arXiv:2408.06428*, 2024a.

K. Dozono, T. E. Gasiba, and A. Stocco. Large language models for secure code assessment: A multi-language empirical study. *arXiv preprint arXiv:2408.06428*, 2024b.

X. Du, G. Zheng, K. Wang, J. Feng, W. Deng, M. Liu, and Y. Lou. Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag. 2024.

A. Fatima, S. Bibi, and R. Hanif. Comparative study on static code analysis tools for c/c++. In *2018 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, pages 465–469. IEEE, 2018.

Figma. tool for making wireframes, 2024. URL https://www.figma.com/. Accessed: 2025-01-01.

K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek. Benefits and drawbacks of adopting a secure programming language: Rust as a case study. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, pages 597–616, 2021.

Google. Chrome extensions reference, 2024. URL https://developer.chrome.com/docs/extensions/reference. Accessed: 2025-01-01.

M. Grover, J. Cummings, and T. Janicki. Moving beyond coding: why secure coding should be implemented. *Journal of Information Systems Applied Research*, 9(1):38, 2016.

C. Guß. How to prove that your c/c++ code is safe and secure. 2020.

A. Haleem, M. Javaid, and R. P. Singh. An era of chatgpt as a significant futuristic support tool: A study on features, abilities, and challenges. *BenchCouncil transactions on benchmarks, standards and evaluations*, 2(4):100089, 2022.

J. He, M. Vero, G. Krasnopolska, and M. Vechev. Instruction tuning for secure code generation. *arXiv preprint arXiv:2402.09497*, 2024.

helloleads. widely used extensions, 2020. URL https://www.helloleads.io/. Accessed: 2025-01-01.

W. Hou and Z. Ji. Comparing large language models and human programmers for generating programming code. 2024.

M. Howard, D. LeBlanc, and J. Viega. *Deadly Sins of Software Security-Programming Flaws and How to Fix Them, 2010*. 2011.

S. Hsu, M. Tran, and A. Fass. What is in the chrome web store? In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 785–798, 2024.

L. Hüther, K. Sohr, B. J. Berger, H. Rothe, and S. Edelkamp. Machine learning for sast: A lightweight and adaptable approach. In *European Symposium on Research in Computer Security*, pages 85–104. Springer, 2023.

Y. Jiang, Y. Zhang, X. Su, C. Treude, and T. Wang. Stagedvulbert - multi-granular vulnerability detection with a novel pre-trained code model. 2024.

B. Jin, H. Li, and Y. Zou. Impact of extensions on browser performance: An empirical study on google chrome. *arXiv preprint arXiv:2404.06827*, 2024.

julius. Ai baszed to for data analysis, 2024. URL https://julius.ai. Accessed: 2025-01-01.

A. Kaur and R. Nayyar. A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code. *Procedia Computer Science*, 171:2023–2029, 2020.

M. M. Kholoosi, M. A. Babar, and R. Croft. A qualitative study on using chatgpt for software security: Perception vs. practicality. 2024.

G. Kim, P. Baldi, and S. McAleer. Language models can solve computer tasks. 2024.

J. Lam, E. Fang, M. Almansoori, R. Chatterjee, and A. G. Soosai Raj. Identifying gaps in the secure programming knowledge and skills of students. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education-Volume 1*, pages 703–709, 2022.

K. Li, S. Chen, L. Fan, R. Feng, H. Liu, C. Liu, Y. Liu, and Y. Chen. Comparison and evaluation on static application security testing (sast) tools for java. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 921–933, 2023.

Z. Li, S. Dutta, and M. Naik. Llm-assisted static analysis for detecting security vulnerabilities. 2024a.

Z. Li, Z. Liu, W. K. Wong, P. Ma, and S. Wang. Evaluating c/c++ vulnerability detectability of query-based static application security testing tools. *IEEE Transactions on Dependable and Secure Computing*, 21(5):4600–4618, 2024b. doi: 10.1109/TDSC.2024.3354789.

Z. Li, Z. Liu, W. K. Wong, P. Ma, and S. Wang. Evaluating c/c++ vulnerability detectability of query-based static application security testing tools. *IEEE Transactions on Dependable and Secure Computing*, 2024c.

V. Majdinasab, M. J. Bishop, S. Rasheed, A. Moradidakhel, A. Tahir, and F. Khomh. Assessing the security of github copilot's generated code-a targeted replication study. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 435–444. IEEE, 2024.

Q. Mao, Z. Li, X. Hu, K. Liu, X. Xia, and J. Sun. Towards effectively detecting and explaining vulnerabilities using large language models. 2024.

V. Mdunyelwa, L. Futcher, and J. van Niekerk. Towards a framework for teaching secure coding practices to programming students. In *European Conference on e-Learning*, pages 592–XX. Academic Conferences International Limited, 2021.

P. Mehta. *Creating google chrome extensions*. Springer, 2016.

Microsoft. The stride threat model, 2009. URL https://learn.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20). Accessed: 2025-01-01.

mitre. Common attack pattern enumerations and classifications, 2025a. URL https://cwe.mitre.org/. Accessed: 2025-01-01.

mitre. Att&ck matrix for enterprise, 2025b. URL https://attack.mitre.org. Accessed: 2025-01-01.

mitre. Cve security vulnerability database, 2025c. URL https://cve.mitre.org/. Accessed: 2025-01-01.

mitre. Cwe security weaknesses database, 2025d. URL https://cwe.mitre.org/. Accessed: 2025-01-01.

C. Ni, L. Shen, X. Yang, Y. Zhu, and S. Wang. Megavul: A c/c++ vulnerability dataset with comprehensive code representations. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 738–742. IEEE, 2024.

Nist. Vulnerability datasets, 2024a. URL https://samate.nist.gov/SARD/test-suites/112. Accessed: 2025-01-01.

Nist. Secure software development framework (ssdf), 2024b. URL https://csrc.nist.gov/projects/ssdf. Accessed: 2025-01-01.

D. Noever. Can large language models find and fix vulnerable software? *arXiv preprint arXiv:2308.10345*, 2023.

Owasp. Application security verification standard, 2024. URL https://github.com/OWASP/ASVS. Accessed: 2025-01-01.

Owasp. owasp top 10 vulnerabilities, 2025. URL https://owasp.org/Top10/. Accessed: 2025-01-01.

K. Peffers, T. Tuunanen, M. Rothenberger, and S. Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24: 45–77, 01 2007.

E. Pelofske, V. Urias, and L. M. Liebrock. Automated software vulnerability static code analysis using generative pre-trained transformer models. 2024.

C. P. Pfleeger. *Security in computing (5th Edition)*. Prentice-Hall, Inc., USA, 2002. ISBN 0137989431.

M. Plch. *Secure coding in modern C++*. PhD thesis, Masaryk University, Faculty of Informatics, 2018.

J. Pöial. Challenges of teaching programming in stackoverflow era. In *Educating Engineers for Future Industrial Revolutions: Proceedings of the 23rd International Conference on Interactive Collaborative Learning (ICL2020), Volume 1 23*, pages 703–710. Springer, 2021.

T. Potluri, Y. Shi, H. Shahriar, D. Lo, R. Parizi, H. Chi, and K. Qian. Secure software development in google colab. In *2023 IEEE World AI IoT Congress (AIIoT)*, pages 0398–0402. IEEE, 2023.

T. Richardson and C. N. Thies. *Secure software design*. Jones & Bartlett Publishers, 2013.

R. C. Seacord. *Secure Coding in C and C++*. Addison-Wesley, 2013.

M. Sedova. Comparing educational approaches to secure programming: Tool vs.{TA}. In *Thirteenth Symposium on Usable Privacy and Security ({SOUPS} 2017)*, 2017.

Z. Sheng, F. Wu, X. Zuo, C. Li, Y. Qiao, and L. Hang. Lprotector: An llm-driven vulnerability detection. 2024a.

Z. Sheng, F. Wu, X. Zuo, C. Li, Y. Qiao, and L. Hang. Navrepair: Node-type aware c and cpp code vulnerability repair. 2024b.

A.-M. Stanciu and H. Ciocârlie. Analyzing code security: Approaches and tools for effective review and analysis. In *2023 International Conference on Electrical, Computer and Energy Technologies (ICECET)*, pages 1–6. IEEE, 2023.

D. Stuttard. The web application hacker's handbook: Finding and exploiting security flaws. *Wade AIcorn*, 2011.

S. Sultana, S. Afreen, and N. U. Eisty. Code vulnerability detection: A comparative analysis of emerging large language models. 2024.

L. Tal. Secure sdlc | secure software development life cycle | snyk, 2024. URL https://snyk.io/articles/secure-sdlc/. Accessed: 2025-01-01.

K. Tamberg and H. Bahsi. Harnessing large language models for software vulnerability detection: A comprehensive benchmarking study. 2024.

R. Team. Top 5 c++ security risks, 2022. URL https://snyk.io/blog/top-5-c-security-risks/. Accessed: 2025-01-01.

A. Torbunova, A. Ashraf, and I. Porres. A systematic mapping study on teaching of security concepts in programming courses. In *2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 288–295. IEEE, 2024.

Trello. tool for project management, 2024. URL https://trello.com. Accessed: 2025-01-01.

A. Tuladhar, D. Lende, J. Ligatti, and X. Ou. An analysis of the role of situated learning in starting a security culture in a software company. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, pages 617–632, 2021.

J. vom Brocke, A. Hevner, and A. Maedche. *Introduction to Design Science Research*, pages 1–13. Springer International Publishing, Cham, 2020. ISBN 978-3-030-46781-4. doi: 10.1007/978-3-030-46781-4_1. URL https://doi.org/10.1007/978-3-030-46781-4_1.

H. Washizaki. Swebok v4.0 the newest edition of the internationally acclaimed software engineering body of knowledge. 2024. URL https://www.computer.org/education/bodies-of-knowledge/software-engineering/v4.

R. Widyasari, D. Lo, and L. Liao. Beyond chatgpt - enhancing software quality assurance tasks with diverse llms and validation techniques. 2024.

Y. YaunPan. interactive application security testing yuan 2019. 2019.

X. Yuan, L. Yang, B. Jones, H. Yu, and B.-T. Chu. Secure software engineering education: Knowledge area, curriculum and resources. *Journal of Cybersecurity Education, Research and Practice*, 2016(1):3, 2016.

H. Zhang, S. Wang, H. Li, T.-H. Chen, and A. E. Hassan. A study of c/c++ code weaknesses on stack overflow. *IEEE Transactions on Software Engineering*, 48(7):2359–2375, 2021.

S. Zhao. Github copilot now has a better ai model and new capabilities. *The GitHub Blog*, 2023.

# Appendices

## .1 Software Requirements Specification

# Software Requirement Specification

insecure code detection tool

TNO Public ⟩ TNO 2024 R00000
24 September 2024

ICT, Strategy & Policy
www.tno.nl
+31 88 866 00 00
info@tno.nl

TNO 2024 R00000 – 24 September 2024

# Software Requirement Specification

insecure code detection tool

| | |
|---|---|
| Author(s) | R. Holleman |
| Classification report | TNO Public |
| Title | TNO Public |
| Report text | TNO Public |
| Number of pages | 12 (excl. front and back cover) |
| Number of appendices | 0 |

# Contents

# Revision History

| Name | Date | Changes | Version |
|---|---|---|---|
| R. Holleman | 24-09-2024 | Initial version | V0.1 (concept) |
| R. Holleman | 05-01-2025 | Final version | V1.0 |

# 1 Introduction

## 1.1 Purpose

It is well known that computer science students regularly solve their programming challenges by googling the answer. They often end up on Q&A sites like StackOverflow and Reddit. These are websites where a question about the problem has usually already been asked and answered by members of the community. If the question has not yet been asked and answered, students frequently ask it themselves, and it is usually answered within a few hours. Students also obtain code examples from repositories like GitHub and SourceForge. What all these sources have in common is that the website owner is not responsible for the quality of the code, and therefore, unsafe code examples and snippets are often found among them. This problem is most prevalent on Q&A sites, but it certainly also applies to repositories.

The fact that students use unsafe code snippets is problematic because this teaches them unsafe coding practices, and later in the professional field, they may also produce unsafe code that is then put into production. Additionally, research shows that even among professional programmers, writing secure code remains a continuous challenge, and they do not always check functionally working code examples for unsafe code.

To address this problem, research will be conducted on how a web browser for student can be extended, so that unsafe code snippets on multiple website can automatically be enriched, with relevant information about the identified vulnerability, using generative AI. The artifact to be developed, for which this document serves as the requirement specification, will primarily focus on answering the central research question and secondarily on the implementation of the software in production. In other words, the artifact will initially be a Proof of Concept (PoC) but could eventually be used in education by students. This should help them to identify which code snippets on websites like StackOverflow are safe or unsafe. Additionally, it can help experienced programmers avoid using unsafe snippets from such sources as well. By making the artifact publicly available, it can contribute to a general better understanding of secure coding and, ultimately, to safer software.

## 1.2 Product scope

**Generative AI**
Since 2022, generative AI has been growing rapidly and offering many new possibilities. Before that time, developing a browser tool like the one this SRS is focuses on, capable of detecting unsafe code snippets, regardless of the programming language, would be complex or needs a lot of different external API's. For instance a server would have to be set up for each programming language, running static code analysis software that provides an API. Now, with AI, it is possible to use a single API that makes all of this possible and because of that the artefact would be based on that technology.

**One programming language**
Because the focus is on answering the research question rather than putting the tool into production, one specific programming language has been chosen. The tool will therefore initially be developed and evaluated for this language. The following factors were considered in choosing this programming language:

1. Which languages are most commonly used in general
2. Which languages HBO (university of applied sciences) students learn during their bachelor's program
3. Which languages are best supported by LLMs (Large Language Models)
4. For which languages relevant datasets are available

From the analysis, C++ and Python emerged as the best candidates. Ultimately, C++ is chosen because it is the more vulnerable to unsafe code of the two languages.

**Dutch bachelor students**
The target audience is limited to Dutch bachelor's students, so with regard to point 2, when selecting a programming language, there is no need to consider other countries and universities. The artefact will initially only be evaluated within such an HBO institution.

**One target website**
The artifact will initially be developed only for the Q&A website StackOverflow.

**One target web browser**
The artifact will initially be developed only for one browser, which has yet to be determined.

**No LLM customization**
The AI models will not be modified. Therefore, no fine-tuning or Retrieval-Augmented Generation (RAG) will be applied.

**15 software weaknesses**
The artifact will initially be tested and evaluated with 15 software weakness. Therefore, the focus during development will be on these vulnerabilities.

# 2 Overall Description

## 2.1 Product Perspective

As previously described, the tool must recognize code snippets on Q&A sites like 'StackOverflow', and then provide detailed information to indicate whether it contains unsafe code constructs. The most straightforward solution for extending a web browser is through a browser extension. This way, a browser can be extended without having to modify the browser's source code.

The browser extension must be able to extract code snippets from the relevant website, and then offer this code, combined with a formulated query, to an LLM via an API. This process of submitting and formulating queries could be implemented through various strategies. These possible strategies will need to be researched and created later during the design phase.

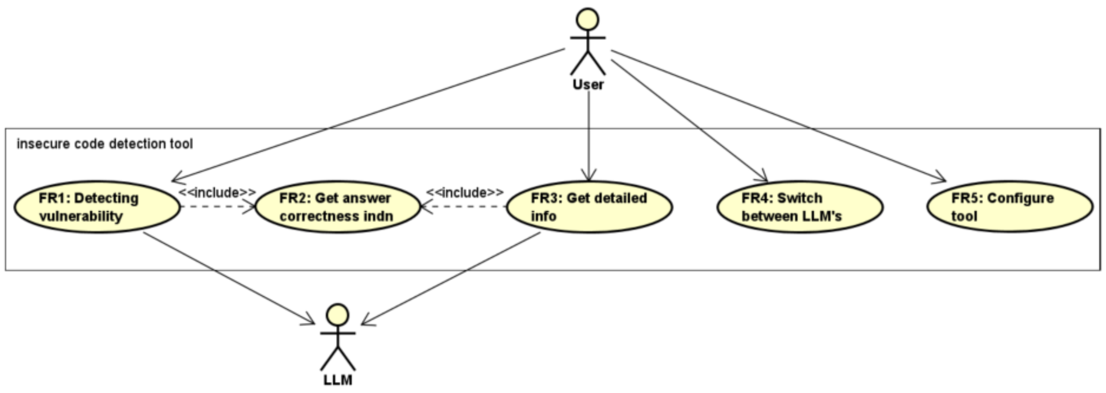
The artifact is ultimately intended to compare and evaluate different LLMs and strategies in the context of the chosen programming language and vulnerabilities. It is also meant to evaluate feasibility and usability. For usability purposes, it is important that the information regarding the vulnerabilities is didactically sound and with that usable in the higher education industry.

## 2.2 Product Functions

In Figure 1, the high-level requirements are modelled. These are explained here and further detailed and prioritized in Chapter 4: System features. All these high-level requirements are necessary for the purpose of the research. To what extent and which sub-requirements are needed is addressed in Chapter 4. Here and further in Chapter 4, the requirements are prioritized according to the MoSCoW method (source).

**FR1: Detecting vulnerability (Must-Have)**
Detect vulnerabilities in code snippets on a Q&A website, using an LLM (API), and show a visual indication. This is the main functionality of the tool and is therefore a 'Must-Have'.

**FR2: Get answer correctness indication (Should-Have)**
Since the information comes from an LLM and it is more likely to provide an incorrect answer than no answer at all (source), it is desirable for the tool to indicate the reliability of the answer. Whether and how this can be realized will be determined through later research. Due to this uncertainty, this functionality is a 'Should-Have'. It is further desirable that this functionality is automatically triggered during the detection process (FR1) and does not need to be manually executed by the users.



User
insecure code detection tool
FR1: Detecting vulnerability
<<include>>
FR2: Get answer correctness indn
<<include>>
FR3: Get detailed info
FR4: Switch between LLM's
FR5: Configure tool
LLM

Figure 1: High Level Functional req.

**FR3: Get detailed information (Must-Have)**
In addition to the visual indication of whether the code snippet is vulnerable, the user should have the option to receive additional information about the vulnerability. Examples of this include: an explanation of the vulnerability, an example of a safer alternative, and references to other sources such as the CWE database, etc.

FR4: Switch between LLM's **(Must-Have)**
It must be possible for the user to switch between different LLMs. This is important for the end user because they want to be able to choose which LLM they use. One reason why, is that different LLM providers come with different costs. Additionally, in the context of the research, it is necessary to switch between LLMs because the different LLMs need to be compared and evaluated. This makes this functionality a 'Must-Have'.

**PF5: Configure tool (Must-Have)**
The tool must be configurable by the user. For example, in the context of the research, it should be possible to switch between different strategies, which can be set through configuration. Another example is that the response timeout should be adjustable. Although the functionality itself is a 'Must-Have', it is highly dependent on the available time. For an initial version, configuration via a configuration file, instead of through a GUI, might be sufficient.

## 2.3 Users and stakeholder

In the context of the primary and secondary objectives of the tool, various stakeholders and users are playing a role. For the research, the group of 'researchers' is particularly important, with lecturers and students added to validate and test the tool. The other stakeholders and roles will play a part later, but their interests must be taken into account from the start.

**Researchers**
Researchers, and in particular the researcher involved in the study for which the 'insecure code detection tool' is being developed, can use the tool to investigate the usability of LLMs in relation to 'insecure code detection.'

**Universities of applied science**
As HBO-I, the overarching platform for HBO institutions, states in their proposition, they aim to deliver highly skilled IT professionals to the job market (source). In doing so, they contribute to the public good. Secure programming skills are an important part of being an IT professional, and institutions therefore have an interest in teaching this skill as effectively as possible. The effectiveness of education is a key requirement in this regard. The time a student has to become an IT professional is limited, and teaching hours costs an institution money, so it must be utilized as efficiently as possible. The 'insecure code detection tool' can help with this, and the institutions could play a role in promoting and implementing it into the curriculum.

**Lecturers**
Lecturers can use the tool as part of their lessons and promote it among their students. Additionally, they can use the tool themselves, as it is known that even graduate IT professionals often have limited knowledge of 'secure programming.' As part of the research for which the tool is primarily being developed, they can also be involved in validating and testing the tool. After all, they should be the knowledge carriers regarding 'secure programming.

**Students**
This is the most important stakeholder because the tool is ultimately intended for this target group. Whether they will use the tool directly depends on its usability. For them, it is particularly important that the information provided by the tool is accessible, comprehensible, and accurate. Within this group, there is a wide range of skill levels, as first-year students have little knowledge of IT concepts and lack a framework to connect them, while fourth-year students have developed much more of a framework and understanding of the concepts (source).

**Software developer (IT-professional)**
In addition to education, the tool can also be used by graduated software developers. This is also one of the roles that lecturers fulfill.

# 3 System features

For each 'product functionality' from chapter 2, several sub-requirements are formulated below. The priority of each sub-requirement is indicated using the 'MoSCoW' method (source). The 'must-haves' must be realized at a minimum for the research, and the 'should-haves' would ideally be realized. The 'could-haves' and 'would-haves' are desirable but are by definition outside the scope.

| **FR1: Detecting vulnerability** | **MoSCoW** |
|---|---|
| 1. As a user I want to extract a code snippet from the website StackOverflow. | Must |
| 2. As a user I want to extract a code snippet from the website reddit. | Could |
| 3. As a user I want to extract a code snippet from all possible websites. | Would |
| 4. As a user I want to know of a code snippet contains C/C++ code. | Must |
| 5. As a user I want to know of a code snippet contains Python code. | Could |
| 6. As a user I want to know of a code snippet contains Java code. | Would |
| 7. As a user I want to know of a code snippet contains C# code. | Would |
| 8. As a user I want to know of a code snippet contains JavaScript code. | Would |
| 9. As a user I want that the 10 most relevant C/C++ vulnerabilities from the CWE database are detected. | Must |
| 10. As a user I want that all the relevant C/C++ vulnerabilities from the CWE database are detected. | Could |
| 11. As a user I want to be informed about the extent of unsafe code instructions in the code snippet. | Must |
| 12. As a user I want to get specific information about framework vulnerabilities if a framework is involved. | Would |
| 13. As a researcher I want to be able to provide an offline code snippet. | Must |
| 14. As a researcher I want be able to create different strategies for getting the best possible answer. | Must |
| 15. As a researcher I want to execute the different strategies. | Must |
| 16. As a researcher I want to compare and evaluate the different strategies. | Must |
| 17. As a user, I want that the most common C++ errors made by students are detected | Must |

ICT, Strategy & Policy

Anna van Buerenplein 1
2595 DA Den Haag
www.tno.nl

TNO innovation for life

| FR2: Get answer correctness indication | | MoSCoW |
|---|---|---|
| 1. | As a user I want to get a warning that the provided answer may be incorrect. | Must |
| 2. | As a user I want to get, together with the ‘secure code indication’, the likelihood of correctness of the indication. | Could |
| 3. | As a researcher I want to get information about the likelihood that the right programming language is detected. | Should |
| 4. | As a researcher I want to get information about the correctness of the answer provided by a LLM. | Could |

| FR3: Get detailed information | | MoSCoW |
|---|---|---|
| 1. | As a user I want to get detailed information about the vulnerability that is found. | Must |
| 2. | As a user I want to a get a more secure alternative with a code example proposed. | Should |
| 3. | As a user I want to get additional information about the vulnerability from the CWE database. | Could |
| 4. | As a researcher I want to have a strategy to get detailed information. | Must |
| 5. | As a researcher I want to compare and evaluate the different strategies regarding the detailed information. | Must |

| FR4: Switch between LLM’s | | MoSCoW |
|---|---|---|
| 1. | As a user I want that at least 2 different LLM providers/models can be used. | Must |
| 2. | As a user I want to have the possibility to switch between the LLM providers. | Could |
| 3. | As a researcher I want that the different LLM’s can be compared and evaluated. | Must |

| FR5: Configure tool | | MoSCoW |
|---|---|---|
| 1. | As a user I want to configure which LLM is used. | Must |
| 2. | As a researcher I want to configure which strategy is used. | Must |
| 3. | As a researcher I want to configure which level of logging is applied. | Should |
| 4. | As a researcher I want to configure where the output is directed to (website or data file). | Should |
| 5. | As a user I want to setup credentials regarding the used LLM provider API. | Should |
| 6. | As a user I want to change the time-out regarding the response time from the LLM. | Could |
| 7. | As a user I want to configure the following settings: cost threshold,  target websites, target programming languages, included/excluded vulnerabilities, interface language, auto update code snippet, customize answer and update mode. | Would |

ICT, Strategy & Policy

Anna van Buerenplein 1
2595 DA  Den Haag
www.tno.nl

TNO innovation for life

# 4 Other Nonfunctional Requirement

The Nonfunctional requirements are categorized by the FURPS model (source), which is used to classify and categorize software requirements. It stands for Functionality (subject of chapter 4), Usability, Reliability, Performance, and Supportability. These categories help define the quality attributes of the tool.

| **U: Usability** | | **MoSCoW** |
|---|---|---|
| 1. | As a user I want to be able to see at a glance and based on one single icon, of the code snippet contains possible vulnerabilities. | Must |
| 1. | As a user I want that the detection occurs close to the source without causing additional | Must |
| 2. | As a user, I want to access detailed vulnerability information directly through a clear UI element, without needing to perform manual investigation. | Must |
| 3. | As a user I want to know how to configure the tool, without reading the manual. | Must |
| 4. | As a user I want that links to external sources, like the CWE reference is clickable. | Should |
| 5. | As a user I want that the cost of using the LLM are as low as possible. | Should |
| 6. | As a user I want that the tool is available in my own language. | Would |
| 7. | As a user, I want to access detailed vulnerability information directly through a clear UI element, without needing to perform manual investigation. | Must |

| **R: Reliability** | | **MoSCoW** |
|---|---|---|
| 1. | As a researcher I want have reporting and logging available. | Must |
| 2. | As a researcher, I want the key routines to be covered with unit tests. | Should |
| 3. | As a researcher, I want that the ‘must have’ requirements are covered with acceptance tests. | Should |
| 4. | As a user I want to be sure that I have the latest version of the tool. | Would |
| 5. | As a user I want have the most accurate and reliable information that is possible. How far this is possible and what the level of accuracy and reliability will be, is subject of the research and can’t be answered upfront. | Would |

ICT, Strategy & Policy

Anna van Buerenplein 1
2595 DA Den Haag
www.tno.nl

TNO innovation for life

| P: Performance | MoSCoW |
|---|---|
| 1. As a user I want that the additional time to load the website is not more than 1000 milliseconds per webpage | Must |

| S: Supportability | MoSCoW |
|---|---|
| 1. As a researcher I want to use the GPT-4o model API from OpenAI. | Must |
| 2. As a researcher I want to use the Gemini 1 model API from Google. | Must |
| 3. As a researcher I want to use the Claude-3 model API from Anthropic. | Should |
| 4. As a user I want to have the tool available for one browser. | Must |
| 5. As a user I want to have the tool available for all major internet browsers | Would |
| 6. As a user I want to have IDE integration support | Would |
| 7. As a user I want that the tool works with phones and tablets. | Would |
| 8. As a user I want that the tool is platform (OS) independent. | Would |
| 9. As a user I want to use the tool without changing the source code of the browser. | Must |
| 10. As a user I want to get information tailored on the level of my experience. | Could |

ICT, Strategy & Policy

Anna van Buerenplein 1
2595 DA  Den Haag
www.tno.nl

TNO innovation for life

## .2 Project Directory Overview

- **Directory: dataset_result**
  - **Directory: strategy1** = result of experiment with strategy 1
  - **Directory: strategy2** = result of experiment with strategy 2
  - **Directory: strategy3** = result of experiment with strategy 3
  - **Directory: strategy4** = result of experiment with strategy 4
  - **File: analyseData.py** = script to analyse CWE's
  - **File: analyseData2.py** = script to analyse CWE's
  - **File: compareStrategies.png** result of script compare_with_straditional.py
  - **File: compare_strategies.py** script to compare strategies
  - **File: compare_with_straditional.py** script to compare with traditional tool metrics
  - **File: compare_with_traditional.png** result of script compare_with_straditional.py
  - **File: demo.response.json** = only for test purposes
  - **File: demo.response2.json** = only for test purposes
  - **File: getOverall_FN.py** = script to get overaal FN
  - **File: get_CWE_FN.py** script to get FN for strategy
  - **File: get_CWE_FP.py** script to get FP for strategy
  - **File: get_top20_FP.py** script to get top 20 FP for strategy
  - **File: get_TP_FN.py** script to get TP and FN for strategy
  - **File: get_TP_FN_FP_items.py** script to get items for get_TP_FN_FP_tables.py
  - **File: get_TP_FN_FP_tables.py** = script to get items result in tables for strategy 4
  - **File: strategy_analysis.json**
- **Directory: dataset_run_scripts** = run scripts for MEgavul dataset
- **Directory: dataset_server** = webserver to provide datasets
  - **Directory: datasets** = SARD and Megavul datasets
  - **Directory: src** = the webserver
- **Directory: src** = extension source folder
  - **Directory: __tests__** = unittests
- Directory: public = HTML and manifest files for extention

## .3 prompt strategies

### .3.1 Stragety 1

Listing 1: strategy 1: simple and minimal instruction

```
Analyze the code provided after the label "code_to_analyse:" and determine if
it has any security weaknesses. If the code has weaknesses, identify all
applicable Common Weakness Enumerations (CWEs) with a maximum of 5 CWEs. Add
each CWE number to the "CWEs" array *without* the "CWE-" prefix.
Your response MUST be in valid JSON format only with no extra information,
characters or newlines. The JSON must have the following structure:

{
    "isCodeSecure": false,
    "CWEs": []
}

code_to_analyse:
```

Listing 2: strategy 2: Most chance to find CWE but detects more unwanted CWE’s

```
You are a brilliant software security expert. Analyze the code provided after
the label "code_to_analyse:" and determine if it has any security weaknesses.
Your response MUST be in valid JSON format with no extra information,
characters or newlines. The JSON should adhere to the following structure:

{
    "isCodeSecure": false,
    "CWEs": []
}

Follow these steps:

1. Carefully examine the code provided after "code_to_analyse:".
2. Determine if the code is secure (true) or not secure (false).
Set the "isCodeSecure" field accordingly.
3. If the code is NOT secure (isCodeSecure: false), identify all
applicable Common Weakness Enumerations (CWEs) with a maximum of 5
CWEs. Include also potential weaknesses.
4. Add each CWE number to the "CWEs" array *without* the "CWE-"
prefix. For example, if you identify CWE-79, add only ’79’ to the array.
```

```
 5. Double-check your analysis and ensure the JSON is valid and accurate.
 Correct any mistakes before finalizing your response.

 code_to_analyse:
```

Listing 3: strategy 3: Less unwanted CWE's detected but chance of missing one is bigger

```
You are a brilliant software security expert. Analyze the code provided
after the label "code_to_analyse:" and determine if it has any security
weaknesses. Your response MUST be in valid JSON format with no extra
information, characters or newlines. The JSON should adhere to the
following structure:

 {
     "isCodeSecure": false,
     "CWEs": []
 }

 Follow these steps:

 1. Carefully examine the code provided after "code_to_analyse:".
 2. Determine if the code is secure (true) or not secure (false). Set the
 "isCodeSecure" field accordingly.
 3. If the code is NOT secure (isCodeSecure: false), identify all applicable
 Common Weakness Enumerations (CWEs) with a maximum of 5 CWEs.
 4. Add each CWE number to the "CWEs" array *without* the "CWE-" prefix. For
 example, if you identify CWE-79, add only '79' to the array. Only include CWEs
 that are definitely present and represent a security bug. Do not include CWEs
 that might be present or are not certain.
 5. Double-check your analysis and ensure the JSON is valid and accurate.
 Correct any mistakes before finalizing your response.

 code_to_analyse:
```

Listing 4: strategy 4: detailed output with code improvements

```
You are a security expert specialized in analyzing C++ code for vulnerabilities.
Your task is to assess the provided C++ code and determine whether it is secure.
If you find any issues, identify them clearly, provide detailed explanations of
the vulnerabilities, and map them to relevant CWE (Common Weakness Enumeration)
identifiers.

Use the following process to ensure a thorough evaluation:
```

```
1) Initial Analysis:
1.1) Analyze the given C++ code for security vulnerabilities.
1.2) If the code is secure, explain why it is considered safe.
1.3) If vulnerabilities are present, explain each issue in detail with
    references to specific lines or sections of the code.

2) CWE Mapping:
For every identified issue, map it to a relevant CWE identifier.
Provide a brief explanation of the CWE and why it applies.

3) Suggestions for Improvement:
Suggest clear and actionable steps to fix each vulnerability.
Provide code snippets or examples when possible.

4) Recursive Improvement (RCI):
Re-analyze the code after addressing the vulnerabilities using your own
suggestions. Ensure that the revised code resolves all identified issues
and does not introduce new vulnerabilities. Repeat this process iteratively
until the code is secure or no further improvements can be made.

Guidelines:
Provide explanations in clear and concise language suitable for developers.
Prioritize identifying critical and high-severity issues first, followed by
lower-severity ones. Document your reasoning for all conclusions to ensure
transparency and reproducibility. Your response MUST be in valid JSON format
with no extra information, characters or newlines outside the json object.
Double-check your analysis and ensure the JSON is valid and accurate. Check
for any extra or missing characters or newlines.

Output Format:

{
    "Secure": false,
    "Explanation": "",
    "Vulnerabilities Found": {
        "Issue1": "[line number(s)] - Description (CWE-ID)",
        "Issue2": "[line number(s)] - Description (CWE-ID)"
},
"Improvement": {
    "Fix1": "Suggested Fix for Issue 1",
    "Fix2": "Suggested Fix for Issue 2"
},
"FinalCode": ""
```

```
}

    1) Security Assessment:
    Secure/Not Secure
    Explanation

    2) Vulnerabilities Found:
    Issue 1: Description (CWE-ID)
    Issue 2: Description (CWE-ID)
    ...

    3) Suggestions for Improvement:
    Suggested Fix for Issue 1
    Suggested Fix for Issue 2
    ...

    4) Final Secure Code:
    Provide the final secure version of the code after implementing fixes.
    Use this scheme to ensure that the analysis is thorough and iterative,
    making the code as secure as possible.

    Code for Analysis:
```

## .4 run scripts

Listing 5: Megavul run script

```
{
    "119": {
        "amount": 20
    },
    "120": {
        "amount": 20
    },
    "401": {
        "amount": 20
    },
    "772": {
        "amount": 20
    },
    "131": {
        "amount": 20
    },
    "476": {
        "amount": 20
    },
    "416": {
        "amount": 20
    },
    "415": {
        "amount": 20
    },
    "190": {
        "amount": 20
    },
    "787": {
        "amount": 20
    }
}
```

## .5 JSON output format

Listing 6: output of strategy 1  2 and 3

~~~
[
    {
        "data": {
            "cwe_ids": [
                "CWE-190"
            ],
            "filename": "000082110_1.txt"
        },
        "language": "lang-cpp",
        "code": "[source code with vulnerability]",
        "result": {
            "isCodeSecure": false,
            "CWEs": [
                190,
                20
            ]
        }
    },
    {
        "data": {
            "cwe_ids": [
                "CWE-401"
            ],
            "filename": "000042110_1.txt"
        },
        "language": "lang-cpp",
        "code": "[source code with vulnerability]",
        "result": {
            "isCodeSecure": false,
            "CWEs": [
                401
            ]
        }
    },
    ``` etc
]
~~~

Listing 7: output of strategy 4

~~~
[
    {
        "data": {
            "cwe_ids": [
                "CWE-416"
            ],
            "filename": "000102225_1.txt",
            "type": "TP",
            "cwe": "CWE-416",
            "actual_cwes": [
                "CWE-416"
            ],
            "predicted_cwes": [
                "CWE-416",
                "CWE-415"
            ]
        },
        "language": "lang-cpp",
        "code"",
        "result": {
            "Secure": false,
            "Explenation": "[explenation from LLM]",
            "Vulnerabilities Found": {
                "Issue1": "[line X] - explanation from LLM. (CWE-416)"
                ```etc
            },
            "Improvement": {
                "Fix1": "explanation from LLM."
                ```etc
            },
            "FinalCode": "improved code fro LLM"
        }
    },
    ```etc
]
~~~

## .6 Experiment results

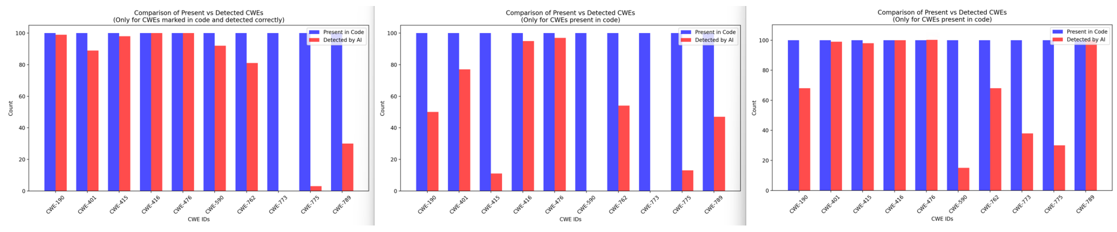


Figure 1: Strategy 1 - TP and FN comparision
From left to right: GPT, Gemini, Claude

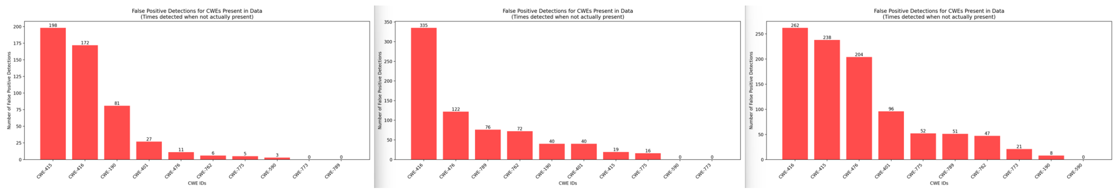


Figure 2: Strategy 1 - FP comparision
From left to right: GPT, Gemini, Claude

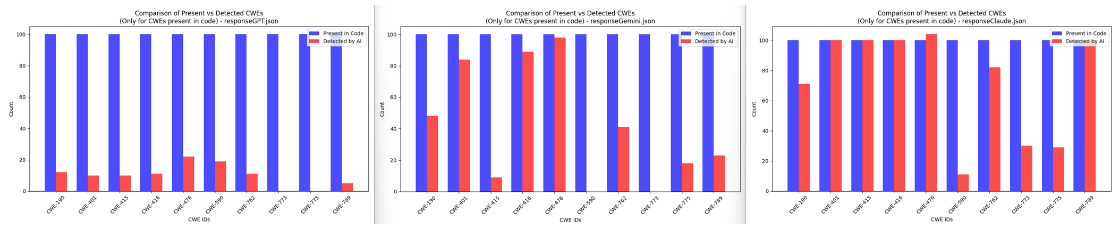


Figure 3: Strategy 2 - TP and FN comparision
From left to right: GPT, Gemini, Claude

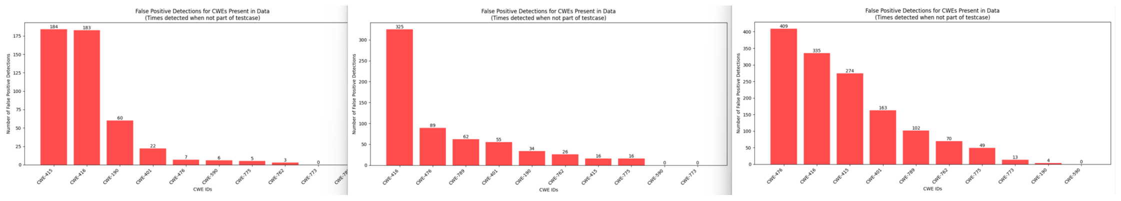


Figure 4: Strategy 2 - FP comparision
From left to right: GPT, Gemini, Claude

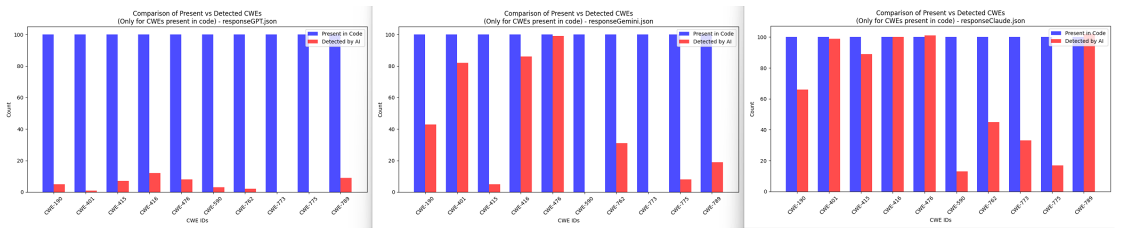


Figure 5: Strategy 3 - TP and FN comparision
From left to right: GPT, Gemini, Claude

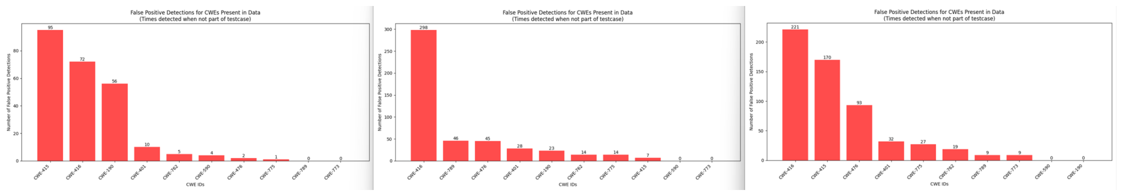


Figure 6: Strategy 3 - FP comparision
From left to right: GPT, Gemini, Claude

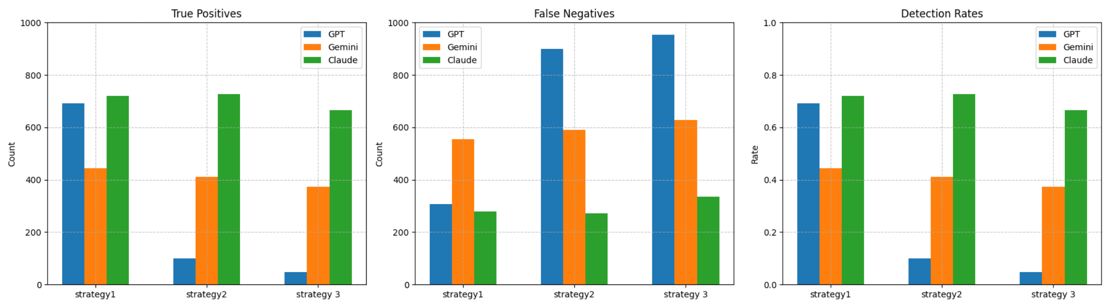


Figure 7: strategy comparision

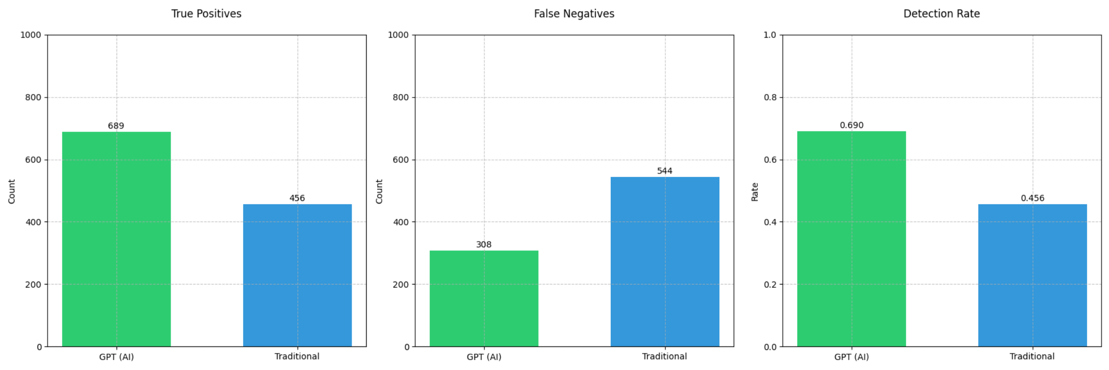


Figure 8: TP/FN - AI vs traditional tools

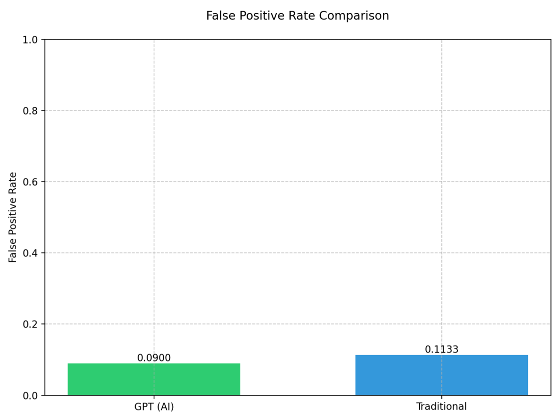


Figure 9: FP - AI vs traditional tools

## .7 Interview question / answers (in Dutch)

## Inleidende vragen (5 minuten, 5-10)

### 1. Wat beschouwt u als de belangrijkste aspecten van secure programming?

Het is belangrijk om in de gaten te houden dat ik onderwijs geeft in een wo opleiding. Op hbo niveau zouden de antwoorden anders zijn. Voor mij is het belangrijkste aspect dat de student inzicht krijgt in de fundamentele aspecten van cybersecurity, waarbij mijn focus vooral ligt om het computer systems perspectief. De studenten moeten leren begrijpen hoe het systeem werkt en waarom het fout kan gaan.

### 2. Wat is uw mening m.b.t. het algemene niveau van software developers m.b.t. secure programming en ziet u daar een verandering in de afgelopen decennia?

Dat kan ik moeilijk beoordelen, dat zou een betere vraag zijn voor mensen die in de praktijk werken. Onder onze studenten zie ik niet echt een verandering, maar zij krijgen vanaf het begin al low-level concepten.

## C++ uitdagingen in het onderwijs (15 minuten, 10-25)

### 3. Zijn er bepaalde veelvoorkomende fouten of misvattingen die studenten maken als het gaat om veilig programmeren in C++? Hoe kunt u deze het beste adresseren?

Ik laat de studenten primair oefenen in C. In zie daarbij vooral de klassieke memory errors, zoals buffer overflows en use-after-free. Studenten hebben vaak geen goed idee wat ze doen als ze met pointers werken.

### 4. Wat zijn binnen C/C++ de grootste uitdagingen omtrent secure programming?

Vooral de klassieke memory errors, zoals buffer overflows en use-after-free.

### 5. Op internet kom je nog veel oude stijl C++ (zoals b.v. RAW pointer) tegen, hoe moeten we hier in het onderwijs mee omgaan, aangezien we ze modern C++ willen aanleren?

Het is belangrijk studenten (op het wo in ieder geval) beide te leren. Raw pointers gebruiken is belangrijk om goed inzicht te krijgen in hoe systemen werken. Smart pointers zouden gebruikt moeten worden bij programmeren in de praktijk (of beter nog: veilige talen).

## Secure programming curriculum (45 minuten, 25-70)

### 6. Op welke manier zou secure programming volgens u het beste in het onderwijs passen en waarom?

Hoe wij het aanpakken past goed in een minor van de bachelor, waarbij de meer technisch aangelegde bachelorstudenten ervoor kiezen dit vak met relatief veel diepgang te volgen. Als het doel enkel is om de praktische vaardigheid aan te leren, dan is het passender dit te integreren in de reguliere programmeervakken (inleiding programmeren).

### 7. Welke onderwijsmethoden (didactische methodes) vind u het meest effectief voor het aanleren van secure C++ programming?

Daar heb ik niet goed zicht op, het vak zoals ik dit geef zie ik niet als programmeeronderwijs. De studenten kunnen al programmeren als ze beginnen.

## 8. Welke basis kennis (kennis vooraf) hebben studenten nodig om betreffende insecure code, het probleem en de oplossing te begrijpen?

Voor het vak zoals ik dat geef moeten de studenten vooral C of C++ kunnen programmeren, en goed begrip hebben van computerarchitectuur en besturingssystemen.

## 9. Wat moet een student weten m.b.t. memory management als het gaat om secure C++ programming?

Het vak zoals ik dat geef is vooral gericht op de meer fundamentele concepten, dus niet specifiek over C++ programming. Wat betreft memory management bespreek ik niet alleen hoe ze het moeten/kunnen doen (bijvoorbeeld RAII en smart pointers), maar vooral ook hoe het werkt (hoe is de geheugenlayout en hoe kan je die informatie gebruiken voor exploits).

## 10. Welke rol speelt theoretische kennis in combinatie met praktische toepassingen bij het leren van secure programming?

Theoretische kennis is cruciaal om de aard van de dreiging goed te begrijpen. Praktisch toepassen zonder theoretische kennis zal zeker helpen in de eenvoudige gevallen, maar het enkel gebruiken van programmeerregels geeft niet het inzicht om met complexere gevallen om te gaan of dreigingsanalyses te maken.

## 11. Welke informatie is met betrekking tot onveilige code voor studenten relevant?

Kennis van wat er mis kan gaan, hoe daar misbruik van gemaakt kan worden, en wat daartegen gedaan kan worden.

## 12. In hoeverre is informatie uit CVE/CWE databases en andere formele bronnen relevant en belangrijk voor studenten en zit daar, tussen verschillende leerjaren en niveaus (beginnend tot gevorderd), verschil in?

Deze zijn vooral relevant voor vakken die meer gericht zijn op management, maar mijn vak is meer technisch van aard.

## 13. Is een specifiek antwoord gericht op C++ of een algemeen en taal onafhankelijk antwoord beter?

De studenten oefenen met C omdat je daarin het meest ziet van memory management, maar de geleerde concepten zijn fundamenteel en onafhankelijk van de taal. In mijn colleges en het tentamen toon ik een variatie aan programmeertalen, ook afhankelijk van het meest relevante toepassingsgebied.

## 14. Welke didactische uitdagingen ziet u in het kader van het aanleren van secure programming?

Ik zie dit vak niet als programmeeronderwijs, en aanleren van secure programming is geen kerndoel van het vak, maar eerder een bijeffect.

## 15. In hoeverre is het belangrijk het gevolg van een kwetsbaarheid aan studenten kenbaar te maken?

Cruciaal, zonder dit inzicht kan een student niet goed de mogelijke risico's analyseren.

### 16. Welke specifieke onderwerpen of concepten vindt u essentieel om op te nemen in een curriculum voor secure C++ programming?

Daar heb ik niet goed zicht op, het vak zoals ik dit geef zie ik niet als programmeeronderwijs. De studenten kunnen al programmeren als ze beginnen.

## Toekomst en trends (15 minuten, 70-85)

### 17. Wat is uw visie op het gebruik van AI m.b.t. secure programming?

Gezien de grote tijdsbesparing wordt is dit nu al een belangrijk gereedschap geworden, en dat zal het enkel meer worden. Ontmoedigen heeft dan ook geen zin. Huidige LLMs hebben geen goed inzicht in vulnerabilities, en schrijven regelmatig kwetsbare code. Dit is een risico voor programmeurs die de genereerde code blindelings gebruiken. Bewustwording hiervan is belangrijk.

### 18. Welke trends ziet u nu en voor de toekomst m.b.t. secure programming?

Geleidelijk zullen er meer defenses en safe languages gebruikt gaan worden, maar ik verwacht geen fundamentele verschuivingen. Toenemend gebruik van AI is relevant zoals hierboven genoemd.

### 19. Wat vind u van het idee om gamification te gebruiken in het aanleren van secure programming, i.p.v. de oplossing en uitleg direct te geven (Find The Vulnerability-FTV)?

We maken hier gebruik van voor een mastervak, en het is erg populair bij de studenten en een goede manier om het materiaal te oefenen. Echter, er zijn problemen met ongeoorloofd samenwerken (moeilijk te ontdekken in dit geval) en de aanpak is voor sommige groepen studenten intimiderend (dit zet de diversiteit onder druk).

## Afsluitende vraag (5 minuten, 85-90)

### 20. Welke bronnen of materialen beveelt u aan voor zowel docenten als studenten in het kader van secure (C++) programming? Zou u een dergelijk lijstje naar mij willen mailen?

Voor web security heeft OWASP veel goed materiaal. Voor de andere onderwerpen hebben we ons lesmateriaal over de jaren zelf opgebouwd, en zouden niet echt iets aanbevelen.

### 21. Zijn er nog andere aspecten of inzichten over secure C++ programming die u belangrijk vindt om te delen?

Niet dat ik me zo kan bedenken.