INSTITUTO
UNIVERSITÁRIO
DE LISBOA

Improving Industrial Cybersecurity Training: Insights into Code Reviews Using Eye-Tracking

Samuel Miguel Riegel Correia

Master in Computer Engineering

Supervisor:
Doctor Maria Cabral Diogo Pinto Albuquerque
Assistant Professor
Iscte – Instituto Universitário de Lisboa

Co-Supervisor:
Doctor Tiago Espinha Gasiba
Senior Key Expert and Cybersecurity Researcher
Siemens AG

September, 2024

TECHNOLOGY
AND ARCHITECTURE

Department of Information Science and Technology

Improving Industrial Cybersecurity Training: Insights into Code Reviews Using Eye-Tracking

Samuel Miguel Riegel Correia

Master in Computer Engineering

Supervisor:
Doctor Maria Cabral Diogo Pinto Albuquerque
Assistant Professor
Iscte – Instituto Universitário de Lisboa

Co-Supervisor:
Doctor Tiago Espinha Gasiba
Senior Key Expert and Cybersecurity Researcher
Siemens AG

September, 2024

I am deeply grateful to my mother for her unwavering support throughout my academic journey. Her encouragement, understanding, and sacrifices have been instrumental in my success.

Acknowledgments

I extend my sincere thanks to my supervisors for their guidance and support throughout the development of this project. Additionally, I am grateful to Siemens for their support and collaboration, which greatly contributed to the success of my work.

Resumo

À medida que as ciberameaças se tornam mais sofisticadas, os setores industriais enfrentam novos desafios na salvaguarda de infra-estruturas críticas. Esta dissertação explora o potencial de melhorar a formação em cibersegurança industrial através da aplicação de tecnologias de rastreamento ocular durante as revisões de código. Através da analise dos processos cognitivos de profissionais de cibersegurança enquanto tentam identificar vulnerabilidades no código de software, no que pode ser derivado de dados de rastreio ocular, este trabalho visa fornecer informações valiosas para melhorar as práticas de programação segura. Através de uma revisão sistemática da literatura, um inquérito, e de experiências de rastreio ocular, este estudo identifica tarefas críticas no ciclo de vida do desenvolvimento de software, avalia a eficácia de revisões de código e investiga a relação entre os padrões de atenção visual e o sucesso na deteção de vulnerabilidades. Especificamente, analiza a forma como os profissionais detectam algumas das vulnerabilidades de cibersegurança mais predominantes, incluindo SQL injections e cross-site scripting. Os resultados contribuem tanto para o conhecimento académico como para aplicações práticas, abrindo o caminho para melhores metodologias de formação e reforçando as defesas industriais em termos da cibersegurança no desevolvimento de software.

Palavras-Chave: *revisão de código*, *cibersegurança*, *vulnerabilidades de código*, *rastreamento ocular*, *programação*

Abstract

As cyber threats become more sophisticated, industrial sectors face unprecedented challenges in safeguarding critical infrastructure. This dissertation explores the potential of enhancing industrial cybersecurity training through the application of eye-tracking technology during code reviews. By analysing the cognitive processes of cybersecurity professionals as they identify vulnerabilities in software code, in what can be derived from eye-tracking data, this work aims to provide valuable insights into improving secure coding practices. Through a systematic literature review, survey, and eye-tracking experiments, this study identifies critical tasks in the software development lifecycle, evaluates secure code review effectiveness, and investigates the relationship between visual attention patterns and vulnerability detection success. Specifically, it examines how professionals detect some of the most prevalent cybersecurity vulnerabilities, including SQL injection and cross-site scripting. The findings contribute to both academic knowledge and practical applications, paving the way for improved training methodologies and strengthening industrial cybersecurity defences in software development.

KEYWORDS: *code reviews*, *cybersecurity*, *code vulberabilities*, *eye-tracking*, *programming*

Contents

List of Figures

List of Tables

List of Acronyms

AOI: areas of interest

CSP: content security policy

CWE: common weakness enumeration

FOV: field of view

IDE: integrated development environments

LLM: large language model

LRRQ: literature review research questions

PDU: pupil diameter unit

SAST: static application security testing

UX: user experience

VR: virtual reality

XSS: cross-site scripting

CHAPTER 1

Introduction

1.1. Context

The industrial sector is facing unprecedented challenges in cybersecurity as systems become progressively interconnected and threats become increasingly sophisticated, frequent, and costly [1]. The integration of digital technologies in industrial environments exposes many new vulnerabilities which can be exploited by cybercriminals [2]. This evolution has made cybersecurity a top priority for organizations, especially in critical infrastructure sectors such as communications, energy, and transportation, where system downtime or breaches can result in significant operational and financial losses.

Recent high-profile cyberattacks have highlighted the weaknesses in systems and existing security measures. As industrial environments adopt new technologies such as IoT, cloud computing, and automation, the attack surface grows, making it essential to adopt more stringent security practices. Governments and other regulatory bodies have responded by introducing cybersecurity frameworks and standards that aim to improve the resilience and imperviousness of industrial systems against cyber threats.

Secure coding is one of the most important areas of cybersecurity, with its primary goal being the identification and elimination of vulnerabilities in software systems. A cybersecurity vulnerability is a weakness or flaw in software that can be exploited by attackers to compromise system security, potentially leading to unauthorised access, data breaches, or other malicious activities. In this work, we will reference the common weakness enumeration (CWE) vulnerability list [3]. This list not only catalogues vulnerabilities but also outlines the relationships between various types of vulnerabilities.

Secure code reviews are an important task in the software development lifecycle, playing a pivotal role in creating secure code and reducing these types of vulnerabilities. Code reviewing is the practice of systematically examining source code to find defects and improve code quality [4]. A secure code review focuses on finding possible cybersecurity vulnerabilities. Code reviews are also commonplace in development lifecycles and have been set as requirements in industrial standards such as the ISO/IEC 62443 series and many others.

Due to their importance, code reviews have often been researched with the objective of determining how to best conduct them. Code reviews mostly involve reading program code while attempting to find flaws. According to our research, this makes eye-tracking technologies an appropriate and popular approach when attempting to study professionals' cognitive processes during this task, mainly because of the strong visual component associated with the task.

Eye-tracking is a method used to collect information regarding individuals' cognitive processes using non-invasive methods [5, 6]. It can reveal to us the thought processes of individuals while resolving tasks without causing any additional effort or distraction, which is not the case when, for instance, conducting think-aloud studies [7]. Eye trackers monitor individuals' visual attention by collecting eye movement data and, with this data they can, among other things, map a user's gaze to a specific location of a screen the user is looking at. These devices can also capture other important information such as the saccades also known as rapid eye movements, blink rate, blink duration, and pupil diameter, which allow for a more in-depth analysis of the user's attention [8].

In our research, we focus on the area of study known as gaze tracking. Eye-tracking and gaze-tracking are two terms that are often used interchangeably, even though a subtle distinction exists. Eye-tracking mainly involves detecting eyes, obtaining a precise interpretation of eye positions, and frame-to-frame tracking of detected eyes. Gaze tracking or gaze estimation, however, is the process of modelling a person's 3D line of sight or, in other words, determining where a person is looking [9]. In a sense, gaze tracking is a use case of eye-tracking technology.

The work we present here was, in part, initially introduced in a research paper written for the 5th International Computer Programming Education Conference [10]. In that paper, we provide a preliminary analysis of the importance of secure code reviews among industrial cybersecurity professionals, using data collected through surveys and eye-tracking experiments. This dissertation extends the scope of the aforementioned conference paper.

Here, we delve deeper into the methodologies used, present new results, and employ other analysis techniques to provide a more comprehensive understanding of our findings. We present a detailed examination of the data collected and draw more significant conclusions about the role of eye-tracking in improving these practices. By expanding on the initial research, this dissertation aims to contribute valuable insights into the enhancement of cybersecurity measures within the software development lifecycle, specifically when it comes to code reviews.

1.2. Motivation

Numerous research studies have successfully delved into the application of this technology to gain insights into the cognitive processes of individuals while undertaking programming tasks [6, 11–13]. The promising results of these studies have paved the way for further research and development in this field. However, very little research has focused on cybersecurity.

Considering the increasing importance of cybersecurity in industrial settings, and the potential consequences of overlooking these vulnerabilities, studying how processes, training, or artefacts related to security can be improved, is of great importance.

Recognising an opportunity to enhance internal processes related to cybersecurity, Siemens decided to explore the potential of eye-tracking technology in this domain. By

investing in this project, the company aims to improve internal processes, while also contributing to the scientific community by connecting behaviour analysis and eye-tracking in the context of cybersecurity. This study took place, on-site, at Siemens Munich-Perlach.

In our work, we aim to explore this crucial subcategory of eye-tracking studies by analysing tasks critical to cybersecurity, such as detecting cybersecurity vulnerabilities in program code through code reviews. The study identifies key tasks in the software development lifecycle, evaluates secure code review effectiveness, and investigates the relationship between visual attention patterns and vulnerability detection success.

1.3. Research Questions

As a guideline for our work, we chose to address the following research questions:

RQ1 What tasks in the software development lifecycle do industrial cybersecurity professionals consider to be most crucial in mitigating cybersecurity vulnerabilities?

RQ2 How successful are industrial cybersecurity professionals at conducting secure code reviews?

RQ3 Is there a relation between the patterns revealed using eye-tracking technology and the code reviewers' success in spotting the vulnerabilities?

The first research question (*RQ1*) aims to understand how important experts consider different types of tasks, including secure code reviews, in reducing cybersecurity vulnerabilities. The second and third research questions were created to determine how proficient experts are at identifying these vulnerabilities (*RQ2*) and what characteristics influenced their performance (*RQ3*).

1.4. Outline

This document consists of six subdivided chapters. After this, Introduction chapter, the Previous Work chapter presents the state of the art of industrial cybersecurity, standards, and code review research methods. We also conduct a systematic literature review of publications related to our research.

In the Methodology chapter, we present a survey and an experiment involving eye-tracking, with the results from these two parts being used to answer our research questions.

Next, the Results chapter, showcases the different results we obtained from both the survey and the experiment. This chapter is followed by our Discussion presenting an analysis of the aforementioned results and threats to their validity. Finally, in our sixth chapter, Conclusion, we present our conclusions, limitations, and possible future work.

CHAPTER 2

Previous Work

In this chapter, we present the state of the art of industrial cybersecurity, research on code reviews and eye-tracking technology. We also present previous work related to the research we planned to conduct, which was gathered through a systematic literature review.

2.1. State of the Art

In response to growing threats, cybersecurity standards have been developed to provide structured guidelines for securing industrial systems. One of the most critical components of these standards is the requirement for systematic code reviews, which are integral to the validation of software security. Secure code reviews play a vital role in identifying and addressing vulnerabilities early in the software development process, ensuring that security is built into the product from the outset.

Several cybersecurity standards and frameworks mandate or recommend code reviews. A foundational standard in this area is the NIST SP 800-53 (National Institute of Standards and Technology) [14], a security framework initially developed for U.S. federal agencies. NIST SP 800-53 provides a set of guidelines for federal information systems, emphasising the importance of secure code reviews as a method to identify security weaknesses. While originally aimed at federal institutions, this standard has become widely adopted across various sectors due to its robust security controls, which include recommendations for both manual and automated code reviews to verify software security compliance.

The PCI DSS (Payment Card Industry Data Security Standard) [15], another influential security standard, is designed to protect cardholder data in payment systems. Mandated by the major credit card companies, PCI DSS applies to organizations that handle branded credit cards from major schemes like Visa, MasterCard, and American Express [16]. This standard specifically requires that organizations review code for vulnerabilities, either through manual code reviews or automated scanning tools. As financial data remains a top target for cybercriminals, PCI DSS's stringent security measures, including mandatory code reviews, are critical for protecting sensitive financial information.

The OWASP ASVS (Application Security Verification Standard) [17] serves as an industry-recognized framework for web application security, outlining best practices and security verification techniques. OWASP ASVS recommends both manual and automated code reviews as part of an organisation's application security verification process. The standard underscores the importance of code reviews in identifying a broad range of security issues, from input validation errors to authentication weaknesses.

Another example of these standards is the IEC 62443 series (International Electrotechnical Commission), which provides a comprehensive framework for securing industrial automation and control systems. The series is designed to address the full lifecycle of cybersecurity, from product development to system maintenance, and emphasises the use of code reviews to ensure that software meets security requirements. Specifically, IEC 62443-4-1 [18] outlines security development lifecycle requirements for industrial control systems, recommending secure development practices such as testing and risk reduction, which can include code reviews as part of ensuring secure software development. IEC 62443-4-2 [19] focuses on the technical security requirements for industrial automation components, emphasising the importance of secure design and implementation.

By recommending or requiring code reviews, these standards emphasise the importance of thorough scrutiny of source code to detect and eliminate security vulnerabilities. As a verification and validation strategy, code reviews are effective in uncovering coding errors, design flaws, and non-compliance with security protocols, all of which could be exploited by malicious actors. Implementing code reviews as part of a broader security strategy ensures that software is rigorously examined for potential risks, thereby reducing the overall attack surface.

Through our research, we found that an often-seen approach in research on the topic of code reviewing is the usage of eye-tracking devices due to their capability to expose the cognitive processes of participants without much if any interference [6].

The upcoming systematic literature review section will delve deeper into studies conducted on this topic, however, these studies primarily aim to understand how programmers process information and solve problems during coding challenges. To do so, researchers analyse data obtained from eye-tracking devices and identify areas of focus and attention of the participants, measure their cognitive load, and evaluate the effectiveness of their problem-solving strategies.

When it comes to the current state of the art of eye-tracking technology, there are different types of devices. Remote eye-tracking, according to our research, is the most popular and widespread option due to its flexibility, ease of use, and the multitude of use cases (example seen in Figure 1). These remote devices are systems which do not require direct contact with the participant, instead, the device is placed at a distance and captures the position of the participant's eyes.

These standalone devices use specialised cameras, typically with infrared capabilities, which track the previously mentioned eye-specific information and give approximations of the user's gaze [8]. In the case of these infra-red-based eye-tracking devices, the invisible infra-red light is directed into the eyes of the users, which is reflected by the retina, causing the pupils to appear very bright. This so-called corneal reflection is captured by the device's cameras while image processing algorithms detect and track the eyes. Sharifa et al. [12] discuss how this data can be further processed and modelled to obtain reliable gaze location predictions.

FIGURE 1. Gazepoint GP3 Eye-tracking device

Tobii is a leading company in the field of gaze-tracking technologies [20]. They produce mounted eye trackers, as well as head-mounted units in the form of glasses and virtual reality (VR) headsets with embedded eye-tracking capabilities. The company provides equipment for various applications, including scientific research, marketing, user experience (UX), healthcare, and automotive technology. Their success and recognition are partly attributed to the development of technologies specifically designed for video games and e-sports.

Other companies exist, offering comparable devices with similar performance and features. Gazepoint [8] is a Canadian company that develops high-performance solutions for eye-tracking, biometric research and neuromarketing. Their products and solutions address the needs of researchers, development, marketing, UX, as well as education.

Siemens selected the Gazepoint GP3 eye-tracking device for this research due to its precision and suitability for research applications. The GP3 is engineered to provide reliable and accurate data, making it a valuable tool for studying the cognitive processes of professionals during code reviews. Its proven integration capabilities with research software and successful use in similar studies reinforced the decision to utilise this device for conducting eye-tracking experiments in the context of this study.

While some cost-effective eye-tracking systems exist, the prices for these specialised systems are usually very high, which can limit their access for research purposes. Alternative solutions exist which do not require an eye-tracking device. An implementation of such a system is GazeRecorder, which simply requires a webcam to function. Our research has shown that it is possible to conduct gaze-tracking studies and obtain fairly reliable recordings of gaze predictions of users with this program. According to our experience, accuracy is this method's biggest downside when compared to standalone gaze-tracking devices. GazeRecorder has been successfully used in many research projects [21–23] and is free for non-commercial uses.

Through our research, we found that eye-tracking usage has steadily increased over recent years, with the creation of new user-friendly and cost-efficient systems being an

important factor in its popularisation. These systems are now commonly used for many tasks such as UX studies, as input devices for applications, and in various research areas, including programming and cybersecurity.

2.2. Systematic Literature Review

In this systematic literature review, we present the current state of research using eye-tracking technologies in cybersecurity, programming, and UX. While our work focuses on analysing processes related to cybersecurity, we also chose to include publications on programming and UX as these subjects are also relevant to the context of our work. Programming has a clear connection to cybersecurity, and the analysis of a task such as a code review is closely tied to software development and programming. Furthermore, we consider UX studies to be relevant to our work since the usage of eye-tracking in this field is widespread and, as such, general procedures which can be applied to our study are well documented and defined.

We will first outline our defined search criteria, followed by a presentation of the conclusions regarding the research trends identified through this systematic literature review.

2.2.1. Systematic Literature Review Criteria Definition

To conduct this systemic literature review, we defined several criteria which are presented here. The following literature review research questions (LRRQ), seen in Table 1, were created to help guide the research procedure.

Table 1. Literature Review Research Questions

LRRQ1	How many articles on eye-tracking have been published over the years?
LRRQ2	How many articles specifically cover eye-tracking in programming, cybersecurity, or user experience studies?
LRRQ3	What are common subjects that these studies cover?
LRRQ4	What studies have been conducted with eye-tracking?

After defining our research questions, we focused on creating a search prompt that would help us answer these questions. This search prompt incorporated specific keywords defined to obtain all articles which refer to eye-tracking and one of the aforementioned subjects: cybersecurity, programming, or user experience.

This search prompt is composed of two parts conjoined by an "AND" to create the intersection of results obtained from both. The first part specifies articles that refer to eye-tracking technologies by using synonyms of this term and other spellings of it. The second part defines the subjects that should be referred to in the publications. Note that to obtain an answer to LRRQ1 only the first component of the search prompt, relative to only eye-tracking, was used. The following corresponds to our basic search prompt:

```
('Eye-tracking' OR 'Eye Tracking' OR 'Gaze-tracking' OR 'Gaze Tracking')
AND
('Cybersecurity' OR 'Programming' OR 'Code' OR 'User Experience')
```

The appropriate translations of the keywords were also included to include articles that are not only written in English but also in Portuguese, German, or Spanish:

```
('Eye-tracking' OR 'Eye Tracking' OR 'Gaze-tracking' OR 'Gaze Tracking'
OR 'Blickverfolgung' OR 'Rastreamento Ocular' OR 'Registro Visual')
AND
('Cybersecurity' OR 'Cyber-Security' OR 'Cyber Security'
OR 'Cybersicherheit' OR 'Cibersegurança' OR 'Seguridad Cibernética'
OR 'Programming' OR 'Code' OR 'Programmierung' OR 'Programação'
OR 'Programación' OR 'User Experience' OR 'Benutzererfahrung'
OR 'Experiência de Utilizador' OR 'Experiencia de Usuario')
```

The inclusion criteria outlined in Table 2, define the characteristics that articles must meet to be considered for inclusion.

TABLE 2. Systematic Literature Review Inclusion Criteria

IC0	Publication is written in English OR Portuguese OR German OR Spanish.
IC1	Publication includes the defined keywords either in the title OR abstract.
IC2	Publication defined in a relevant subject area such as Computer Science, or Engineering.
IC3	Publication openly accessible or within ISCTE's scientific license.
IC4	Publication was released in the last 5 years (from 2018 onwards).
IC5	Publication type: article OR paper OR abstract.
IC6	Publication covers a technical aspect of creating an experiment on eye-tracking in one of the desired fields OR describes and analyses the results of such a study.
IC7	Publication presents an analysis of subject behaviour with eye-tracking, in a field related to cybersecurity OR programming OR user experience related to the usage of an application.

While we were able to apply most of these inclusion criteria automatically when searching for publications, the last two required a manual review of the articles. This process helped identify some relevant trends that have developed in the subject areas specified, even if some publications were ultimately excluded based on some of the criteria.

Instead of individually applying each inclusion criterion, we chose to group these into filters. Filters were applied gradually while registering the number of articles retained after applying each one of these. Note that IC0 is always applied. The following are the filters we created and the inclusion criteria they represent:

Filter 1: IC1

Filter 2: IC2 AND IC3

Filter 3: IC4 AND IC5

Filter 4: IC6 AND IC7

Exclusion criteria describe characteristics which should not be present in the included articles, beyond what would correspond to a negation of the inclusion criteria. These criteria required a manual review of the articles to be applied and were, for this reason, applied at the same time as Filter 4. The exclusion criteria can be seen in Table 3.

TABLE 3. Systematic Literature Review Exclusion Criteria

EC1	Publication presents sub-par linguistic correctness.
EC2	Publication related to analysing a very specific demographic, such as individuals with special needs.
EC3	Publication describes the use of eye-tracking technologies with something other than a normal computer screen (mobile phone usage, virtual/augmented reality).
EC4	Publication describes the usage of eye-tracking for user inputs.
EC5	Publication describes the usage of eye-tracking data to dynamically adapt the content presented to the user.
EC6	Publication describes the presentation of eye-tracking data to allow users to visualize other users' gazes.

Lastly, we defined which databases we would use to conduct our systematic literature review. These databases were selected as they contain a vast amount of publications from different fields of research including a large emphasis on articles related to computer science. Additionally, these databases allowed us to apply many of the inclusion criteria automatically, which greatly accelerated our review process. These are the databases we chose to consult:

- ACM Digital Library
- Scopus
- IEEE Xplore
- b-on library (using EBSCOhost)

Due to the overlapping content of these databases, duplicate articles were removed after all filters were applied. Consequently, the reported number of articles does not account for duplicates discovered in multiple databases, except for the numbers presented after applying all filters.

2.2.2. Systematic Literature Review Results

To determine the answer to LRRQ1, we first looked at the evolution of the number of articles which refer to the term eye-tracking or one of its other forms and translations. Figure 2 shows the number of articles that refer to eye-tracking between 2013 and 2023. From this graph, we can see that there exists an established and growing interest in using this technology for scientific research.

This analysis highlights the disparity in the number of publications we obtained from each database. We observed that the Scopus and b-on databases presented, by a considerable margin, the largest number of articles referring to eye-tracking. These two are comprehensive multidisciplinary databases that include articles from various scientific disciplines. In contrast, the ACM digital library and IEEE Xplore are much smaller and specialize in certain disciplines, namely, engineering, technology, and computer science.

Note that duplicate articles may be considered in the numbers we present as these were only removed after applying the final filter.

Next, we focused our research on determining the popularity of the usage of eye-tracking technology in the use cases being considered. To do so, the previously defined criteria were applied, step by step, and results were recorded. Table 4 presents the number of articles obtained from each database after applying all filters and exclusion criteria described in the previous section.

The number of articles per year was also analysed. This information gives us some insight into how the popularity of our subjects has changed over the years. We present two graphs: Figure 3 shows the number of articles published from 2013 onwards before applying any filters; Figure 4 presents the number of articles obtained from each database after applying all filters.

After applying all criteria and removing any duplicates, the total amount of articles obtained after applying all filters is 124. Whereby four articles are on cybersecurity, 91 on programming, and 29 on UX.

Table 5 shows the number of articles corresponding to each category, divided by databases. This table gives insight into LRRQ2 by presenting how many articles have been published for each subcategory. As some of the publications we obtained are present in multiple databases, the total number of articles differs from the sum of the articles found in each of the databases.

Table 5 shows that the number of articles on cybersecurity studies is relatively small compared to the other categories, representing only 3% of total publications. We found only four articles in this category, these were analysed and presented in subsection 2.2.4. Despite eye-tracking being commonly used to analyse user experiences, the number of articles on UX was lower than anticipated, with less than one-fourth of articles retained after applying all filters being related to UX.

The research questions LRRQ3 and LRRQ4, which require further analysis, are addressed in the following parts.

2.2.3. Trends in Research

In the publications we analysed, we found that articles on UX often included experiments in which a group of users was asked to compare different implementations of user interfaces [24–26]. Other topics included the use of eye-tracking as a user input [27] and the usage of the live eye-tracking data to automatically modify the displayed content [28].

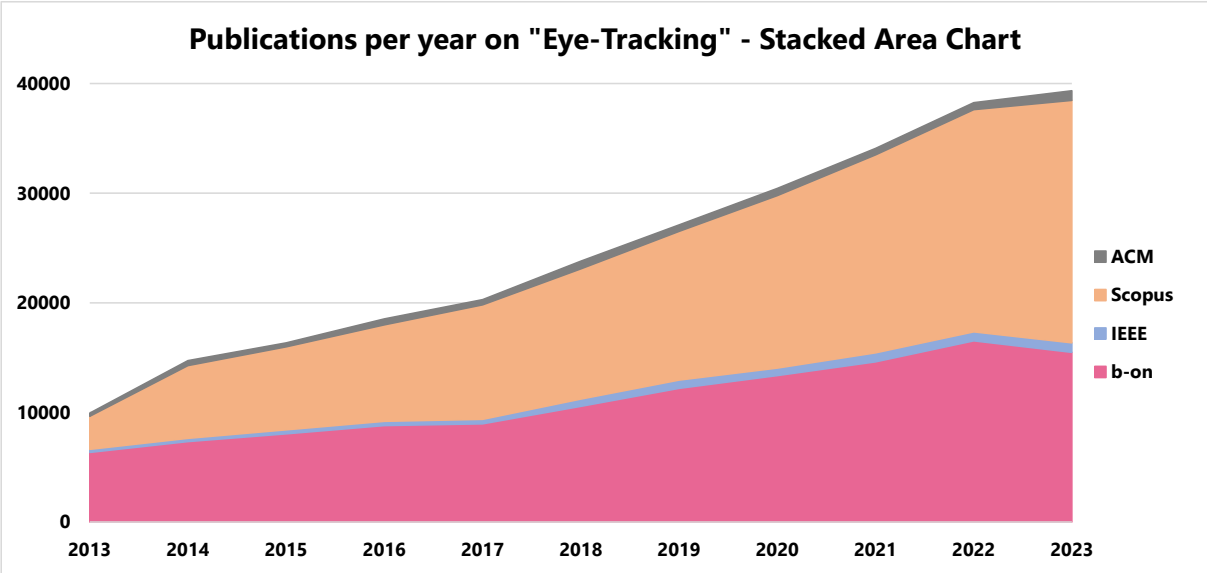


FIGURE 2. Distribution of articles which refer "Eye-tracking" 2013-2023

TABLE 4. Number of articles for each filter and database

Database	No Filter	F1	F2	F3	F4
ACM	3914	200	198	110	48
Scopus	16583	1094	248	187	46
IEEE	751	191	189	117	40
b-on	52439	967	335	179	39

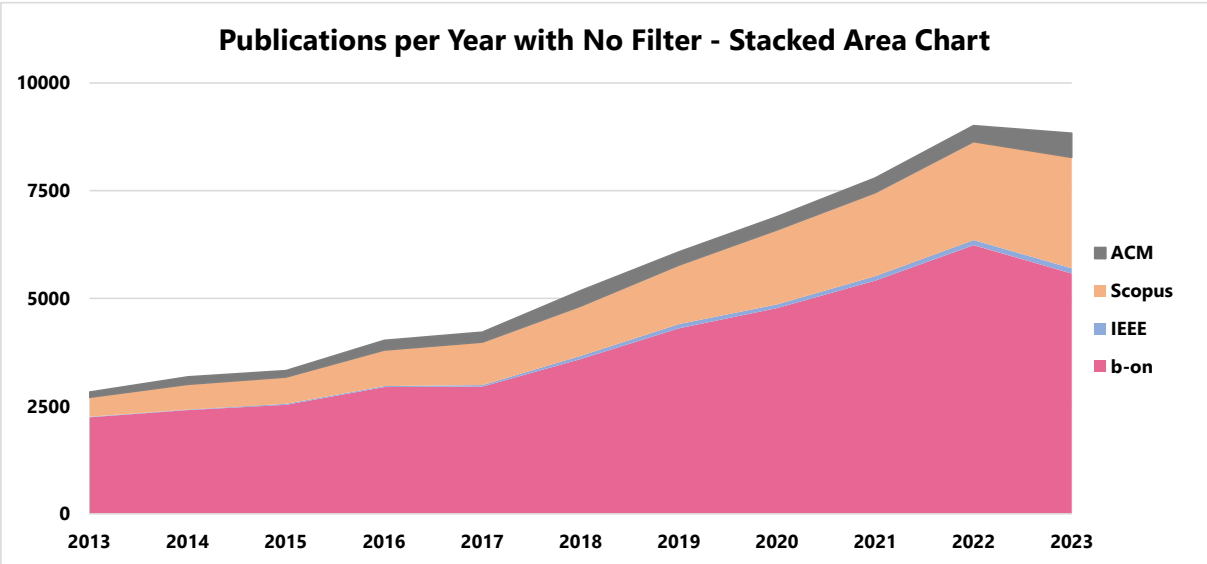


FIGURE 3. Distribution of articles in selected topics with no filters 2013-2023

The usage of eye-tracking data to dynamically change the content presented to the user is often referenced in the context of virtual or augmented reality applications. These publications also mentioned several user experience testing strategies which would also be relevant to our work, such as starting from the easiest tasks and working towards the

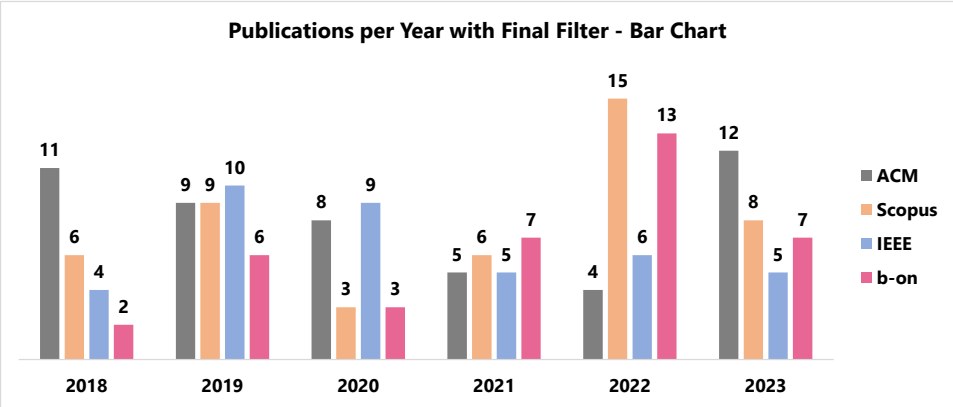


FIGURE 4. Distribution of articles in selected topics with the final filter 2018-2023

TABLE 5. Number of articles for each category and database

Database	Cybersecurity	Programming	User Experience
ACM	2	41	5
Scopus	1	32	13
IEEE	3	28	9
b-on	1	27	11
TOTAL	4 (≈3%)	91 (≈74%)	29 (≈23%)

harder ones, making sure the participant feels comfortable to ensure that we obtain close to real-world results, and other elements related to these types of experiments such as practice runs and the creation of procedure guides.

As for the articles we discovered on programming, these included comparisons between the way experienced and novice programmers look at code [29], how developers consult documentation and search for guidance online [11, 30], and how model-generated code compares to human-written code in terms of readability [31], among other topics.

Out of all of the articles on programming, analysis of how individuals look at code or comparisons between how different groups of individuals look at code were, by far, the most common subjects in the discovered articles.

Other studies sought to determine the effectiveness of sharing gaze data between users to more accurately convey ideas between individuals. These articles usually focus on pair programming, and attempt to help pairs of programmers to more easily follow each other's thought processes [32].

Many articles analysed several topics related to education, especially education related to programming. For instance, researchers have leveraged eye-tracking devices to help determine a student's understanding of code [33, 34], while similar studies have been conducted for other disciplines such as mathematics [35], or even to analyse how students read English text [23]. Others also explored specific educational challenges related to the education of children or individuals with certain conditions such as dyslexia [36]. The

vast majority of these articles on education had the objective of informing educators on how to improve their teaching processes.

It was noted that for the topic of programming many articles explicitly mention education and many more implicitly have an impact in this field. This comes from the fact that many articles study how certain individuals read code, something that is very relevant when attempting to educate people on the best practices of this task.

2.2.4. Trends in Cybersecurity Research

After applying all filters, we discovered four articles directly referring to cybersecurity. Furthermore, three of these articles were related to coding practices from a cybersecurity perspective and aim to further secure coding practices, while the last one describes the procedure to conduct an eye-tracking study to help in training against phishing attacks.

On Integrating Security-Relevant Information in Non-Security API Documentation to Support Secure Coding Practices by Gorski et al. [37] looks at how programmers read documentation to implement certain features. Specifically, this article focuses on how developers look at API documentation and content security policy (CSP) implementation. This revealed, among other findings, that programmers often solely concentrate on code examples, overlooking essential information crucial to secure coding. The article also highlights how documentation should be adapted or enhanced to more effectively convey security-relevant information. The main conclusions include a recommendation to integrate important cybersecurity information into the code examples.

Two studies identified in our review focused on how educational methods can be adapted to enhance secure coding practices. *Understanding and Improving Secure Coding Behavior with Eye Tracking Methodologies* by Davis and Zhu [38] conducts a study in which students manually analysed source code, and were tasked to select the code snippet which presented the best solution out of several options. From this study, the authors determined that a "think before you act" approach is critical when solving these types of problems, concluding that the most effective and efficient strategy to create secure code is to fully understand the errors before attempting to fix them. In the study, this is reflected by individuals who answered correctly having a higher fixation duration on each possible answer.

This article uses CWE-based definitions to categorise and describe vulnerabilities. The usage of these definitions is commonplace in the field of cybersecurity since they provide a standardised framework and common language for identifying and discussing software weaknesses. These definitions facilitate clearer communication among security professionals by ensuring that everyone uses the same terminology and understands the nature of vulnerabilities consistently. With CWE's comprehensive list of known software weaknesses, professionals can precisely identify the type and characteristics of a specific

weakness, which is essential for accurate documentation. Not only does CWE provide in-depth descriptions of weaknesses and flaws, but it is essentially an ontology with relations between issues and possible attack vectors that originate from these issues.

Moreover, CVEdetails.com [39] is an online database that catalogues software vulnerabilities, offering detailed information about each one, including its impact, severity, and occurrences. This database aligns with the CWE framework, providing a structured approach to understanding and addressing security weaknesses in software. Additionally, we found CVEdetails.com to be useful in designing our study, particularly in determining which types of vulnerabilities have the highest number of registered occurrences.

The previous article, as well as *Minimizing Cognitive Load in Cyber Learning Materials* by Bernard et al. [40], both present alternative learning frameworks. The latter focuses on determining how cognitive load influences cybersecurity education and how to avoid cognitive overload when working with this complex subject. It proposes a model that "incorporated Bloom's taxonomy and the design principles of segmentation and interactivity" to reduce intrinsic and extraneous load while increasing germane load. This article presents a study with the proposed learning framework vs a traditional learning module and uses different eye-tracking metrics to measure these cognitive loads, specifically fixation duration and pupil size.

Lastly, *Eye-Tracking System as a Part of the Phishing Training* by Madleňák and Kampová [41], proposes the use of eye-tracking in phishing prevention. Phishing is one of the most prevalent cyber threats we face today. These attacks use social engineering techniques to gain the victims' trust. As such, the most effective way of preventing this type of attack is through training. The article focuses on enhancing anti-phishing training through the use of eye-tracking. One of its stated claims is that through eye-tracking, we can understand which elements of e-mails or websites users focus on and, with this information, "gain insights into common mistakes and areas of vulnerability". The authors also propose an experiment involving a mock phishing email to analyse users' interactions with potentially malicious content, relying on key eye-tracking metrics.

We found that articles related to eye-tracking use in cybersecurity are closely related to education. The publications we highlighted were directly related to how we can create better pedagogical frameworks to educate individuals on cybersecurity, or how we could adapt provided learning materials such as API documentation to further safe cybersecurity practices.

Most articles about programming may also be relevant to cybersecurity and to the topic of reducing cybersecurity vulnerabilities, as analysing how individuals look at code may also give some insight into how they deal with certain cybersecurity challenges. Articles which focus on education are also relevant since, as we have seen, education is essential when trying to guarantee safe behaviours, be they specific to coding or general computer usage.

2.2.5. Surveys and Guides

A considerable amount of publications exist which specifically provide researchers with guides on how to best conduct eye-tracking experiments involving programming code.

In *Analysis of Software Developers' Coding Behavior* [42], the authors conduct a comprehensive review of objectives and techniques crucial to analysing software developers' coding behaviour using eye-tracking. This publication consists of a systematic literature survey and was significantly helpful in the design of our experiment. One of the elements from this article that was particularly important for our study is a decision matrix describing each software development lifecycle task in terms of its properties, such as type of stimuli, interactivity, viewing dimensions, and other characteristics. Each task in the decision matrix is associated with possible approaches and visualisations to analyse the eye-tracking data obtained from studying these tasks. From these tasks, we considered the following to be crucial in cybersecurity and, additionally, be suitable candidates for analysis through an eye-tracking study: code reviewing, analysis of static application security testing (SAST) tool outputs, reading documentation, and researching online resources (e.g. Stack Overflow).

This previously mentioned article, along with additional works [6, 43], provide survey results from the analysis of publications involving the use of eye-tracking in computer programming. These articles study all elements involved in these studies, from the eye-trackers used to the analysis of the experiments' results. These articles were instrumental in defining our approach to conduct the experiment and in evaluating the data we would obtain.

Some of the metrics these publications mentioned include fixation rate and pupil diameter. Fixation rate, i.e., the number of fixations per second has been linked to cognitive effort, interest, and exploration in tasks such as finding vulnerabilities in code [12, 43]. Pupil diameter is a relevant characteristic which has previously been linked to higher cognitive workload. Studies incorporating both eye-tracking and magnetic resonance/spectrography, have shown that pupils can dilate up to 0.5 mm above their relative baseline value during high cognitive loads [12, 13].

2.2.6. Tools

Beyond articles on these eye-tracking studies, many articles present tools to help in the creation of such experiments involving programming, the most notable of which being iTrace [44]. This tool helps create eye-tracking experiments in integrated development environments (IDE) such as Visual Studio [45] and Eclipse [46], by matching the gaze position of users with the fragments of code they are looking at. iTrace allows users to freely navigate the code while enabling researchers to associate gazes to specific parts of the code easily. By allowing individuals to use environments with which they are familiar, users feel more comfortable and closer to their natural state when performing these experiments.

iTrace is referenced in many of the articles we analysed and has been used in many eye-tracking experiments involving the analysis of code-reading behaviours. Since its creation, this tool has received updates, adding new features and plugins for other development environments. Some updates to iTrace are presented in articles before their introduction to the main application (e.g., [47] and [48]).

We have found that Drew T. Guarnera, Jonathan I. Maletic, and Bonita Sharif, some of the authors of the original paper, are also prominent authors of articles on eye-tracking for development environment experimentation. Not only are these authors featured in articles related to the updates to the iTrace tool, but they have co-authored many other articles on the research of cognitive processes of programmers using eye-tracking.

While iTrace allows researchers to obtain data which correlates to the exact positions of specific code instructions, the majority of studies we observed conduct their research using other methods such as areas of interest (AOI). These AOI are portions of the screen, usually defined using rectangles, which contain related information. For instance, a study may use AOI to differentiate parts of a coding question [49], to determine how a user analysed each option in a multiple choice question [38], or to register when a user is looking at code or at the information of the code's author [50].

While the granularity of results obtained with iTrace may be appropriate for some experiments, as we have seen, the usage of AOI is the standard approach for this type of study. iTrace does not provide tools to create and evaluate AOI, however, while we found no publications which explored this possibility, researchers could define these some other way and profit from both the granular analysis provided by the tool and the results obtained using AOI.

While we had initially planned to use iTrace for our study, due to technical reasons, we were not able to do so. An issue presented itself which could not be easily addressed even with the support from the tool's developer and community. As such, an alternative program was sought, as further efforts to resolve this issue would not be in the best interest of the project.

Many other tools exist for this type of analysis, with most studies being conducted with the companion programs provided by the eye-tracker manufacturers themselves (e.g., Tobii Pro Lab [51] or Gazepoint Analysis [52]). There also exist open-source tools like the Open Gaze and Mouse Analyzer (OGAMA) [53] which presents itself as an excellent alternative to these closed-source programs. OGAMA's main features include the creation of slideshows, recording gaze data, the database-driven preprocessing and filtering of gaze data, the creation of attention maps, AOI definition, and replay [54]. Despite being released over 15 years ago (in 2008) and no longer being actively developed by its original author, this tool is still used in numerous recent publications to conduct eye-tracking experiments [12, 33, 50]. For these reasons, OGAMA was chosen to be used in our study.

2.2.7. Conclusions on the Systematic Literature Review

By conducting this systematic literature review we were able to gather information on eye-tracking research for the topics we are focused on. We found that this technology has and continues to be used in several different forms with very distinct objectives, be it as an input for an application, to evaluate different implementations of a certain system by analysing how a user interacts with it, or to simply study how an individual reads program code. Notably, very few publications exist which specifically focus on using eye-tracking in cybersecurity research which further motivated us to continue our work.

We observed that research targeting how individuals analyse code frequently intertwines itself with education, as the insights gained from these projects can usually be used to enhance educational processes. While our results show that the number of publications on cybersecurity was limited, the ones we identified consistently focused on education. Similar to previous work, our research does not solely concentrate on education, however, it will naturally involve this subject.

In the publications gathered, we observe a variety of approaches, even for similar objectives. Specifically, there are differences in equipment, software, and techniques used to evaluate and interpret the eye-tracking data. Most of the studies involving code did not measure per-instruction fixations, but rather measured relatively large groupings such as options in multiple choice questions or blocks of code, with relatively few going to the lengths of registering per-instruction fixations and analysing this information. We believe this might be due to issues related to the precision and accuracy of such measurements, making the use of AOI in many cases preferable and more reliable.

This systematic literature review aided us in the selection of the tools we would use in our study, as through it we were able to obtain insights into various equipment and techniques for eye-tracking studies related to the work we planned on conducting. Publications we found particularly informative included the various surveys conducted on the usage of eye-tracking in computer programming. The surveys found in those publications reviewed a large variety of work conducted in this field and presented the equipment used, techniques applied, and conclusions obtained. These studies influenced our formulation and approach to conducting our study.

CHAPTER 3

Methodology

We approached our research questions through an empirical study, as the industrial context of the dissertation allowed for such a study to be conducted. Our study consists of two parts, a survey and an experiment where the survey aims to answer *RQ1* while the experiment tackles *RQ2* and *RQ3*. The survey and the eye-tracking experiment were conducted with the same participants, with the survey preceding the code review experiment.

3.1. Survey Design

A survey was created which collected participant demographics and responses to questions designed to answer *RQ1*. With this data, we were also able to determine whether any correlations existed between participants' performance in the experiment and their background information. The data collected through this survey is presented in Table 6.

The demographical background data we collected, marked BG, included: age, gender, level of education, and years of work experience in cybersecurity. Additionally, a question was included to determine the participant's self-reported proficiency in the C++ programming language as this was the language of the code snippets the participants would be presented with during the experiment. The participants' self-reported C++ proficiency was recorded through a five-point Likert scale [55]. On this scale, a value of one corresponds to novice, while five corresponds to expert-level proficiency.

Regarding the questions designed to answer *RQ1*, marked A, participants were asked to evaluate the following tasks also via a five-point Likert scale, where a rating of one corresponds to not important, and five corresponds to very important. These questions are represented by data points A1 through A4:

- *T1* Code reviewing
- *T2* Analysis of SAST tool outputs
- *T3* Reading documentation
- *T4* Researching online resources (e.g. Stack Overflow or other community-based resources)

When selecting the tasks to be studied with this survey we chose some which we considered relevant to the industry in terms of reducing or eliminating cybersecurity vulnerabilities, based on the research we had conducted. Additionally, the tasks chosen were also good candidates for eye-tracking studies since they involved patterns which could be easily recorded and analysed with this technology. Furthermore, an open-ended question was also included, corresponding to A5, which allowed participants to indicate

TABLE 6. Survey Data

BG1	Age
BG2	Gender
BG3	Education: Highest level of education completed
BG4	Experience: Years of work experience in cybersecurity
BG5	C++ Proficiency: Self reported proficiency in the C++ programming language
A1	Importance given to *T1*
A2	Importance given to *T2*
A3	Importance given to *T3*
A4	Importance given to *T4*
A5	Additional SW development tasks considered important

any other tasks they considered important, but which weren't included in the previous list.

This survey was created with Microsoft Forms [56], and participants responded to it on a computer supplied by the researcher before conducting the experiment. A PDF version of this survey can be found in Appendix A.

3.2. Experiment Design

To answer *RQ2* and *RQ3* we created an experiment in which participants would conduct a secure code review of several code snippets, with their main goal during these reviews being to find cybersecurity vulnerabilities. The data collected through this experiment is presented in Table 7

To evaluate the accuracy of our participants' answers (B1), we defined what constitutes a correct answer. To simplify this variable, we considered answers to be either correct or incorrect following some criteria we defined. This criteria is presented in section 3.3.

We would also like to give an additional note on the pupil diameter data (B8). While this characteristic is supported by our eye-tracking device and was recorded during the experiment sessions, OGAMA does not natively present this data to users. As such, we extracted this information from the raw data files before our analysis.

We were not able to determine the unit of measurement the pupil diameter values correspond to, according to the Gazepoint Analysis manual and Gazepoint API manual, the data should have been recorded in millimetres [57, 58]. Through testing, we found that these values could not possibly be in millimetres, nevertheless, they remained very relevant and useful in our research. We will, from this point onwards, refer to the unit of measurement for this characteristic as pupil diameter unit (PDU).

For this experiment, we defined what resources we required, the code snippets which our participants would analyse, and the procedure we would follow when administering the experiment. These will be presented in the following subsections.

TABLE 7. Experiment Data

B1	Accuracy: Accuracy of the description of vulnerabilities, i.e. right or wrong answer for each code snippet
B2	Time to Detect Vulnerability: The time spent before the participant identified the vulnerability
B3	Experiment Duration: Time to complete the experiment
B4	Code Snippet Analysis Duration: Time for the analysis of each code snippet
B5	Fixations: Number of fixations, their duration, and location
B6	Gaze Time: Time looking at a part of the code (for our analysis using AOI)
B7	Total Gaze Time: Time spent fixating on the code, i.e. the sum of fixation times
B8	Pupil Diameter: The diameter of the participants' pupils.

We also conducted a trial experiment, with the researcher who would administer the experiment acting as a participant. This trial served to find issues in the experiment setup we had created and any possible flaws in the experiment guide.

3.2.1. Hardware, Software, and Other Resources

The Gazepoint GP3 Eye-tracking device was used. Several factors influenced the selection of this device such as the fact that it was created mainly for research purposes and that it has been used successfully in other similar research projects. This device was obtained directly from the manufacturer.

As for the software used to record and analyse most of the data, the Open Gaze and Mouse Analyzer (OGAMA) [53] was used. OGAMA supports most features of other closed-source alternatives and even surpasses them in some aspects such as customisability and data output variety.

Other physical resources were also procured for the experiments, these included a computer, a monitor, and a room where the experiment could take place. The computer used was a personal work computer which exceeded all of the used software's recommended requirements. The monitor, which the participants would use to look at the code snippets, was a standard office monitor, namely a 24-inch Fujitsu brand display (b24w-5 eco). A meeting room was also reserved in which the experiments could be conducted, this room was equipped with a height-adjustable desk, seating for the participant and the researcher, and fully adjustable shutters to control the ambient light of the room.

3.2.2. Code Snippets

There are three types of code snippets we decided to include in our experiment: code snippets with vulnerabilities, proposed solution code snippets, and placebo code snippets. The first were code snippets which included a common vulnerability which our participants would have to find. The second was code snippets which presented solutions to some of the common vulnerabilities, not all of which were correct implementations of said

solutions. For this second type of code snippet, we asked our participants if the code we presented them solved the vulnerability. Lastly, placebo code snippets were code snippets representing programs with no cybersecurity vulnerabilities. As with the code snippets with vulnerabilities, participants were simply tasked with identifying any existing vulnerabilities.

The selection of relevant code snippets was essential as these would dictate what kind of results and, consequently, conclusions we would obtain from the experiment. In line with some of the articles we reviewed, we utilised the definitions of vulnerabilities from the CWE repository [3] to assign specific names and definitions to each vulnerability that participants were tasked with identifying.

For our vulnerable code snippets, we elected code snippets representing the most common weaknesses according to the number of registered occurrences on CVEdetails.com [39]. A Python script was created to obtain the number of registered occurrences of each issue on this website. This script can be found in Appendix C.

Table 8 shows the top ten most common vulnerabilities by number of registered occurrences on CVEdetails.com, their number of occurrences, and frequency i.e. the percentage they represent out of all registered occurrences.

We chose the top five most common vulnerabilities, i.e. CWE-79, CWE-119, CWE-89, CWE-20, and CWE-787. These also occupy top positions in CWE Top 25 Most Dangerous Software Weaknesses [59], which takes into consideration not only the frequency but also the severity of threat vectors.

Next, we searched for example code snippets for each of these weaknesses. Each of the snippets we chose, and not just the vulnerable code snippets, needed to be displayed as a single static image, without the need for scrolling. This meant that we had to limit our code snippets to around 50 lines of code to accommodate this requirement. This limitation resulted from our decision to use the eye-tracking study software OGAMA which doesn't support scrolling through images or text. Furthermore, the code snippets must be written in C++ as, according to our experience, this programming language is widely used in the industry and, particularly, in the company where the study took place. Most code snippets we discovered during our research were originally written in languages other than C++, these were translated to uniformise our code snippets.

For CWE-787, "Out-of-bounds Write", the code snippet was obtained from the CWE definition page's example snippets, specifically, example 5 [60]. CWE-119 and CWE-20, "Improper Restriction of Operations within the Bounds of a Memory Buffer" and "Improper Input Validation", had their code snippets based on the code snippets in Chetan Conikee's "seeve" repository [61]. For CWE-89, "SQL Injection", the code snippet was created based on relevant documentation including its CWE definition [62]. Finally, the code snippet for CWE-79, "Cross-site Scripting", was based on the corresponding code example in Yes We Hack's "Vulnerable Code Snippets" repository [63].

TABLE 8. Top Vulnerabilities and Number of Registered Occurrences

Ranking	Vulnerability	Occurrences	Frequency
1	CWE-79	26726	14.22%
2	CWE-119	11959	6.36%
3	CWE-89	11265	5.99%
4	CWE-20	10503	5.59%
5	CWE-787	10091	5.37%
6	CWE-200	7899	4.20%
7	CWE-22	5837	3.11%
8	CWE-125	5758	3.06%
9	CWE-352	5658	3.01%
10	CWE-264	5495	2.92%

We consider it important to give a brief overview of the contents of these code snippets, as this contextualisation will be important when considering our results. CWE-89's code snippet was a program that established a connection with a MySQL database and executed a query considering a username input by the user without any sanitation, which is where the SQL injection vulnerability originated. As for the code snippet used for CWE-79, it consisted of a small website that received requests and returned a simple HTML page with some information obtained from the request and therein lay the XSS weakness. In contrast, the other code snippets in our experiment included vulnerabilities closely related to memory management: Out-of-Bounds Write, Improper Restriction of Operations within the Bounds of a Memory Buffer, and Improper Input Validation.

For the analysis of the results, we defined what regions of these programs contained the vulnerable code. This region of the code is referred to as "target" and consists of a rectangle surrounding the vulnerable code. An example of this can be seen in Figure 5. The size and location of these rectangles were different for each vulnerable code snippet.

As for the code snippets representing solutions to vulnerable code, these are referred to CWE-89 and CWE-79. After each of these vulnerable code snippets, we first presented our participants with the lines of the code which contained the weaknesses and explained these to them in case they had not been able to identify them. Only after this did we present participants with the proposed solutions to these weaknesses, specifically, two for SQL injection and one for the cross-site scripting (XSS) vulnerability.

The most common approach to fixing a SQL injection weakness is to use prepared statements. Both of the solutions we presented to our participants used prepared statements, however, the first solution incorrectly implemented these, so the code still contained the same vulnerability. The second solution to the SQL injection vulnerability consisted of a correct implementation of prepared statements using the same C++ library which was used to establish the connection to the database.

For the XSS vulnerability, CWE-79, we only presented one solution to our participants. This solution consisted of a short program which sanitised any user input by replacing

```cpp
#include <iostream>
#include <string>
#include <boost/beast/http.hpp>
#include <boost/asio.hpp>

namespace http = boost::beast::http;
namespace asio = boost::asio;
using tcp = asio::ip::tcp;

void handleRequest(const http::request<http::string_body>& req,
        http::response<http::string_body>& res) {
    std::string query = req.target().to_string();
    size_t pos = query.find('?');
    std::string response = "<h1>" + query.substr(pos + 1) + "</h1>";

    res.set(http::field::content_type, "text/html");
    res.body() = response;
    res.prepare_payload();
}

int main() {
    try {
        asio::io_context io_context;
        tcp::acceptor acceptor(io_context, tcp::endpoint(tcp::v4(), 1337));

        while (true) {
            tcp::socket socket(io_context);
            acceptor.accept(socket);

            http::request<http::string_body> req;
            http::read(socket, req);

            http::response<http::string_body> res(http::status::ok, req.version());
            handleRequest(req, res);

            http::write(socket, res);
        }
    } catch (std::exception& e) {
        std::cerr << "Exception: " << e.what() << std::endl;
    }

    return 0;
}
```

FIGURE 5. Code snippet containing vulnerability CWE-79 with "target" area highlighted with a yellow rectangle

symbols which could be used to inject code, namely: &, \, ', <, and >, by applying HTML entity coding[1] to characters in the query received by the server.

These solutions were created by us, considering some common tactics used to solve these problems and mistakes which, according to our experience, are very common, such as the incorrect usage of prepared statements when addressing SQL injection issues.

We included two placebo code snippets, i.e. two code snippets with no flaws, referred to as Placebo 1 and Placebo 2. Placebo 1, corresponds to a program which counts the number of times a certain character appeared in a string, with both the character and the string being supplied by the user. Placebo 2, was a program which found the highest

[1]HTML entity encoding is a technique used in web security to convert certain reserved characters into their corresponding HTML entities (e.g., "&" becomes "&", "<" becomes "<"), preventing browsers from interpreting user-supplied input as executable code. This is commonly used to prevent injection attacks such as XSS.

common factor between two user-supplied values using recursion. These placebo code snippets were obtained from programiz.com [64].

Participants were allowed to look at the code snippets for as long as they deemed necessary, only moving to the next code snippets once they felt they were confident to do so. To remain within schedule, the researcher administering the experiment did, however, ask the participant if they were ready to advance if, for example, the weakness in one of the vulnerable code snippets had been found. Finally, following an important and well-established rule in user experience testing, we ordered our tasks by their difficulty [65], in our case, this corresponded to ordering the code snippets by their interpretation difficulty which we determined according to our experience. We arrived at the following order of code snippets:

(1) CWE-787: "Out-of-bounds Write"
(2) CWE-119: "Improper Restriction of Operations within the Bounds of a Memory Buffer"
(3) Placebo 1
(4) CWE-20: "Improper Input Validation"
(5) CWE-89: "SQL Injection"
(6) CWE-89: Incorrect solution
(7) CWE-89: Correct solution
(8) Placebo 2
(9) CWE-79: "Cross-site Scripting"
(10) CWE-79: Solution

These code snippets can be found in Appendix D.

3.2.3. Trial Experiment Conclusions & Experiment Guide

A document detailing the procedure was created to add consistency and structure to our experiments. This document covers all aspects of the experiment namely: introduction, experiment procedures, and debriefing.

A trial run of the experiment was conducted to check if any issues existed in the experiment procedure. During this trial, the experiment guide was altered and some previously unknown issues in the code, such as syntax errors or additional vulnerabilities, were also identified and corrected. Additionally, it was found that ambient brightness and light sources in the eye-tracker's field of view (FOV) strongly influence its performance. After some experimentation and the consultation of relevant documentation [66], we determined that, for optimal results, the environment in which the experiment takes place should, as much as possible, be devoid of natural light and have minimal artificial lighting. The room we had access to for the study had adjustable blinds which remained fully closed during the experiments to ensure consistent and reliable readings from our eye-tracking device.

Besides recording the screen and the eye movement data, audio recordings of the sessions were made as they would facilitate the posterior analysis of the results from our experiments. These audio recordings were timestamped during the experiments to mark the presentation of each different code snippet. All participants consented to these recordings beforehand.

The experiment was administered in English and German, depending on the preferred language of the participant. For either language, the experiment guide was closely followed. The document corresponding to the complete experiment guide can be found in Appendix B.

3.3. Evaluation

To evaluate data relative to *RQ1* we followed a standard analytical approach. Participants rated each task on a five-point Likert scale, which conveys the comparable importance given to each of the tasks they evaluated. We relied on simple metrics and visualisations such as diverging stacked bar charts to give us a good overview of the survey results (A1-A4). For the open-ended question (A5), each answer was processed to count the number of times each additional task was mentioned, this allowed us to present which other tasks our participants considered important.

As for *RQ2* and *RQ3*, our analysis was considerably more complex. We used different visualisation techniques for the eye-tracking data, analysed correlations between many distinct variables, and looked at the gaze paths of our participants.

To evaluate the accuracy of our participants' answers (B1), we defined what constitutes a correct answer. To simplify this variable, we considered answers to be either correct or incorrect, while guaranteeing that any relevant details obtained from the answers were registered notes taken by the researcher or the audio recording created during the session. For the sake of consistency, we defined criteria used to determine what would constitute a correct or incorrect answer.

For snippets 1, 2, 4, 5, and 9, if a participant correctly identified the vulnerability, we considered the answer to be correct while registering any additional comments and remarks which we believed to be relevant.

For our placebo code, in snippets 3 and 8, participants who identified vulnerabilities and gave an example of how the issue could occur, had their answers considered to be incorrect. We chose these parameters as we assumed that in case the participant took the time to trace the program carefully they would find the program to be clear of any vulnerabilities.

As for our code on solutions to CWE-89 (snippets 6 and 7) and CWE-79 (snippet 10), answers which stated that these solved the issue were considered incorrect only when referring to snippet 6. Answers for the other two code snippets were only considered incorrect if the participants stated that the issue would not be corrected.

For any of the code snippets, we generally looked positively upon answers in which participants mentioned that they would have to consult the documentation to give a definitive answer. We did so believing that, given the necessary resources, these participants would provide an accurate answer.

Some of the data obtained from the eye-tracking device was, by nature, qualitative. This includes data like the locations of fixations of participants which must be carefully analysed.

AOI are crucial in our analysis as they allow us to obtain quantitative data on specific code segments and simplify the evaluation of results obtained from our eye-tracking device. We used these in our analysis to determine the time our participants spent looking at the AOI corresponding to the vulnerable code (B6) in our vulnerable code snippets. Additionally, the relative size of each AOI and the time before looking at them were also considered.

On the individuals' performances during the experiment, we consider that a better result is one in which the user had a higher proportion of correct responses i.e. detected more vulnerabilities in vulnerable code, identified if the solutions to vulnerabilities we presented were correct, and identified that no flaws existed in placebo code (B1). Our participants' response accuracy was also used to create different groups of participants to compare their performances and determine what characteristics led to their differing performances.

The time to detect a vulnerability (B2) was manually measured by identifying how long it took our participants to mention or describe the vulnerabilities they found in the code. In conjunction with the accuracy measure, this allowed us to further evaluate the performance of our participants. This characteristic was also analysed in conjunction with the time it took participants before looking at the locations which we considered to contain the vulnerable code and how long they looked at these parts of the code (B6, B7). Furthermore, B3, the experiment duration, is obtained as the sum of the time our participants spent analysing each code snippet (B4).

Data on the fixations of our participants (B5) is more complex to evaluate than the previous characteristics. This data involves the location and duration of each fixation. We believe fixations to be the most important variables obtained from our eye-tracker and crucial in analysing the thought processes of our participants. Fixation rate and pupil diameter (B8) are two data points which are very important to our study as they have been proven to have a strong correlation with cognitive load and the understanding of code.

Heatmaps are a good method to evaluate which parts of the code received the most attention. These help us quickly see which parts of the code our participants focused on the most. This is done by colouring different regions of the observed code snippets based on the amount of time users spent focusing on each part of the code using the fixation data. In the heatmaps we created, hotter colours indicate longer fixation times, while

cooler colours and no colouring at all are present in regions which received less attention. By enabling us to quickly identify which segments of the code snippets received the most attention, we can determine which participants focused on what parts of the code and compare their gaze patterns with their performances. Other visualisation techniques and the manual analysis of experiment replays were also useful.

For the analyses of our research questions, we also considered the background information variables (BG1-BG5). With this background information, we attempted to identify correlations between the answers or the performance of our participants and their profiles.

CHAPTER 4

Results

In this chapter, we present the results obtained during our study, divided into two parts: the survey findings and those from the eye-tracking experiment. Where the survey findings allow us to address *RQ1*, and the eye-tracking experiment allows us to address *RQ2* and *RQ3*.

A total of twelve individuals of various ages, levels of education, and work experience participated in our study. All participants were industrial cybersecurity professionals actively working in this field. Many of our participants work in secure coding training and cybersecurity training, with some specialising in penetration testing. Participants dedicated an average of 40 minutes to the study, which consisted of a structured introduction, the survey, and the experimental tasks.

The ages of our participants can be seen in Table 9. All participants were above 24 and only one was above 54.

TABLE 9. Number of participants per age group

Less than 25	25-34	35-44	45-54	Over 54
0	8	2	1	1

In terms of education, all participants had a bachelor's degree or higher, with most participants having a master's degree (Table 10).

TABLE 10. Number of participants per educational degree

Bachelor's Degree	Master's Degree	Doctorate
1	7	4

As for their work experience in cybersecurity, our participants had an average of seven years of experience with the participant with the least experience having three and the most experienced one 25. Lastly, ten participants were male while the remaining two were female.

Participants indicated C++ proficiencies between the values one and three out of a possible five points. This was surprising to us, especially as the participants were cybersecurity specialists and some were even trainers for C and C++ coding practices. The average response was 2, corresponding to beginner-level knowledge. To determine why our participants indicated these low values we inquired further, having obtained two main explanations for these results:

- The participants do not actively program in their daily tasks or do not use C++ regularly
- While individuals may be knowledgeable about the syntax of the language, they are not very familiar with or don't use some of its paradigm's concepts such as inheritance or lambda expressions

The experiment we devised consisted of code snippets that did not require advanced knowledge of C++ to be fully comprehended. We believe that basic comprehension of C++ or any similar language was sufficient to not limit the participant's performance when attempting to find cybersecurity vulnerabilities in the code snippets.

Additionally, we instructed participants to describe the behaviour they expected code to have whenever they found a particular instruction or method which they did not comprehend. This way, if a participant struggled with their unfamiliarity with a specific element in the code snippet, they would state what they believed the code would do and proceed with their code analysis based on what they had affirmed. During our experiments, no participants were seemingly limited by their unfamiliarity with the programming language.

4.1. Survey Results

The answers to this question are presented in Figure 6. The professionals we interviewed considered *T1* - code reviewing, to be among the most critical tasks when it comes to reducing cybersecurity vulnerabilities. In the survey, all participants gave code reviews an importance of four or five, out of five. Specifically, 50% of participants considered it very important, with the remaining rating it as important. Additionally, 25% of participants indicated that code reviews are more important than any of the other tasks we presented them with.

Tasks *T2* and *T3* had similar results, being rated four points out of five, on average. While not quite as critical as *T1*, participants considered these tasks important.

The greatest outlier we observed in our survey were the responses on task *T4* - searching online resources (e.g. Stack Overflow or other community-based resources), which respondents considered relatively unimportant. Cybersecurity professionals do not consider these important when attempting to mitigate cybersecurity vulnerabilities, with the average rating being approximately 2.5.

After asking participants why they consider this task to be less important than the others, some mentioned that these websites contain incorrect information or non-standard cybersecurity procedures. Essentially, participants said that these resources have their uses when it comes to finding solutions to programming problems; however, when it comes to questions on cybersecurity, it is better to consult the appropriate documentation or standards to ensure the best practices are followed.

Ten out of our twelve participants responded to the open-ended question, providing answers which mentioned various additional tasks they considered important in mitigating cybersecurity vulnerabilities. Participants indicated tasks such as: defining coding

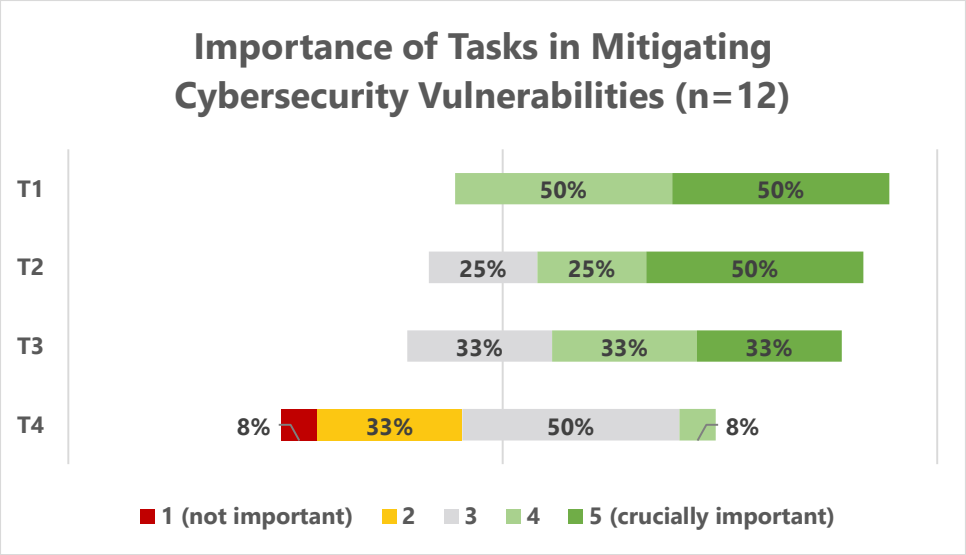


FIGURE 6. Importance given by participants to tasks in mitigating cybersecurity vulnerabilities

guidelines, discussions between colleagues, and unit testing. Two tasks, however, were repeatedly mentioned by multiple participants, underscoring their importance: penetration testing and secure coding training/workshops. One-fourth of the participants mentioned these two tasks.

4.2. Experiment Results

We will present our experiment results in four parts. These correspond to the three, previously mentioned, types of code snippets, namely, code snippets containing one of the top five most common vulnerabilities, code snippets containing proposed solutions to two of these vulnerabilities, and placebo code snippets not containing any vulnerabilities, and a part presenting results which consider the data across all code snippets.

We would like to note that, due to technical issues which we will discuss in chapter 5, the eye-tracking data of one of our participants had to be excluded. Consequently, our analysis of the eye-tracking dataset is based on eleven participants.

For our analysis, we tested the null hypothesis to assess whether the observed correlations were merely coincidental. We considered p-values above 5% to indicate a lack of statistical significance, which aligns with the standard threshold for distinguishing significant from non-significant results [67]. However, we also decided to present some of the results relative to non-statistically relevant correlations we found as, considering our experience and research, we believed these could become more robust when calculated in a larger dataset.

4.2.1. Code Snippets with Vulnerabilities

Our participants' performance when analysing the code snippets containing the weaknesses in the code revealed the data seen in Table 11.

TABLE 11. Results on the analysis of code snippets with cybersecurity vulnerabilities

Vulnerability	CWE-787	CWE-119	CWE-20	CWE-89	CWE-79
Discovery Rate	17%	50%	42%	100%	83%
Avg. Time Analysed (s)	234	289	259	214	242
Avg. Time to Find Vuln. (s)	205	168	137	91	189
Avg. Time on Target (s)	53	100	51	23	36
Avg. Time to Target (s)	11	5	41	37	9
Size of Target (% of screen)	2.27	7.00	3.75	2.29	1.83
Avg. Fixations per Second	4.16	4.08	4.01	3.69	3.86
Avg. Pupil Diameter (PDU[1])	17.19	17.38	17.72	17.66	16.99

The accuracy of responses varied considerably between the code snippets. For instance, in our first snippet of CWE-787, only two users identified the vulnerability in the code, while all users identified the one seen in the CWE-89 code snippet.

CWE-89 and CWE-79 stood out from the rest by being correctly discovered the most, by a considerable margin. These two correspond to SQL injection and XSS respectively and, according to our experience, are some of the most commonly discussed and documented program code weaknesses. Through information gathered during the experiments and additional interviews with our participants, we were led to believe that the fact that these vulnerabilities are so well-known by professionals made them stand out and be easily identifiable.

We compared the number of weaknesses found to the background information we had received from our participants, however, this did not reveal any strong relations. The participants' self-reported C++ proficiency also only presented a weak Pearson correlation to performance with 0.14. Lastly, the educational degree was the only background information which presented even a moderate correlation to performance, even so, its correlation value was only 0.30.

Figure 7 shows the number of participants by the number of vulnerabilities found. To analyse which characteristics or patterns define those participants who performed better, we started by creating two groups of participants: high-performers and all other participants. We chose to consider participants who found more than three weaknesses to be high-performers, with the remaining participants belonging to the "other participants" group. While this analysis yielded some important results, we believed that comparing the performance of just three participants to the rest was less than ideal, and would produce unreliable data. As such we chose to follow a different approach, creating two different groups for each vulnerable code snippet, where one had the participants who discovered the weakness and the other those who did not. Most of the phenomena we had

[1]As mentioned in 3.2, from what we can understand, the pupil diameter's unit of measurement should be in millimetres, however, after some testing we do not believe this to be the case. This is why we chose to use the unique unit of measurement: PDU to describe this characteristic.

observed when using the high-performers/others division were also found with using this new approach.

Figure 8, shows us that participants who didn't find the vulnerabilities, on average, spent more time looking at the code. Note that all participants correctly identified the weakness in the code snippet containing CWE-89.

We created an AOI to assist in interpreting our results. This AOI was called "target" and delimited the sections of the code snippets that we considered to contain the flaw our participants were expected to find. For each of our code snippets the size of the region corresponding to the vulnerable code varied, with some AOI being double the size of others. We consider this important to consider when looking at some of the data relative to the target AOI, as a very strong correlation of 0.92 exists between the size of the AOI and the time participants looked at the target. The null hypotheses for this correlation was also tested, and having obtained a p-value of 2.56%, this is an example of a correlation which is strong enough to be statistically significant.

Furthermore, it is important to mention that the normal inaccuracies of eye-tracking devices, combined with exploratory eye movements, led to the device sometimes registering a fixation on the target even when our participant was not actively reading that part of the code. Exploratory eye movements refer to quick, often unconscious shifts in gaze as participants scan the code without focusing on specific details. These movements are not directly related to code comprehension but are part of the natural visual search process. As a result, this impacted the average time to target metric, as these premature fixations caused the system to record the participants' attention on the target sooner than when they actually started reviewing that section of the code.

While CWE-89 was the fastest to be found by our participants, CWE-79 was one of the code snippets in which participants took the longest to find vulnerabilities. By rewatching the recordings made with the eye-tracking software, we see that our participants usually followed the code's execution path, which, for CWE-79's code snippet, took fairly long before reaching the part of the program containing the flaw. Additionally, this program used some libraries with which the participants were unfamiliar, causing them to take longer to analyse this code and, thus, take longer before analysing the target code.

The average time our participants spent looking at the part of the code containing the vulnerability, i.e. our target, seen in Figure 9, presents a pattern different to the one seen for the total time spent looking at code. Individuals who found the weaknesses spent, on average, more time looking at the target than those who did not.

We compared the heatmaps of our participants to determine if any differences exist in the gaze locations of those who discovered vulnerabilities and those who did not; an example of this can be seen in Figure 10. Using these heatmaps we can confirm what the data had previously hinted to us: participants who discovered the weaknesses spent considerably more time looking at the part of the code corresponding to the target, i.e. the part of the program containing the flaw, highlighted with the yellow rectangle.

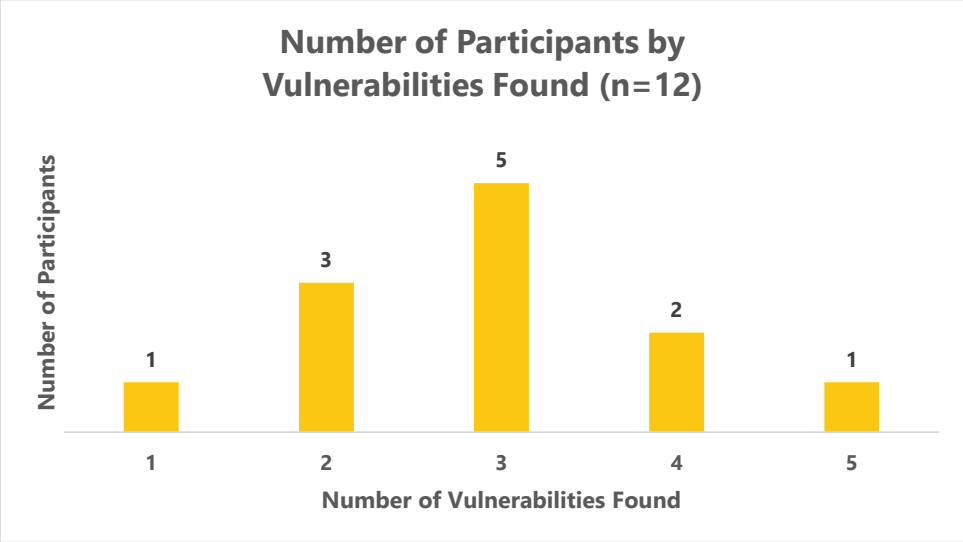


FIGURE 7. Number of participants per number of vulnerabilities found

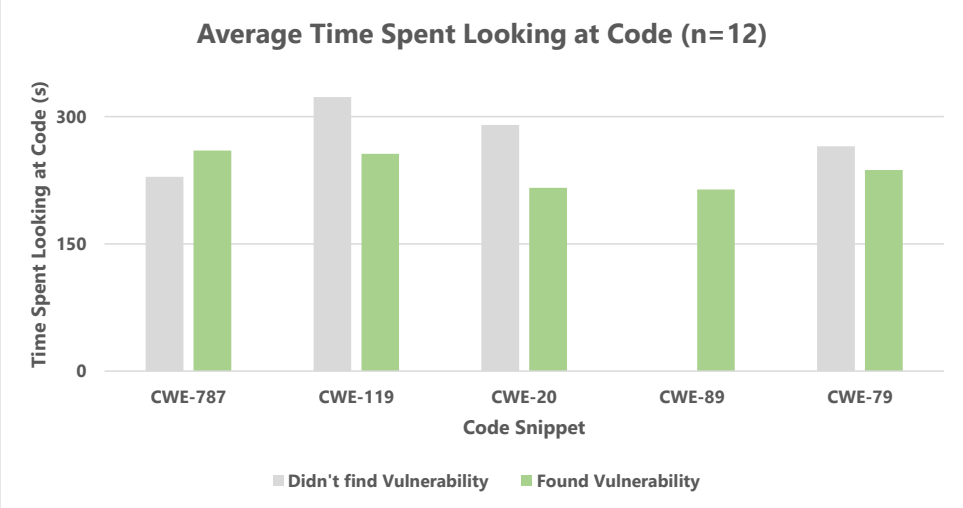


FIGURE 8. Average time participants spent looking at each code snippet containing vulnerabilities

The heatmaps of CWE-89, containing the "SQL Injection" vulnerability, are the clearest example of this difference, this phenomenon was however seen across all code snippets.

We also decided to analyse our participants' gaze paths by rewatching the recordings to determine if any specific strategies were followed which contributed to their success. After carefully rewatching the recordings and taking notes of scan path characteristics, we were not able to discern any notable patterns which distinguished the strategies of successful participants from others.

The average fixation rate of individuals was fairly consistent across the various code snippets, evidenced by averaging the standard deviation of the fixation rates measured by each participant, obtaining a value of 0.44. There is, however, a large difference between the average fixation rates of our participants, with some having fixation rates over three times larger than others, see Table 12.

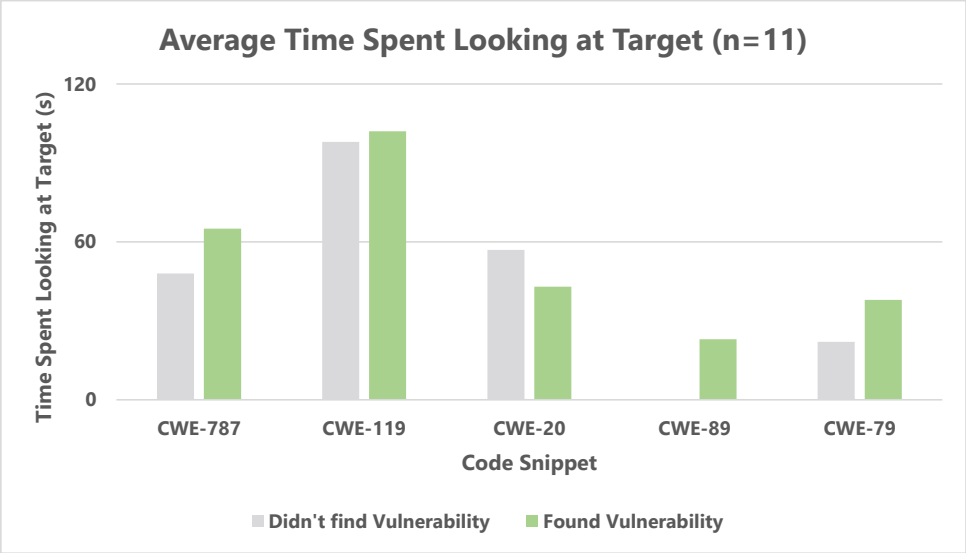
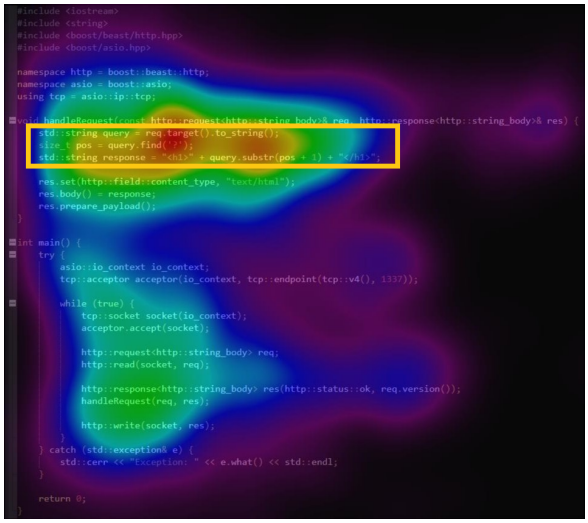
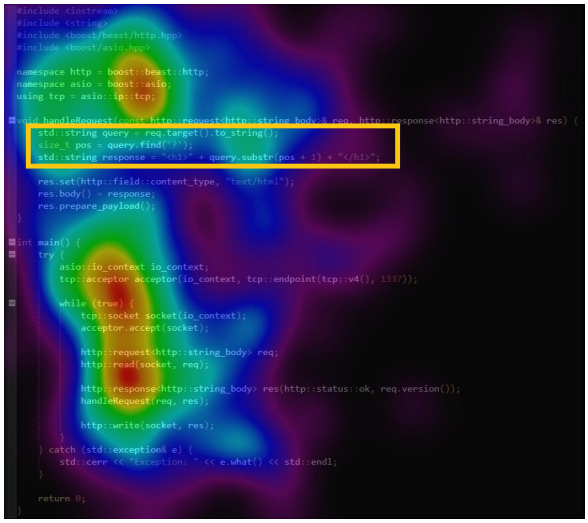


FIGURE 9. Average time participants spent looking at the target in each code snippet



(A) CWE-89 Heatmap - Participants who discovered the vulnerability



(B) CWE-89 Heatmap - Participants who did not discover the vulnerability

FIGURE 10. Heatmaps for Code Snippet of CWE-89

TABLE 12. Average fixation rates of participants (fixations per second)

Average	Standard Deviation	Minimum	Maximum
3.83	1.36	1.73	5.96

We then sought to determine if any notable correlations existed between the fixation rates and participants' performance or background information. In terms of the accuracy of responses, participants who found more vulnerabilities had a lower average fixation rate with a moderate to weak correlation value of -0.36. Figure 11 showcases the relation between the number of vulnerabilities found and the average fixation rate.

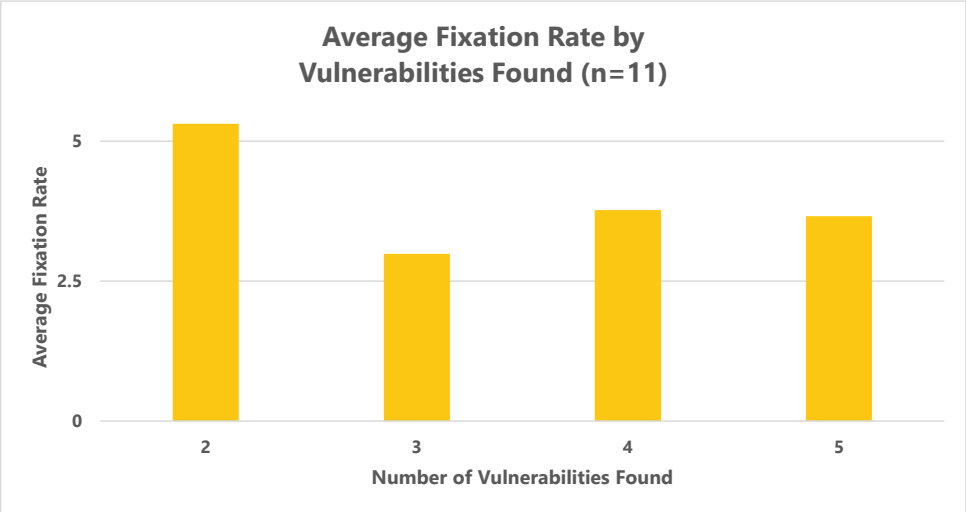


FIGURE 11. Participants' average fixation rate per number of vulnerabilities found

Some statistics on the results we obtained related to pupil diameters can be seen in Table 13. As with the fixation rate, the average dilation of our participants' pupils varied relatively little from code snippet to code snippet, while considerable variability exists between participants. We also see a large difference between the largest and smallest pupil dilation of participants, with some presenting over double the average size of others. As mentioned previously, the values of pupil sizes we present should not be considered as being representative of the true size of our participants' pupils in millimetres, but instead, as a relative measure meant to be used to compare the differences in size across the various code snippets.

Similarly to our analysis of fixation rates, we analysed some of the relations between the various metrics we had registered and our participants' average pupil size. We found a relation between the average pupil size of each participant and the number of weaknesses they found. This correlation has a value of -0.44. This shows us that participants who found more vulnerabilities, on average, had a noticeably smaller average pupil diameter. This correlation is slightly stronger than the one we observed with fixation rate, a characteristic which shares some similarities with pupil dilation. However, this correlation has a p-value of $\approx 17\%$, meaning that any conclusions from this data point should be considered very carefully.

When grouping our participants by the number of vulnerabilities found and obtaining the average pupil diameters of these groups we found a strong correlation of -0.82 with the number of found vulnerabilities. However, when testing the null hypothesis for this value, we obtained a p-value of $\approx 18\%$, indicating that this value may also just be coincidental.

4.2.2. Proposed Solution Code Snippets

Table 14 shows the results of these code snippets with proposed solutions to CWE-89: "SQL Injection" and CWE-79: "Cross-site Scripting".

TABLE 13. Pupil diameters of participants (in PDU)

Average	Standard Deviation	Minimum	Maximum
17.39	3.70	10.74	24.70

TABLE 14. Results on the analysis of proposed solution code snippets

Code Snippet	CWE-89 Inc. Sol.	CWE-89 Cor. Sol.	CWE-79 Solution
Correct Answer Rate	83%	100%	100%
Avg. Time Analysed (s)	70	85	98
Avg. Fixations per Second	3.50	3.69	3.45
Avg. Pupil Diameter (PDU)	17.66	17.74	16.65

CWE-89 - Incorrect Solution was the only proposed solution code snippet which received incorrect answers, with two participants affirming that this solution eliminated the vulnerability. In CWE-89 - Correct Solution, even though we considered all our participants to have correctly indicated that this fixed the issue, many participants noted that they would have to consult the library's documentation to make sure that the problem was indeed fixed.

For CWE-79 - Solution, participants said that they would follow a different type of implementation in a real-world setting. Instead of creating rules to replace certain characters, participants said that one should opt to use libraries designed to sanitize this type of request, as many edge cases may be overlooked with a manual approach. They also stated that the usage of libraries for this type of situation is the industry standard practice.

We observed that these shorter code snippets, which had less than ten lines of code each, also took the least amount of time to be analysed by our participants. The two individuals who incorrectly stated that the CWE-89 - Incorrect Solution resolved the vulnerability, looked at the code for slightly less time than the rest but this difference should not be considered noteworthy, deviating by less than 10% from the average.

These code snippets presenting solutions to the vulnerabilities, presented the lowest average fixation rates out of all code snippets. Other than that, no clear conclusions were obtained from neither the pupil dilation nor the heatmaps obtained from these code snippets.

4.2.3. Placebo Code Snippets

As mentioned before, our experiment included two placebo code snippets, simple programs containing no cybersecurity vulnerabilities. Table 15 shows the results for the placebo code snippets.

Out of all full code snippets i.e. not including the ones presenting solutions to weaknesses, Placebo 1 was analysed for the least amount of time and yet was correctly identified as not containing any flaws by all of our participants. Most participants were successful in determining that Placebo 2 contained no vulnerabilities but took considerably longer

TABLE 15. Results on the analysis of placebo code snippets

Code Snippet	Placebo 1	Placebo 2
Correct Answer Rate	100%	75%
Avg. Time Analysed (s)	190	272
Avg. Fixations per Second	4.16	4.04
Avg. Pupil Diameter (PDU)	17.62	17.81

to analyse the code. While both programs were very similar in size, the use of recursion in the second was, according to our analysis, the main factor which led to participants taking longer to analyse it. Participants would try to follow the program execution path for different inputs to determine if any weaknesses existed and this process would take a considerable amount of time with this recursive code.

In Placebo 1, participants would, at most, indicate that the method to obtain the user input may be unsafe, however, they also stated that they would have to consult documentation which, as with the other code snippets, was seen positively in terms of the evaluation of their performance.

Our results on Placebo 2 were very different as one-fourth of our participants identified a vulnerability or issue, even though no issues existed in our program. Participants found issues related to the user inputs and the usage of the module operation with these inputs, suspecting that issues may exist when these values are 0 or in situations such as when the two inputs are both 1 which, according to some participants, could lead to a stack overflow. However, these issues would not occur when using the program. We also observed that the module operation made some participants hesitant in determining the code execution path.

Our participants who found weaknesses in our placebo code all had some of the highest accuracies when analysing the five code snippets containing vulnerabilities. Specifically, two of these participants found four out of the five vulnerabilities with the last finding all five. When it comes to their fixation rates, these vary a lot, with one participant having a fixation rate much lower than the average, another a very high rate, and the third one being very close to the average value.

Participants who incorrectly identified weaknesses in Placebo 2, which also correspond to our most successful participants, had some of the smallest average pupil diameters. On average these had a pupil diameter of 14.76 while the average pupil diameter for this code snippet was 17.81.

We also compared the heatmaps of the participants who incorrectly identified weaknesses in Placebo 2 with those who did not. While the heatmaps were very similar and both showcased that the main focus point was the recursive function, we noted that those who incorrectly identified flaws focused more of their attention on the lines of code containing input validation. The issues these individuals found were mostly related to and, were they to exist, could be fixed by altering the input validation, which explains the attention they dedicated to this part of the program.

4.2.4. All Code Snippets

Finally, we looked at some characteristics of our participants considering all code snippets. This analysis revealed, among other things, how some of these characteristics changed during the experiment. Table 16 showcases how the total time that participants took to analyse the code snippets varied quite significantly.

These big differences in the total experiment time were investigated to see if they were related to any other characteristics we recorded. The correlation of experiment duration with fixation rate had a value of -0.59, meaning that, on average, lower fixation rates were observed for longer experiment durations. The p-value associated with this correlation was 5.60%, meaning that it is just above the threshold for statistical relevance. Pupil diameter, on the other hand, showed a weaker, but positive correlation to duration with a value of 0.34, but its high p-value makes it hard to consider these results statistically relevant.

The average pupil dilation and fixation rate changed and evolved along the, at times lengthy, extension of the experiment we created. The graphs corresponding to their evolution throughout the experiment can be seen in Figure 12.

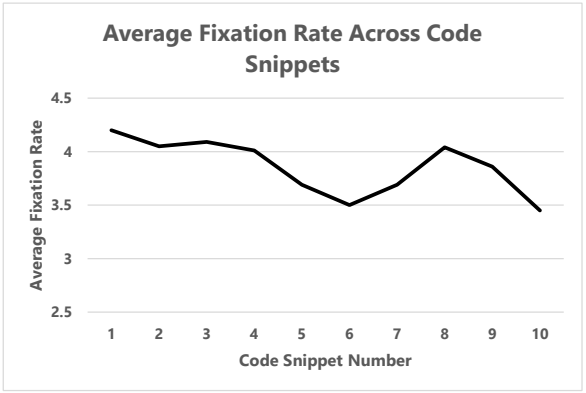
Considering the broad scope of all code snippets we found some additional relationships between the variables we recorded. Fixation rate and pupil diameter are two characteristics which share some unique properties such as providing insight into the cognitive load participants undergo during tasks such as secure code reviews. Despite these similarities they share, the correlation between these two characteristics, while still considerable, is lower than one might anticipate, at only 0.44, which again, considering a high p-value is not statistically significant by itself.

When it came to background information, while still only moderate, the most significant relations regarding the fixation rate were with the age group and the years of experience both of which had correlation values of -0.47 with fixation rate. These correlations had p-values over 12%, meaning that just like with other weaker ones, they could be coincidental.

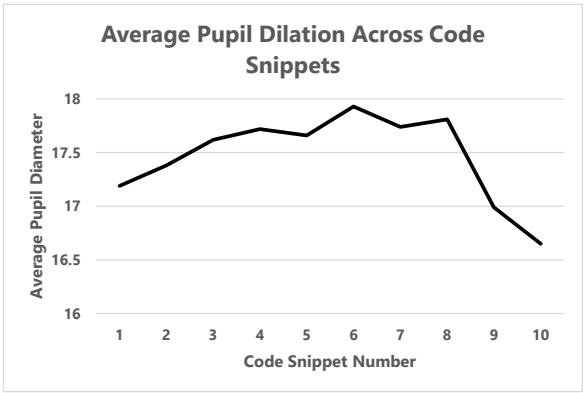
Age, education, and years of experience have strong relations with pupil diameter with correlation values -0.65, -0.59, and -0.54, respectively. We also analysed the p-values for these correlations, finding the values 3.04%, 5.60%, and 8.64%.

TABLE 16. Total Experiment Duration (Minutes:Seconds)

Average	Standard Deviation	Minimum	Maximum
33:27	10:13	17:17	51:56



(A) Timeline of average fixation rate (fixations per second) along the experiment



(B) Timeline of average pupil dilation (PDU) along the experiment

FIGURE 12. Timelines for changes in fixation rate and pupil dilation along the experiment

CHAPTER 5

Discussion

Our work covered different subjects related to analysing user and programmer behaviour using eye-tracking devices in various subjects. The central, and most important, topics we covered were the analysis of industrial cybersecurity professionals' opinions on code reviews and the strategies or techniques they apply when conducting this task with the objective of detecting cybersecurity vulnerabilities.

To structure our research, we defined several research questions, which were answered with our work. These research questions, seen below, served as a guide for the various elements of this project.

RQ1 What tasks in the software development lifecycle do industrial cybersecurity professionals consider to be most crucial in mitigating cybersecurity vulnerabilities?

RQ2 How successful are industrial cybersecurity professionals at conducting secure code reviews?

RQ3 Is there a relation between the patterns revealed using eye-tracking technology and the code reviewers' success in spotting the vulnerabilities?

5.1. Previous Work & Design of Study

We started by conducting a review of previous publications related to our planned work. For this review, we presented the state of the art in cybersecurity for code development, code reviews, and eye-tracking. We also conducted a systematic literature review to analyse recent publications which related studies on human behaviour in the fields of programming, cybersecurity, and user experience using eye-tracking technologies. Furthermore, other relevant publications that were not initially included in the systematic literature review or state of the art were also considered, these were found through unstructured searches or by consulting relevant references in discovered works.

Through the review of previous work, we were able to conclude that the popularity of eye-tracking technology in research is great and has steadily grown, with more and more articles referencing and presenting work using this technology. We found that in 2013 a total of 10000 publications were released, whereas in 2023 alone, close to 40000 were created across the four databases we analysed. The same type of growth pattern is also seen when it comes to publications in the aforementioned fields of study which we chose to focus on, showcasing the interest that exists on this subject.

The analysis of previously conducted work was fundamental in the design of our study. The studies we found, especially those related to programming, often had similar objectives as ours, making these publications quite useful during the design process. All elements

of the design of our survey and experiment were heavily influenced by the literature we analysed.

An essential part of our experiment creation was the software and hardware used to record and analyse the eye-tracking data. Through the review of related work, we found that studies used all sorts of different resources with different advantages and disadvantages. Beyond this, some software tools also had their own publications in which they were presented. We based our choice of eye-tracker, eye-tracking study software, and other resources, on this research.

5.2. RQ1 - What tasks in the software development lifecycle do industrial cybersecurity professionals consider to be the most crucial in mitigating cybersecurity vulnerabilities?

The results of our survey show us that, from the four tasks we presented to our participants, namely: code reviewing, analysis of SAST tool outputs, reading documentation, and researching online resources (e.g. Stack Overflow), participants consistently found code reviewing to be the most important task in mitigating cybersecurity vulnerabilities. The analysis of SAST tool outputs and reading documentation were also seen as very important, even if not quite as much as code reviewing. Regarding the former, it is important to note that participants were asked about the importance they associate with the analysis of SAST tool outputs and not to rate the importance of using such tools. Our objective here was to evaluate the importance these professionals give to different types of tasks which require some attentive human input, and not how important they find the tools involved in these tasks.

Researching online resources, such as Stack Overflow, or other community-based resources was rated to be the least relevant task in mitigating cybersecurity vulnerabilities by a considerable margin. We also asked our participants to justify their responses, this was especially relevant to help us understand the results of this last task. The main explanation for the relatively low importance given to consulting online resources was that these community-based resources may be unreliable and, as such, should not be relied on to resolve cybersecurity weaknesses. Participants noted that the usage of these types of resources is very common and important in the software development process, as they can be used to easily find efficient solutions to common problems. However, when it comes to cybersecurity or other critical tasks and infrastructure, relying on community-based resources is not recommended as they can be unreliable. Instead, the usage of industry-followed resources such as standards and official documentation is encouraged.

Participants were also asked if there were any other tasks which they considered important in mitigating cybersecurity vulnerabilities. From the responses to this open-ended question, two tasks stood out as they were mentioned by a fourth of our participants: secure coding training and penetration testing. Some participants justified their responses and provided explanations as to why they deemed these tasks to be important. Penetration testing was considered essential for identifying weaknesses that might be overlooked

during secure code reviews, providing a real-world assessment of the system's security. As for secure coding training or workshops, these were highlighted because they equip developers with the knowledge and skills to review code thoroughly.

The answers we received, show that participants were considering the question of how to mitigate vulnerabilities from a software development lifecycle or coding perspective. We feel confident in stating this because no participants mentioned risk reduction strategies such as updating software, privilege management, or other top cybersecurity mitigation strategies not directly related to code [68]. This corresponded to the ideal scenario as the development of safe software is the main area which we planned to investigate.

All in all, the survey we conducted shows us that code reviews are highly valued among industrial cybersecurity professionals and essential for effective vulnerability mitigation. The importance of education was also highlighted by the fact that several participants mentioned it in their responses. The analysis of SAST tool outputs and reading documentation was also seen as highly relevant, while the usage of community-based resources is not recommended when it comes to reducing or eliminating cybersecurity vulnerabilities.

5.3. RQ2 - How successful are industrial cybersecurity professionals at conducting secure code reviews?

For *RQ2*, we analysed the performance of participants during the experiment. This mostly took into consideration the answers participants gave and whether we considered them to be correct or not. In our experiment, participants were presented with three types of code snippets: code snippets containing vulnerabilities, proposed solutions to some of these vulnerable code snippets, and placebo code snippets which contained no vulnerabilities.

Five out of the ten code snippets our participants reviewed were code snippets containing some of the most common code-related cybersecurity weaknesses. During the experiment, we found that the most frequently detected vulnerabilities were SQL injections and XSS. We believe that the main factor which explains why these two were found at a significantly higher rate than the other types of weaknesses, was their renown. According to our experience, these are some of the most commonly discussed cybersecurity vulnerabilities. The widespread knowledge of these types of weaknesses may have facilitated their detection by our participants. Both SQL injections and XSS are well known, can be directly linked to several types of attacks, and continue to be commonplace in many software projects or websites. As mentioned in the previous sections, we chose our vulnerabilities based on their number of reported occurrences and, when it comes to this number, XSS is, by a considerable margin, the most frequent. SQL injections also have a high report frequency, according to our data.

Our other vulnerabilities were issues related to memory management, namely, memory allocation and buffer over/underflows. These can be a lot harder to detect as they can require some advanced knowledge of the inner workings of the system's memory management processes to be detected.

An important measure in determining how easily participants find certain weaknesses, besides how often participants correctly identified them, is the time it took participants to find them. The code snippets we used were not ideal for this type of comparative analysis as the differing contexts in which these vulnerabilities exist have influence over the program size, execution path, and complexity. As such, a direct comparison between the average times to find them would not make much sense. There are however some notable results related to the time participants took to analyse the code snippets and find weaknesses. Our two longest programs, by lines of code, were, by a considerable margin, the SQL injection and XSS code snippets. However these were not analysed the longest, in fact, SQL injection was analysed for the least amount of time and participants found the weakness, on average, almost one minute sooner than for any of the other vulnerabilities. These insights can be valuable in estimating how long code review sessions should take, depending on the complexity and familiarity of the vulnerabilities involved.

Through our analysis of the gaze data, we concluded that two factors made it so, even for these very large programs, participants found the vulnerabilities very quickly: first, participants were well acquainted with these two types of weaknesses, since, as we just mentioned, these are very commonly discussed issues. Second, and this was seen when analysing our gaze data, the code execution path has a large impact on the time it takes participants to find the vulnerabilities as this usually coincided with their code scan path. The execution path of the SQL injection code snippet would quickly lead our participants to the vulnerable part of the code while, for the XSS code, this would take much longer.

We believed it to be warranted to compare our results to the performance of SAST tools when it comes to identifying these weaknesses. Research has been conducted to compare and analyse the performance of SAST tools. While this research didn't focus on the same types of vulnerabilities we considered, it did show us that SAST tools may be able to find flaws that manual code reviews might miss, complementing them nicely.

In this research, it was found that, while experts excelled at detecting both XSS and SQL injection vulnerabilities, automated tools seemed to struggle with the detection of injection vulnerabilities. SAST tools also struggled with flaws related to improper adherence to coding standards which we believe experienced professionals would easily detect [69–71]. On the other hand, these code analysis tools performed well in the detection of memory-related issues which our participants had struggled with. Research has also included analysis and comparisons to new analysis techniques using large language model (LLM)s. While these techniques seem to exceed the performance of traditional SAST tools, they present the same limitations when, for instance, detecting injection vulnerabilities [72].

For a conclusive comparative analysis between SAST tools and manual secure code reviews, research involving several SAST tools and new techniques such as LLMs, additional participants, and more code snippets representing different types of vulnerabilities should be conducted. We believe that this topic, the comparison between manual code reviews

and automated review tools, warrants further research and has the potential to provide important, scientifically based, insights on the performance of experts and automated tools in code reviews.

Participants were very successful in determining if the proposed solutions we presented to them did, or did not, eliminate the issues they were supposed to correct. The only incorrect answers were given by two participants who said that the incorrect solution to the SQL injection vulnerability fixed this issue when, in fact, it did not. The proposed solutions for this vulnerability relied on the use of prepared statements, also known as parameterized queries. When prepared statements aren't used, as seen in our vulnerable code snippet, user inputs are simply inserted into string values that form the query, making it possible for individuals to gain unintended access to the database. If implemented correctly, prepared statements treat user inputs as parameters in the queries, ensuring that they are not interpreted as SQL instructions.

In the incorrect solution we present, the prepared statement is created with a string that already contains the user inputs, this means that the inputs are not parameterised. According to our experience, this is a common mistake developers often make, and for this reason, we decided to include it in our experiment. While many developers may instinctively know that prepared statements are the solution to this kind of issue, they often fail to implement them correctly, in fact, all participants acknowledged that prepared statements are a solution to SQL injection weaknesses. Even experienced professionals, on occasion, commit this mistake.

The solution we presented to our participants for the XSS vulnerability was a small sanitisation program which altered escape characters present in the string to encoded versions of these characters. While our solution did remove this weakness, most of our participants noted that this approach is not ideal or standard practice in combating this specific type of issue. Participants stated that sanitisation libraries should be used in this scenario, as there are many edge cases which a hand-made solution such as this one may overlook. These answers and comments on the implementation showcase our experts' knowledge and experience on the subject.

The two placebo code snippets we included in the code consisted of small, relatively simple programs which did not contain any cybersecurity weaknesses. While all participants stated that the first placebo code did not contain any issues, one-fourth of our participants incorrectly identified a vulnerability in the second. The participants who incorrectly identified a weakness in this second program were also the ones who had the highest success rate with the code snippets containing vulnerabilities. We believe that this may indicate that individuals who were more prone to finding weaknesses were also more inclined to detect them when they weren't present, as was evidenced by these results. Furthermore, the additional complexity due to the recursiveness of the program made it all the more difficult to analyse, especially as many of the participants were not accustomed to recursive programs.

5.4. RQ3 - Is there a relation between the patterns revealed using eye-tracking technology and the code reviewers' success in spotting the vulnerabilities?

For *RQ3*, our in-depth analysis explains what factors differentiated the performances of our participants and if we can derive any patterns associated with our participants' behaviour when analysing code in search of security vulnerabilities. For this analysis, we considered the participants' eye gaze data as well as the background information we had collected.

In our analysis, we focused on comparing participants with correct responses to those with incorrect ones, in the case of the vulnerable code snippets, this meant comparing those who found the weaknesses to those who did not. We compared several different characteristics, including the time participants spent looking at the code and, specifically, at the part containing the vulnerability. We found that, compared to other participants, those who found the weakness spent less total time looking at the code while also spending more time looking at the parts of the code containing the vulnerability i.e. the target.

These results were visually demonstrated by the heatmaps we generated. Through these, we determined that, for the vulnerable code snippets, participants who correctly identified weaknesses had a seemingly more focused approach to looking at the code. We found that these participants had heatmaps with smaller, and more intense gaze hotspots around the target. These results made us believe that individuals with more knowledge of the weaknesses are quicker to find them as they look at the program more efficiently. We found that the most successful participants focused most of their attention on areas involving input validation or query construction which, according to our experience, are common hotspots for vulnerabilities, and also often coincided with the target we had defined.

It should be noted that, in our experiment, participants were allowed and encouraged to look at the code snippets for as long as necessary to analyse them thoroughly. Considering this, it is possible to assume that participants who found the vulnerabilities may have been more eager to move on to the following code snippets. Furthermore, after identifying these flaws, participants would describe what they had found and why they considered it to represent a weakness. This may explain why they also spent more time looking at the target.

The analysis of the gaze paths of our participants when reading the code was also of great importance. Previous work has found differences between the gaze paths of experts and those of novices. Specifically, research has found that experts follow the main methods execution path more often than novices [33]. We were unfortunately not able to follow up on this research as not all code snippets had the structure necessary for this type of analysis and, for the few code snippets that did, we found that participants, in most cases, followed the main method whether they found the vulnerability or not.

We analysed various characteristics of our participants to identify any correlations with their performance and presented many of the important results we found. In some cases, these correlations were not statistically significant, as determined after analysing their p-values. This was mostly due to the small sample size, as moderate and even stronger correlations tended to have high p-values. However, we included some of these correlations as, per our experience, we believe they could remain relevant with a larger dataset in which they would be considered statistically significant.

Eye-tracking characteristics such as fixation rate and pupil dilation of our participants were analysed. These two have often been shown to have a strong correlation to cognitive load, specifically, higher values are related to higher cognitive load [12, 13] and, naturally, these characteristics have a notable correlation between themselves. Some of these studies even incorporated cerebral imaging techniques alongside eye-tracking, to accurately map how cognitive loads influence pupil dilation and other characteristics [73]. While one may assume that a higher cognitive load is good as it can be indicative of a higher degree of involvement with the code, excessive cognitive workload may lead to less efficient code interpretation, or be a sign that a participant is struggling with the interpretation of the code snippets.

We observed that the average pupil size and fixation rate decreased throughout the experiment. This may be a sign that towards the end of the experiment, our participants were less involved in the code reviewing process. While the results, in terms of response accuracy, do not indicate that our participants' performance was impacted, this result can be indicative of the effect that longer secure code review tasks have on the engagement of reviewers when analysing code.

While our participants' fixation rates remained relatively consistent throughout the experiment, they varied significantly between the participants themselves. We found a small correlation indicating that participants with lower average fixation rates had a higher success rate in the experiment but, as with any of these weaker correlations, this should be considered very carefully as with the current dataset they cannot be considered statistically relevant.

Pupil dilation, which shares many characteristics with fixation rate, had a slightly stronger relationship with the success rate of participants. We also found that participants with lower average pupil diameters were, on average, more successful in finding vulnerabilities. Something to note, however, is that this measure of the pupil diameter may be influenced by several factors which makes a direct comparison between the values more difficult. Factors such as the size of the individual's eyes and proximity to the device may have had a great influence on this value. Another factor may be the average amount of light in the room which, while we attempted to limit the amount of natural light, we were not able to control completely, which could have further affected pupil diameters.

When it comes to the proposed solution code snippets, an analysis of the eye-tracking data on these was not very insightful for our analysis. These code snippets were all

quite succinct which made an in-depth analysis of the various eye-tracking characteristics not reveal any clear patterns. We did, however, observe that these presented the lowest average fixation rates out of all code snippets. Drawing a parallel to the notion that fixation rate is related to cognitive load, we consider that this may be related to the lower amount of information that participants have to process when analysing these short code fragments.

The analysis of eye-tracking data was, however, quite relevant in the analysis of results on the placebo code snippets. As previously mentioned, our participants were very successful in determining that these code snippets did not contain any weaknesses. Our first placebo code snippet was analysed for less time than any other comparable code snippet, yet it was correctly identified as not containing any flaws by all of our participants. This hints that this code snippet was simple to interpret and, as such, participants were very confident in moving on to the next code snippet without finding any vulnerabilities. Our second code snippet, consisting of a program which obtained the highest common factor between two values using recursion, was, on average, analysed for some of the longest time out of all code snippets.

The eye-tracking data we recorded showed that its recursiveness was one of the main factors which led our participants to spend a lot reviewing this code. The analysis of recursive programs is a lot more complex than a program using standard iterative structures as it adds a layer of complexity which many individuals were not very familiar with. The effect of this program's complexity is evidenced in our results. When reviewing the eye-tracking data, we found that participants spent a disproportional amount of time analysing the part of the code with the recursive method, attempting to follow the code execution with several different input values to determine if a weakness exists. We believe that, if this program were non-recursive, our participants would have been able to analyse it more easily and might not have mistakenly identified some vulnerabilities.

We also explored the data to find if the background characteristics of our participants were related to their performance. We found these characteristics to not present any significant additional information in terms of which participants performed better during the experiment. The strongest relation we observed was with the educational degree of our participants, and even so, this relation was relatively weak and considered statistically insignificant given our dataset.

5.5. Threats to Validity

An important threat to the validity of our study is the limited number of participants in our study. Since we exclusively invited industrial cybersecurity experts to participate, this limited the number of participants we had access to and could invite to our study, restricting our analysis and limiting our conclusions of both the survey and experiment data.

Additionally, although we tried to create a suitable environment for the eye-tracking experiment to the best of our abilities, we encountered some issues due to our equipment's

sensitivity to several factors. Some elements we tried to control included: limiting the amount of light present in the room, creating a natural environment for our participants, and following special instructions for users who, for example, use glasses. When adopting these measures we closely followed the instructions in the device manufacturers' manual [66]. The issues we encountered led to some of the recorded data not being reliable enough to be used. Furthermore, the static nature of the code snippets used in the experiment may not have fully captured the dynamic, real-world environments in which secure code reviews typically occur, which may also influenced the results

Despite these facts, the results are actively being used internally in the company to guide the improvement of training. We believe the presented results reflect our own experience in the industry and present valuable insights both to academia and to industrial practitioners.

CHAPTER 6

Conclusions

Cybersecurity has become an ever-growing concern as industrial systems grow increasingly interconnected and exposed to sophisticated cyber threats. The importance of securing these systems from malicious attacks is paramount, particularly as vulnerabilities in software can lead to severe operational, financial, and reputational damages. In response to these growing challenges, this study focused on improving cybersecurity practices by exploring how industrial cybersecurity professionals conduct secure code reviews, a critical component in detecting and eliminating vulnerabilities in program code.

To achieve these goals, we designed a study, aimed at understanding the cognitive processes of experts during code reviews as well as their thoughts and opinions on this task. The study was divided into two parts: a survey to gauge expert opinions on critical tasks in vulnerability mitigation and an experiment in which participants reviewed code snippets while their eye movements were tracked. The study involved twelve cybersecurity professionals with varying levels of expertise. With the collected data we were able to analyse their opinions on code reviews as well as the strategies and techniques they apply when conducting code reviews to detect cybersecurity issues, and through this, provide actionable insights for improving future training and tool development.

Our findings shed light on several critical aspects of how secure code reviews are conducted in the context of industrial cybersecurity. First and foremost, the survey results highlighted that code reviews are widely regarded as an indispensable tool in the software development lifecycle for finding and reducing vulnerabilities. Participants consistently ranked code reviews among the most crucial tasks for mitigating cybersecurity vulnerabilities, reinforcing their value in secure coding practices.

The experiment itself revealed valuable insights into how experts approach the identification of different types of vulnerabilities. Familiarity played a significant role, as participants were far more adept at identifying widely-discussed vulnerabilities such as SQL injection and XSS. These well-documented and frequently encountered vulnerabilities, were often detected quickly and accurately. In contrast, memory-related vulnerabilities such as buffer overflows (CWE-787) posed a greater challenge. These findings suggest that less discussed vulnerabilities require more time and focus, making them more prone to oversight.

Eye-tracking technology proved instrumental in understanding how experts scan and analyse code during reviews. The eye-tracking device provided us with data related to the participants' cognitive focus, revealing to us what characteristics correlate to successful secure code reviews. For instance, those who consistently focused on critical areas of the

code, such as input validation or query construction, were more likely to identify security issues. This suggests that targeted training that emphasizes vulnerability "hotspots" in the code could lead to more efficient code reviews and improve overall vulnerability detection rates.

Some of the characteristics we analysed included the time participants spent reviewing each code snippet. This allowed us to gain insights into the time required to identify certain vulnerabilities. These insights can be valuable in estimating how long secure code review sessions should take depending on the complexity and familiarity of the vulnerabilities involved. This data can help organizations in planning more effective and time-efficient code review processes.

The findings of this research have practical implications for both industry practices and cybersecurity training programs. First, the study underscores the importance of continuous exposure to a diverse range of vulnerabilities during training. The difficulties participants were faced with when identifying buffer overflows and other memory-related issues highlight the need for more focused training on less familiar, yet dangerous, vulnerabilities. This approach could enhance the readiness of cybersecurity professionals to deal with a wider array of security threats.

While the study provides valuable insights, it is not without its limitations. One of which is the relatively small sample size. With only twelve participants, the results may not be fully representative of the broader industry. Expanding the participant pool in future research could provide a more comprehensive understanding of secure code review behaviours.

Technical limitations also played a role in the study. Eye-tracking devices, such as the one we used, can be quite sensitive to diverse factors, this influenced the accuracy of some of the data. While these technical issues were minimised as much as possible, they nonetheless highlight the challenges inherent in conducting eye-tracking experiments. Furthermore, the static nature of the code snippets used in the experiment may not have fully captured the dynamic, real-world environments in which secure code reviews typically occur.

Building on the foundation of this study, several avenues for future research emerge. For instance, expanding the participant pool to obtain more reliable conclusions. Also, further research could be conducted into less common vulnerabilities, such as memory corruption, concurrency issues, or code snippets could be included which answer previously highlighted research questions such as which types of vulnerabilities take the longest to be discovered.

The results also emphasise the potential for improving tools used during code reviews. By incorporating insights from eye-tracking data, systems could be developed that guide reviewers' attention to the most critical areas of the code. For example, tools could be designed to highlight regions of the code where vulnerabilities are most likely to occur, thereby reducing cognitive load and improving the accuracy of reviews. Although previous

work in this specific area has begun to emerge [74], we believe that there is still great potential for further research in this field.

Another promising area of exploration would be the comparison of the performance of human code reviewers to that of automated tools. While automated code analysis tools are commonly used in the industry, there is still a gap in understanding how they perform in comparison to manual reviews conducted by experienced professionals. This research could provide critical insights into whether a combination of automated and human reviews offers the most effective approach to vulnerability detection.

Finally, future studies could examine a wider range of code-related tasks beyond secure code reviews, such as debugging or refactoring, to identify if insights gained from this research can be applied to other aspects of software development. By continuing to explore these areas, we can further enhance the role cybersecurity professionals have in safeguarding industrial systems against cyber threats.

References

[1] Federal Cyber Security Authority. 'The state of IT security in Germany in 2023.' Accessed: Mar. 25, 2024. (2023), [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Securitysituation/IT-Security-Situation-in-Germany-2023.pdf.

[2] M. S. John, B. Swanston and A. Chatterjee, *Cybersecurity stats: Facts and figures you should know*, Accessed: Sep 22, 2024, Aug. 2024. [Online]. Available: https://www.forbes.com/advisor/education/it-and-tech/cybersecurity-statistics.

[3] Common Weakness Enumeration, *CVE → CWE mapping "root cause mapping" guidance*, Accessed: Jan 22, 2024, Mar. 2024. [Online]. Available: https://cwe.mitre.org/documents/cwe_usage/guidance.html.

[4] A. Bacchelli and C. Bird, 'Expectations, outcomes, and challenges of modern code review,' in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, ISBN: 9781467330763. DOI: 10.1109/ICSE.2013.6606617.

[5] Z. Sharafi, Z. Soh and Y. G. Guéhéneuc, 'A systematic literature review on the usage of eye-tracking in software engineering,' *Information and Software Technology*, vol. 67, pp. 79–107, Nov. 2015, ISSN: 0950-5849. DOI: 10.1016/J.INFSOF.2015.06.008.

[6] U. Obaidellah, M. A. Haek and P. C. Cheng, 'A survey on the usage of eye-tracking in computer programming,' *ACM Computing Surveys*, vol. 51, 1 Jan. 2018, ISSN: 15577341. DOI: 10.1145/3145904.

[7] K. Kevic, B. M. Walters, T. R. Shaffer, B. Sharif, D. C. Shepherd and T. Fritz, 'Tracing software developers' eyes and interactions for change tasks,' in *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015*, Association for Computing Machinery, Inc, Aug. 2015, pp. 202–213, ISBN: 9781450336758. DOI: 10.1145/2786805.2786864.

[8] Gazepoint, *Applications & history of eye tracking technology*, Accessed: Mar. 25, 2024. [Online]. Available: https://www.gazept.com/eye-tracking.

[9] M. Q. Khan and S. Lee, 'Gaze and eye tracking: Techniques and applications in adas,' *Sensors (Switzerland)*, vol. 19, 24 Dec. 2019, ISSN: 14248220. DOI: 10.3390/s19245540.

[10] S. Riegel Correia, M. Pinto-Albuquerque, T. Espinha Gasiba and A.-C. Iosif, 'Improving Industrial Cybersecurity Training: Insights into Code Reviews Using

Eye-Tracking,' in *5th International Computer Programming Education Conference - ICPEC 2024*, A. L. Santos and M. Pinto-Albuquerque, Eds., ser. Open Access Series in Informatics (OASIcs), vol. 122, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 17:1–17:9, ISBN: 978-3-95977-347-8. DOI: 10.4230/OASIcs.ICPEC.2024.17. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.ICPEC.2024.17.

[11] C. S. Peterson, J. Saddler, T. Blascheck and B. Sharif, 'Visually analyzing students' gaze on C++ code snippets,' in *IEEE/ACM 6th International Workshop on Eye Movements in Programming, EMIP 2019*, Institute of Electrical and Electronics Engineers Inc., May 2019, pp. 18–25, ISBN: 9781728122434. DOI: 10.1109/EMIP.2019.00011.

[12] Z. Sharafi, Y. Huang, K. Leach and W. Weimer, 'Toward an objective measure of developers' cognitive activities,' *ACM Transactions on Software Engineering and Methodology*, vol. 30, 3 May 2021, ISSN: 15577392. DOI: 10.1145/3434643.

[13] S. D. Aljehane, B. Sharif and J. I. Maletic, 'Studying developer eye movements to measure cognitive workload and visual effort for expertise assessment,' in *Proceedings of the ACM on Human-Computer Interaction*, vol. 7, Association for Computing Machinery, May 2023. DOI: 10.1145/3591135.

[14] NIST, 'NIST special publication 800-53 revision 5 security and privacy controls for information systems and organization,' National Institute of Standards and Technology, standard, Sep. 2020, Accessed: Sep 22, 2024. DOI: 10.6028/NIST.SP.800-53r5. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-53r5.

[15] PCI SSC, 'Payment card industry data security standard, version 4.0.1,' Payment Card Industry Security Standards Council, standard, Jun. 2024.

[16] PCI Policy Portal, *Payment brands data security & pci dss standards - amex, visa, mastercard, discover*, Accessed: Sep 28, 2024. [Online]. Available: https://pcipolicyportal.com/what-is-pci/payment-brands.

[17] OWASP, 'Application security verification standard 4.0.,' The Open Worldwide Application Security Project, standard, Oct. 2021.

[18] IEC, 'Security for industrial automation and control systems - part 4-1: Secure product development lifecycle requirements,' International Electrical Commission, Standard, 2018.

[19] IEC, 'Security for industrial automation and control systems - part 4-2: Technical security requirements for iacs components,' International Electrical Commission, Standard, 2019.

[20] Tobii, *Global leader in eye tracking for over 20 years*, Accessed: Jan. 23, 2024. [Online]. Available: https://www.tobii.com/.

[21] Y. Abdrabou *et al.* 'Revealing the hidden effects of phishing emails: An analysis of eye and mouse movements in email sorting tasks.' Accessed: Jan. 11, 2024. (May 2023), [Online]. Available: http://arxiv.org/abs/2305.17044.

[22] T. H. C. T. da Silva, M. D. Cavalcanti, J. Pessoa and B. V. Becker, 'Developing a system for graphical analysis of brainwaves during media consumption,' in *II Concurso de Trabalhos de Iniciação Científica (CTIC 2022)*, 2022.

[23] M. I. Ibrahim, R. A. Latif and A. M. Kamal, 'The effects of background music on the screen-based reading material among university students: An eye tracking study,' *Journal of Cognitive Sciences and Human Development*, vol. 9, pp. 117–132, 2 Sep. 2023, ISSN: 2550-1623. DOI: 10.33736/jcshd.5933.2023.

[24] H. Jin, Y. Liu, X. Mu, M. Ma and J. Zhang, 'Usability evaluation and improvement of mission planner UAV ground control system's interface,' *International Journal of Performability Engineering*, vol. 15, pp. 2726–2734, 10 2019, ISSN: 09731318. DOI: 10.23940/ijpe.19.10.p19.27262734.

[25] Z. Zhang, D. Chang, J. Zhang and R. Ding, 'Eye tracking-based usability evaluation of e-government app icon design,' in *2021 IEEE International Conference on Industrial Engineering and Engineering Management*, Institute of Electrical and Electronics Engineers Inc., 2021, pp. 1651–1655, ISBN: 9781665437714. DOI: 10.1109/IEEM50564.2021.9672784.

[26] A. H. Kusumo and M. Hartono, 'The evaluation of academic website using eye tracker and UEQ: A case study in a website of xyz,' in *IOP Conference Series: Materials Science and Engineering*, vol. 703, IOP Publishing Ltd, Dec. 2019. DOI: 10.1088/1757-899X/703/1/012049.

[27] A. L. Santos, 'Javardeye: Gaze input for cursor control in a structured editor,' in *Companion Proceedings of the 5th International Conference on the Art, Science, and Engineering of Programming*, ser. Programming '21, Cambridge, United Kingdom: Association for Computing Machinery, 2021, pp. 31–35, ISBN: 9781450389860. DOI: 10.1145/3464432.3464435.

[28] T. H. Yang, J. Y. Huang, P. H. Han and Y. P. Hung, 'Saw it or triggered it: Exploring the threshold of implicit and explicit interaction for eye-tracking technique in virtual reality,' in *2021 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops, VRW 2021*, Institute of Electrical and Electronics Engineers Inc., Mar. 2021, pp. 482–483, ISBN: 9780738113678. DOI: 10.1109/VRW52623.2021.00123.

[29] N. Peitek *et al.*, 'Correlates of programmer efficacy and their link to experience: A combined EEG and eye-tracking study,' in *ESEC/FSE 2022 - Proceedings of the 30th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Association for Computing Machinery, Inc, Nov. 2022, pp. 120–131, ISBN: 9781450394130. DOI: 10.1145/3540250.3549084.

[30] Z. Gao, T. Wang, M. Wang and Y. Zhang, 'UX testing of developer documentation - a pilot study of oceanbase database documentation,' in *2023 IEEE International Professional Communication Conference (ProComm)*, 2023, pp. 64–72. DOI: 10.1109/ProComm57838.2023.00035.

[31] N. Al Madi, 'How readable is model-generated code? examining readability and visual inspection of github copilot,' in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22, Rochester, MI, USA: Association for Computing Machinery, 2023, ISBN: 9781450394758. DOI: 10.1145/3551349.3560438.

[32] M. Villamor and M. M. Rodrigo, 'Predicting successful collaboration in a pair programming eye tracking experiment,' in *Adjunct Publication of the 26th Conference on User Modeling, Adaptation and Personalization*, ser. UMAP '18, Singapore, Singapore: Association for Computing Machinery, 2018, pp. 263–268, ISBN: 9781450357845. DOI: 10.1145/3213586.3225234.

[33] T. Busjahn, Simon and J. H. Paterson, 'Looking at the main method – an educator's perspective,' in *Proceedings of the 21st Koli Calling International Conference on Computing Education Research*, ser. Koli Calling '21, Joensuu, Finland: Association for Computing Machinery, 2021, ISBN: 9781450384889. DOI: 10.1145/3488042.3488068.

[34] Z. Sharafi, I. Bertram, M. Flanagan and W. Weimer, 'Eyes on code: A study on developers' code navigation strategies,' *IEEE Transactions on Software Engineering*, vol. 48, pp. 1692–1704, 5 May 2022, ISSN: 19393520. DOI: 10.1109/TSE.2020.3032064.

[35] S. Becker, A. Obersteiner and A. Dreher, 'Eye tracking – promising method for analyzing mathematics teachers' assessment competencies?' In *2022 Symposium on Eye Tracking Research and Applications*, ser. ETRA '22, Seattle, WA, USA: Association for Computing Machinery, 2022, ISBN: 9781450392525. DOI: 10.1145/3517031.3529244.

[36] I. McChesney and R. Bond, 'Eye tracking analysis of code layout, crowding and dyslexia - an open data set,' in *ACM Symposium on Eye Tracking Research and Applications*, ser. ETRA '21 Short Papers, Virtual Event, Germany: Association for Computing Machinery, 2021, ISBN: 9781450383455. DOI: 10.1145/3448018.3457420.

[37] P. L. Gorski, S. Möller, S. Wiefling and L. L. Iacono, ''I just looked for the solution!'on integrating security-relevant information in non-security API documentation to support secure coding practices,' *IEEE Transactions on Software Engineering*, vol. 48, pp. 3467–3484, 9 Sep. 2022, ISSN: 19393520. DOI: 10.1109/TSE.2021.3094171.

[38] D. Davis and F. Zhu, 'Understanding and improving secure coding behavior with eye tracking methodologies,' in *Proceedings of the 2020 ACM Southeast Conference*, ser. ACM SE '20, Tampa, FL, USA: Association for Computing Machinery, 2020, pp. 107–114, ISBN: 9781450371056. DOI: 10.1145/3374135.3385293.

[39] Common Vulnerabilities and Exposures, *CVE security vulnerability database. security vulnerabilities, exploits, references and more*, Accessed: Mar 24, 2024. [Online]. Available: https://www.cvedetails.com/.

[40] L. Bernard, S. Raina, B. Taylor and S. Kaza, 'Minimizing cognitive load in cyber learning materials – an eye tracking study,' in *ACM Symposium on Eye Tracking Research and Applications*, ser. ETRA '21 Short Papers, Virtual Event, Germany: Association for Computing Machinery, 2021, ISBN: 9781450383455. DOI: 10.1145/3448018.3458617.

[41] M. Madleňák and K. Kampová, 'Eye-tracking system as a part of the phishing training,' in *2023 21st International Conference on Emerging eLearning Technologies and Applications (ICETA)*, 2023, pp. 359–364. DOI: 10.1109/ICETA61311.2023.10343937.

[42] D. K. Davis and F. Zhu, 'Analysis of software developers' coding behavior: A survey of visualization analysis techniques using eye trackers,' *Computers in Human Behavior Reports*, vol. 7, Aug. 2022, ISSN: 24519588. DOI: 10.1016/j.chbr.2022.100213.

[43] Z. Sharafi, B. Sharif, Y. G. Guéhéneuc, A. Begel, R. Bednarik and M. Crosby, 'A practical guide on conducting eye tracking studies in software engineering,' *Empirical Software Engineering*, vol. 25, pp. 3128–3174, 5 Sep. 2020, ISSN: 15737616. DOI: 10.1007/s10664-020-09829-4.

[44] D. T. Guarnera, C. A. Bryant, A. Mishra, J. I. Maletic and B. Sharif, 'Itrace: Eye tracking infrastructure for development environments,' in *Eye Tracking Research and Applications Symposium (ETRA)*, Association for Computing Machinery, Jun. 2018, ISBN: 9781450357067. DOI: 10.1145/3204493.3208343.

[45] Microsoft, *Visual studio: IDE and code editor for software developers and teams*, Accessed: October 22, 2023. [Online]. Available: https://visualstudio.microsoft.com/.

[46] Eclipse Foundation, *Eclipse IDE | the eclipse foundation*, Accessed: October 22, 2023. [Online]. Available: https://eclipseide.org/.

[47] J. Behler, G. Chiudioni, A. Ely, J. Pangonis, B. Sharif and J. I. Maletic, 'Itrace-visualize: Visualizing eye-tracking data for software engineering studies,' in *2023 IEEE Working Conference on Software Visualization (VISSOFT)*, 2023, pp. 100–104. DOI: 10.1109/VISSOFT60811.2023.00021.

[48] J. Behler, P. Weston, D. T. Guarnera, B. Sharif and J. I. Maletic, 'Itrace-toolkit: A pipeline for analyzing eye-tracking data of software engineering studies,' in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2023, pp. 46–50. DOI: 10.1109/ICSE-Companion58688.2023.00022.

[49] U. Obaidellah, T. Blascheck, D. T. Guarnera and J. Maletic, 'A fine-grained assessment on novice programmers' gaze patterns on pseudocode problems,' in *Eye*

Tracking Research and Applications Symposium (ETRA), Association for Computing Machinery, Feb. 2020, ISBN: 9781450371346. DOI: 10.1145/3379156.3391982.

[50] Y. Huang, K. Leach, Z. Sharafi, N. McKay, T. Santander and W. Weimer, 'Biases and differences in code review using medical imaging and eye-tracking: Genders, humans, and machines,' in *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Association for Computing Machinery, Inc, Nov. 2020, pp. 456–468, ISBN: 9781450370431. DOI: 10.1145/3368089.3409681.

[51] Tobii, *Eye tracking software for behavior research*, Accessed: Feb 21, 2024. [Online]. Available: https://www.tobii.com/products/software/behavior-research-software/tobii-pro-lab.

[52] Gazepoint, *Analysis professional eye-tracking software*, Accessed: Feb 21, 2024. [Online]. Available: https://www.gazept.com/product/gazepoint-analysis-professional-edition-software/?v=3a52f3c22ed6.

[53] OGAMA, *OGAMA - open gaze and mouse analyzer*, Accessed: Feb 21, 2024. [Online]. Available: http://www.ogama.net/.

[54] A. Vosskühler, V. Nordmeier, L. Kuchinke and A. Jacobs, 'OGAMA (open gaze and mouse analyzer): Open-source software designed to analyze eye and mouse movements in slideshow study designs,' *Behavior research methods*, vol. 40, pp. 1150–1162, Mar. 2008. DOI: 10.3758/BRM.40.4.1150.

[55] R. Likert, 'A technique for the measurement of attitudes,' *Archives of Psychology*, vol. 140, pp. 1–55, 1932.

[56] Microsoft, *Microsoft forms | surveys, polls, and quizzes*, Accessed: Feb 27, 2024. [Online]. Available: https://forms.office.com.

[57] Gazepoint, *Gazepoint analysis user manual*, 2023.

[58] Gazepoint, *Gazepoint api manual*, Accessed: Feb. 24, 2024, 2022. [Online]. Available: https://www.gazept.com/dl/Gazepoint_API_v2.0.pdf.

[59] Common Weakness Enumeration, *2023 CWE top 25 most dangerous software weaknesses*, Accessed: Jul 22, 2024. [Online]. Available: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html.

[60] Common Weakness Enumeration, *CWE-787: Out-of-bounds write*, Accessed: Mar 26, 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/787.html.

[61] C. Conikee, *Seeve: A set of vulnerable C code snippets (with mapped CVEs)*, Accessed: Mar 26, 2024. [Online]. Available: https://github.com/conikeec/seeve.

[62] Common Weakness Enumeration, *CWE-89: Improper neutralization of special elements used in an SQL command (SQL injection)*, Accessed: Mar 26, 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/89.html.

[63] Yes We Hack, *Vulnerable-code-snippets: Twitter vulnerable snippets*, Accessed: Mar 26, 2024. [Online]. Available: https://github.com/yeswehack/vulnerable-code-snippets.

[64] Programiz, *C++ Examples | Programiz*, Accessed: Mar 26, 2024. [Online]. Available: https://www.programiz.com/cpp-programming/examples.

[65] E. A. Locke and G. P. Latham, 'Building a practically useful theory of goal setting and task motivation: A 35-year odyssey,' *American Psychologist*, vol. 57, no. 9, pp. 705–717, 2002.

[66] Gazepoint, *Gazepoint control user manual*, 2023.

[67] G. D. Leo and F. Sardanelli, 'Statistical significance: P value, 0.05 threshold, and applications to radiomics—reasons for a conservative approach,' *European Radiology Experimental*, vol. 4, 1 Dec. 2020, ISSN: 25099280. DOI: 10.1186/s41747-020-0145-y.

[68] National Security Agency, *NSA's top ten cybersecurity mitigation strategies*, Accessed: Aug. 7, 2024, Mar. 2018. [Online]. Available: https://www.nsa.gov/portals/75/documents/what-we-do/cybersecurity/professional-resources/csi-nsas-top10-cybersecurity-mitigation-strategies.pdf.

[69] K. Li *et al.*, 'Comparison and evaluation on static application security testing (SAST) tools for java,' in *ESEC/FSE 2023 - Proceedings of the 31st ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Association for Computing Machinery, Inc, Nov. 2023, pp. 921–933. DOI: 10.1145/3611643.3616262.

[70] S. Elder *et al.*, *Do I really need all this work to find vulnerabilities? an empirical case study comparing vulnerability detection techniques on a java application*, Accessed: Sep. 10, 2024, Aug. 2022. arXiv: 2208.01595 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2208.01595.

[71] S. Matěj, 'Evaluation and application of SAST tools,' Bachelor's Thesis, Masaryk University, Faculty of Informatics, 2024.

[72] X. Zhou *et al.*, *Comparison of static application security testing tools and large language models for repo-level vulnerability detection*, Accessed: Sep 18, 2024, Jul. 2024. arXiv: 2407.16235 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2407.16235.

[73] B. P. Bailey and S. T. Iqbal, 'Understanding changes in mental workload during execution of goal-directed tasks and its application for interruption management,' *ACM Transactions on Computer-Human Interaction*, vol. 14, 4 Jan. 2008, ISSN: 10730516. DOI: 10.1145/1314683.1314689.

[74] W. Saranpää *et al.*, 'Gander: A platform for exploration of gaze-driven assistance in code review,' in *Proceedings of the 2023 Symposium on Eye Tracking Research and Applications*, ser. ETRA '23, Tubingen, Germany: Association for Computing Machinery, 2023. DOI: 10.1145/3588015.3589191.

APPENDIX A

Survey

Below you will find a PDF version of the survey presented to participants. This survey was originally created in Microsoft Forms and was presented to participants through the online response portal.

Visualization in SW dev Tasks in Cybersecurity

April, 2024

This is a short survey to determine how industry professionals classify visually intensive software development life cycle tasks in terms of their importance in mitigating cybersecurity vulnerabilities.

* Required

Disclaimer

By responding to this questionnaire, which is completely anonymous, you consent to your data being processed by the project developers for academic purposes. At no stage of the project will your data be shared with third parties.

You can choose not to respond to this questionnaire at any time, simply closing it to cancel your participation.
Data related to canceled participations will not be stored.

Personal details

What is your age? *

- ◯ Under 18
- ◯ 18-24
- ◯ 25-34
- ◯ 35-44
- ◯ 45-54
- ◯ Over 55

What is your gender?

- ◯ Woman
- ◯ Man
- ◯ Non-binary
- ◯ Prefer not to say
- ◯ Other

What is the highest level of education you have completed? *

- ◯ High School Diploma
- ◯ Vocational Education
- ◯ Bachelor Degree
- ◯ Masters Degree
- ◯ Doctorate
- ◯ Other

How many years of work experience do you have in the field of cybersecurity? *

Please enter a number greater than or equal to 0

On a scale from 1 (Novice) to 5 (Expert), how proficient do you consider yourself in C++?

1	2	3	4	5

Survey

We plan to analyse visually intensive software development tasks, as these types of tasks give us insight into how a developer thinks by analysing what he looks at.

On a scale from 1 (not important) to 5 (crucially important), how important are the following software development tasks in mitigating cybersecurity vulnerabilities? *

	1	2	3	4	5
Code reviewing	◯	◯	◯	◯	◯
Analysis of code review tool outputs	◯	◯	◯	◯	◯
Reading documentation	◯	◯	◯	◯	◯
esearching online resources (e.g. Stack Overflow or other community-based resources)	◯	◯	◯	◯	◯

Which additional software development tasks which you consider important in mitigating cybersecurity vulnerabilities? You may also provide a rating from 1-5 to indicate how important you consider these tasks.

APPENDIX B

Experiment Guide

Below you will find the guide used for the execution of the study's experiments, in its integrity.

Eye-tracking Study

Procedure Guide

Introduction

- Greet.
- Introduce yourself.
- Introduce the subject which we wish to investigate, in a way that does not influence the study's results in some undesired way. Also, guarantee their consent for the recording of the experiment:
    - *"We are conducting this study to investigate how people review code when searching for cybersecurity vulnerabilities. You will be asked to look at some code, analyse it and determine if it contains any vulnerabilities. It is important to note that you will not be personally evaluated or tested during this study."*
    - *To conduct this study, we will use an eye-tracking device which will register your eye movements, we will also record the screen and audio. The recordings and all information obtained from this experiment will be anonymised, and only be used in the context of my work, so it will not be passed on to others. Are you okay with us recording the session for the previously mentioned purposes?"*
- Present the subject with the questionnaire and let them answer it on their own.
    - https://forms.office.com/r/8T1SDp6jr7

Experiment

- At this moment the subject should have been introduced to the proceedings, have answered the questionnaire, and should be seated in the position from which they will conduct the test.
- Adjust the desk, chair, eye-tracker, and monitor positions to best suit the subject and begin the calibration process.
- Remind the user of the task and give some additional information.
    - *"You will now analyse a series of code snippets written in c++, with the objective of discovering any potential cybersecurity flaws. Whenever you believe to have discovered a cybersecurity vulnerability let me know. Also, let me know when you find that a code snippet does not contain any vulnerabilities".*
    - *"Again, you are not being evaluated or tested and you should not rush yourself, simply look at the code as you would normally when scanning for potential vulnerabilities."*
    - *"Do you have any questions?"*
    - *"Are you ready to begin?"*
- Initiate the recording in OGAMA, assuring that the correct screen is selected for the recording.
- When the subject discovers a vulnerability:
    1. If it's the correct vulnerability say that they found the vulnerability, we were looking for and ask if they want to keep analysing the code or if they are ready to move on to the next task.

2. If it's the incorrect vulnerability, acknowledge their answer without indicating if it is correct or incorrect, make a note of the subject's answer, and ask if they want to continue to analyse the code or if they would like to move on.
3. If the subject says that the code does not contain any vulnerabilities, move on to the following snippet.

Independently of it being the vulnerability being correctly identified, ask for some rationale on why they believe the code contains a vulnerability.

2. SNIPPETS:
    1) *"A program which, given a string, trims all trailing white spaces."*
        - CWE-787 Out-of-bounds write
        - If a string is completely made of spaces this causes a buffer underwrite. Can access and remove memory outside of scope.
        - Main vulnerability:
            - while (isspace(message[len]))
    2) *"Program which takes an input string and changes some characters to different ones for sanitation purposes."*
        - CWE-119 Improper Restriction of Operations within the Bounds of a Memory Buffer
        - String with a lot of '&' or '<' will overflow. Also, when a very small string is given, it will present other data stored in memory.
        - Main problematic line:
            - dst_buf[dst_index++] = '
                - This without checking for total size of array.
    3) *"This program counts and displays the number of times a certain character is present in a string."*
        - Placebo 1
    4) *"Simple program which declares a two-dimensional board."*
        - CWE-20 Improper Input Validation
        - The user-inputted value is not checked to see if it is a negative value, as such we can use large negative numbers to reserve more memory than expected.
        - Main problematic lines:
            - if (m > MAX_DIM || n > MAX_DIM)
            - struct board_square_t*)malloc(m * n * sizeof(struct board_square_t)
    5) *"This program obtains all information on a user from a MySQL database given his username."*
        - CWE-89 SQL Injection
        - User input should be sanitized.
        - Main problematic line:
            - std::string query = "SELECT * FROM users WHERE username = '" + username + "'";
    6) *"In the previous program, this instruction presents a risk of SQL Injection"*
        - CWE-89 SQL Injection – Detail
    7) *"This is how the vulnerability was tackled."*
        - CWE-89 SQL Injection - First solution

        - Problem:
            - Incorrect implementation of prepared statements, same issue as in the original program.
    8) *"This was the following attempt at mitigating the vulnerability".*
        - CWE-89 SQL Injection - Second solution
        - No issues.
    9) *"Program for finding the highest common factor (HCF), using recursion."*
        - Placebo 2
    10) *"This program consists of a small website which receives requests and displays a simple HTML page."*
        - CWE-79 Cross-site scripting
        - Gets 'q' from 'param' without sanitization or validation.
        - Main problematic line:
            - std::string response = "<h1>" + query.substr(pos + 1) + "</h1>";
    11) *"In the previous program, this instruction presents a risk of Cross-Site Scripting"*
        - CWE-79 Cross-site scripting – Detail
    12) *"This is an attempt made at mitigating the vulnerability."*
        - CWE-79 Cross-site scripting – Solution
        - No issues.
3. The OGAMA recording will automatically finish once all slides are presented.
4. Stop the audio recording.

Debriefing

- Ask the subject some final questions and obtain feedback.
    - *"How confident do you feel about the vulnerabilities you discovered?"*
    - *"Do you have anything you'd like to add or something you'd like to ask?"*

APPENDIX C

Script to find number of occurrences of vulnerabilities

The following Python script was created to find the number of registered occurrences of vulnerabilities on CVEDetails.

```
import requests
from bs4 import BeautifulSoup
import re

def get_vulnerabilities_count(url):
    try:
        # Define headers to mimic a Firefox browser
        headers = {
            'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64
               ; rv:97.0) Gecko/20100101 Firefox/97.0'
        }

        # Fetch HTML content from the URL with headers
        response = requests.get(url, headers=headers)
        if response.status_code == 200:
            # Parse HTML using BeautifulSoup
            soup = BeautifulSoup(response.content, 'html.parser')

            # Find all text containing "XXX vulnerabilities found"
            vulnerabilities_text = soup.find_all(string=re.compile(
               r'\d+vulnerabilities found'))

            if vulnerabilities_text:
                # Extract the number of vulnerabilities
                vulnerabilities_count = int(re.search(r'\d+',
                   vulnerabilities_text[0]).group())
                return vulnerabilities_count
            else:
                return None
        else:
            print("Failed to retrieve data from the URL.
                Status code:", response.status_code)
```

```
            return None
    except Exception as e:
        print("An error occurred:", e)
        return None

# Example usage:
for ii in range(1300):
    url = f"https://www.cvedetails.com/vulnerability-list/cwe{ii
       +1}/vulnerabilities.html"
    # Replace with the desired URL
    vulnerabilities_count = get_vulnerabilities_count(url)
    if vulnerabilities_count is not None:
        print(f"CWE-{ii+1}:", vulnerabilities_count)
    else:
        print(f"CWE-{ii+1}: 0")
```

APPENDIX D

Code Snippets

In this part, you will find the code snippets our participants were tasked with analysing during the experiment. For our code snippets containing vulnerabilities, we have highlighted the vulnerable part of the code, which we referred to as the "target", in red.

D.1. Snippet 1 - CWE-787: "Out-of-bounds Write"

```
#include <iostream>
#include <cstring>

char* trimTrailingWhitespace(char *strMessage, int length) {
char *retMessage;
char *message = new char[length + 1];

// copy input string to a temporary string
std::memcpy(message, strMessage, length);
message[length] = '\0';

// trim trailing whitespace
int len = length - 1;
while (isspace(message[len])) {
    message[len] = '\0';
    len--;
}

// return string without trailing whitespace
retMessage = message;
return retMessage;
}

int main() {
    char str[] = "Test   ";
    char *trimmed = trimTrailingWhitespace(str, strlen(str));
    std::cout << "Trimmed string: " << trimmed << std::endl;
    delete[] trimmed; // Free dynamically allocated memory
    return 0;
}
```

D.2. Snippet 2 - CWE-119: "Improper Restriction of Operations within the Bounds of a Memory Buffer"

```
#include <iostream>
#include <cstring>
#include <cstdlib>

#define MAX_SIZE 16

char* copy_input(const char* user_supplied_string) {
    int dst_index = 0;

    char* dst_buf = new char[4 * MAX_SIZE];
    // Iterate over the user-supplied string
    for (int i = 0; i < strlen(user_supplied_string); i++) {
        if (user_supplied_string[i] == '&') {
            dst_buf[dst_index++] = '&';
            dst_buf[dst_index++] = 'a';
            dst_buf[dst_index++] = 'm';
            dst_buf[dst_index++] = 'p';
            dst_buf[dst_index++] = ';';
        } else if (user_supplied_string[i] == '<') {
            dst_buf[dst_index++] = '&';
            dst_buf[dst_index++] = 'l';
            dst_buf[dst_index++] = 't';
        } else {

            dst_buf[dst_index++] = user_supplied_string[i];
        }
    }
    return dst_buf;
}

int main() {
    char uss[MAX_SIZE]; // Buffer for user input
    read(0, uss, MAX_SIZE); // Read user input
    char* dst_buff = copy_input(uss); // Process user input
    printf("%s", dst_buff); // Output processed input
    delete[] dst_buff; // Free allocated memory
    return 0;
}
```

D.3. Snippet 3 - Placebo 1

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

int main()
{
    string str;
    cout << "Enter a string: ";
    getline(cin, str); // Read user input into str

    char checkCharacter;
    cout << "Enter the character to count: ";
    cin >> checkCharacter; // Read the character to count

    int count = 0;

    for (int i = 0; i < str.size(); i++)
    {
        if (str[i] ==  checkCharacter)
        {
            ++count;
        }
    }

    cout << "Number of " << checkCharacter << " = " << count;

    return 0;
}
```

D.4. Snippet 4 - CWE-20: "Improper Input Validation"

```
#include <iostream>
#include <cstdlib>

#define MAX_DIM 100

struct board_square_t {
    int height;
    int width;
};
```

```
int main() {
    /* board dimensions.*/
    int m, n, error;
    struct board_square_t* board;

    std::cout << "Please specify the board height: " << std::endl;
    error = std::scanf("%d", &m);
    if (EOF == error) {
        std::cout << "No integer passed: Die evil hacker!" << std::
           endl;
    }

    std::cout << "Please specify the board width: " << std::endl;
    error = std::scanf("%d", &n);
    if (EOF == error) {
        std::cout << "No integer passed: Die evil hacker!" << std::
           endl;
    }

    if (m > MAX_DIM || n > MAX_DIM) {
        std::cout << "Value too large: Die evil hacker!" << std::
           endl;
    }

    board = (struct board_square_t*)malloc(m * n * sizeof(struct
       board_square_t));

    return 0;
}
```

D.5. Snippet 5 - CWE-89: "SQL Injection"

```
#include <iostream>
#include <string>
#include <mysql_driver.h>
#include <mysql_connection.h>

int main() {
    try {
        sql::mysql::MySQL_Driver *driver;
        sql::Connection *con;
```

```
        // Create a MySQL Driver object
        driver = sql::mysql::get_mysql_driver_instance();

        // Connect to the MySQL database
        con = driver->connect("tcp://127.0.0.1:3306", "root", "
           password");

        // Select the database schema
        con->setSchema("example");

        std::string username;
        std::cout << "Enter username: ";
        std::getline(std::cin, username);

        // Execute the plain query with the provided username
        sql::Statement *stmt;
        sql::ResultSet *res;

        stmt = con->createStatement();
        res = stmt->executeQuery("SELECT * FROM users WHERE
           username = '" + username + "'");

        // Process the result set
        while (res->next()) {
            std::cout << "User found: " << res->getString("username
               ") << " - " << res->getString("email") << std::endl;
        }

        delete res;
        delete stmt;
        delete con;
    } catch (sql::SQLException &e) {
        std::cerr << "SQL Error: " << e.what() << std::endl;
    }

    return 0;
}
```

D.6. Snippet 6 - CWE-89: Incorrect Solution

```
prepareStatement("SELECT * FROM users WHERE username = '" +
   username + "');
```

```
res = stmt->executeQuery();
```

D.7. Snippet 7 - CWE-89: Correct Solution

```
stmt = con->prepareStatement("SELECT * FROM users WHERE username =
   ?");
stmt->setString(1, username);
res = stmt->executeQuery();
```

D.8. Snippet 8 - Placebo 2

```
#include <iostream>

using namespace std;

int hcf(int n1, int n2);

int main()
{
        int n1, n2;
        const int MAX_VALUE = 1000; // Maximum allowed value

        cout << "Enter two positive integers (maximum value " <<
           MAX_VALUE << "): ";
        cin >> n1 >> n2;

        if (n1 > MAX_VALUE || n2 > MAX_VALUE || n1 < 0 || n2 < 0) {
                cout << "Input values must be positive and not
                   exceed " << MAX_VALUE << ". Please try again."
                   << endl;
                return 1; // Exit with error code
    }

        cout << "H.C.F of " << n1 << " & " <<  n2 << " is: " << hcf
           (n1, n2);

        return 0;
}

int hcf(int n1, int n2){
    if (n2 != 0)
       return hcf(n2, n1 % n2);
```

```
    else
        return n1;
}
```

D.9. Snippet 9 - CWE-79: "Cross-site Scripting"

```
#include <iostream>
#include <string>
#include <boost/beast/http.hpp>
#include <boost/asio.hpp>

namespace http = boost::beast::http;
namespace asio = boost::asio;
using tcp = asio::ip::tcp;

void handleRequest(const http::request<http::string_body>& req,
   http::response<http::string_body>& res) {

    std::string query = req.target().to_string();
    size_t pos = query.find('?');
    std::string response = "<h1>" + query.substr(pos + 1) + "</h1>"
        ;

    res.set(http::field::content_type, "text/html");
    res.body() = response;
    res.prepare_payload();
}

int main() {
    try {
        asio::io_context io_context;
        tcp::acceptor acceptor(io_context, tcp::endpoint(tcp::v4(),
            1337));

        while (true) {
            tcp::socket socket(io_context);
            acceptor.accept(socket);

            http::request<http::string_body> req;
            http::read(socket, req);
```

```
            http::response<http::string_body> res(http::status::ok,
                req.version());
            handleRequest(req, res);

            http::write(socket, res);
        }
    } catch (std::exception& e) {
        std::cerr << "Exception: " << e.what() << std::endl;
    }

    return 0;
}
```

D.10. Snippet 10 - CWE-79: Solution

```
#include <boost/algorithm/string.hpp>
...
std::string userInput = query.substr(pos + 1);
boost::algorithm::replace_all(userInput, "&", "&amp;");
boost::algorithm::replace_all(userInput, "\"", "&quot;");
boost::algorithm::replace_all(userInput, "'", "&#39;");
boost::algorithm::replace_all(userInput, "<", "&lt;");
boost::algorithm::replace_all(userInput, ">", "&gt;");
std::string response = "<h1>" + userInput + "</h1>";
```