THE HUMAN DIMENSION OF SOFTWARE SECURITY AND FACTORS AFFECTING SECURITY PROCESSES

by

Hala Assal

A thesis submitted to
the Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

School of Computer Science

at

CARLETON UNIVERSITY

Ottawa, Ontario

August, 2018

© Copyright by Hala Assal, 2018

Abstract

Usable security for software developers is a research direction that is in its early stages. Even though developers typically have technical expertise, they are not necessarily security experts and need support when dealing with security. This thesis focuses on the human aspect of software security within the overall development process. The research employs mixed methods, including Cognitive Walkthrough studies, interviews, and an online survey study. We started by studying usability issues in code analysis tools, and designed a visual analysis environment to support collaboration between team members and exploration during security analysis of source code. However, while working on this project, we recognized that the software security problem is a larger one, relating to the overall process of integrating security in the Software Development Lifecycle. Thus, through 13 interviews and an online survey with 123 software developers, we explored real-life software security practices, how developers acquire security knowledge, and the motivators and deterrents to software security. Based on our empirical studies, we identified recommendations that can help support developers handle security throughout the Software Development Lifecycle.

Our qualitative and quantitative analyses showed varying approaches to software security, and clear discrepancies between existing and best practices. Through exploring developers' motivations towards software security, we identified both extrinsic and intrinsic motivations. We found that acting towards software security volitionally and for reasons extending beyond mandates can lead to better security processes and better developer-engagement in these processes. Particularly, our studies showed that when the different entities involved in the Software Development Lifecycle communicate and collaborate, and when security is perceived as a common and shared responsibility, this can positively influence software security, *e.g.*, by promoting internal motivations which are associated with improved engagement and cognitive abilities. Towards promoting the internalization of software security, we proposed a human-oriented model to describe how external software security motivations can be internalized. Our model highlights the interplay between security knowledge, team collaboration, and internal motivations to security.

Acknowledgements

Working on this thesis was a wonderful journey with its fair share of ups and downs. I would like to express my gratitude to all those who have supported me throughout this journey.

To my thesis supervisor, Sonia Chiasson, for her continuous support and guidance, and for her insights that helped elevate the quality of this research. Thank you Sonia for always being there, especially in the many sleepless nights before deadlines, and for being a friend, besides being a thesis supervisor.

Thanks to the members of my committee, Heather Lipford, Timothy Lethbridge, Alejandro Ramirez, and Robert Biddle whose expertise, guidance, and feedback helped shape this thesis. I would especially like to thank Robert for his close mentoring, enthusiasm during our many discussions, and support from the early years of my PhD and throughout my journey.

Thanks to participants in our studies who have volunteered their valuable time to share their experiences and offer their insights. Your contributions are key to this work. Also, thanks to my colleagues at the CHORUS lab for their insights and feedback, as well as all the fun discussions we had at the lab.

I would also like to express my deepest thanks to my family for their love and support throughout my studies. To my parents, Fouad and Khairia, for always pushing me to go the extra mile and being the best that I could be. To Walid and Nadia, my wonderful brother and sister, for all their love and understanding. To my in laws for always believing in me and cheering me on. To my husband AbdelRahman, to whom I cannot even begin to express my gratitude. Thank you for all the proofreading, discussions, and insights. Thank you for always being there, keeping my spirits up, and for your unwavering love and motivational support. And finally, thank you my wonderful son Hamza for being so understanding during all the times that mummy had to work and for pretending to work on a thesis while I worked on mine.

Table of Contents

Abstract ii

Acknowledgements iii

List of Tables ix

List of Figures x

List of Acronyms xii

Chapter 1 Introduction 1

1.1 Scope 1

1.2 Motivation 2

1.3 Research Question 4

1.4 Contributions 4

1.5 Thesis Outline 6

1.6 Related Publications 7

Chapter 2 Background and Related Work 8

2.1 Software Engineering 8

2.2 Security Initiatives 9

2.3 Human Factors in Software Engineering 11

2.3.1 Factors Influencing Developers' Practices 11

2.3.2 Motivations for Conducting Code Reviews 12

2.3.3 Reasons for Use and Under-Use of Static-code Analysis Tools . 13

2.4 Human Factors in Software Security 16

2.4.1 Security Tool Adoption 17

2.4.2 Developers' Abilities and Expertise 19

2.4.3 Improving and Introducing New Security Tools and Methodologies . . . 20
2.5 Software Visualizations . . . 22
2.6 Research Gap Analysis . . . 25
2.7 Background on Activity Theory and Self-Determination Theory . . . 25
2.7.1 Activity Theory . . . 26
2.7.2 Self-Determination Theory . . . 27

Chapter 3 Visual Representation of Source Code Vulnerabilities 30
3.1 Using the Cognitive Dimensions Framework for Usability Evaluation . 31
3.2 Using the Cognitive Walkthrough Methodology for Usability Evaluation 33
3.3 FindBugs' Study . . . 34
3.3.1 Study Design . . . 34
3.3.2 Results . . . 36
3.4 Cesar . . . 38
3.5 Cesar's Study . . . 41
3.5.1 Study Design . . . 41
3.5.2 Cesar's Strengths . . . 43
3.6 Future Enhancements . . . 47
3.7 Discussion . . . 50
3.8 Limitations . . . 52
3.9 Summary . . . 53

Chapter 4 Security in the Software Development Lifecycle 54
4.1 Study Design and Methodology . . . 55
4.1.1 Interview Study Design . . . 55
4.1.2 Participant Demographics . . . 55
4.1.3 Analysis . . . 56
4.1.4 Limitations . . . 58
4.2 Results: Security in Practice . . . 59
4.2.1 Exploring Practices by Development Stage . . . 61

4.2.2 The adopters vs. the Inattentive 75
4.3 Software Security Best Practices 78
4.4 Interpretation of Results 80
4.4.1 Current Practices versus Best Practices 80
4.4.2 Factors Affecting Security Practices 81
4.4.3 Future Research Directions 84
4.5 Conclusion 85

Chapter 5 Security Knowledge and Motivation **86**
5.1 Using Grounded Theory for Analysis 87
5.1.1 Researcher Bias 89
5.2 Knowledge Acquisition Taxonomy 89
5.2.1 Formal Learning 92
5.2.2 Semi-Formal Learning 95
5.2.3 Informal Learning 98
5.2.4 Insights Based on the Taxonomy 100
5.2.5 Additional Use for the Knowledge Acquisition Taxonomy . . . 103
5.3 Motivation for Software Security 105
5.3.1 Amotivation 107
5.3.2 Intrinsic and Extrinsic Motivations 109
5.4 Internalizing Software Security 111
5.5 Summary 115

Chapter 6 Survey **116**
6.1 Survey Methodology 116
6.1.1 Survey Design 117
6.1.2 Testing the Survey Tool 118
6.1.3 Participant Recruitment 119
6.1.4 Data Quality 119
6.1.5 Participant Demographics 121
6.2 Survey Analysis 121

6.2.1 Addressing the Research Questions 121
6.2.2 Factor Analysis 122
6.2.3 Developers' Work Motivation 123
6.2.4 Developers' Mental Models of Software Security 123
6.3 Security in the Software Development Lifecycle 124
6.3.1 Efforts Towards Security 124
6.3.2 Behaviours and Attitudes 125
6.3.3 Experiencing Security Issues 127
6.3.4 Strategies to Address Software Security 129
6.4 Motivators and Deterrents to Security 132
6.4.1 Software Security Motivators 132
6.4.2 Deterrents to Software Security 136
6.5 Effect of Different Characteristics on Software Security 139
6.5.1 Development Methodology 140
6.5.2 Company Size 142
6.5.3 Test-Driven Development 142
6.6 Discussion 143
6.6.1 RQ1: How Does Security Fit in the Development Lifecycle in Real Life? 144
6.6.2 RQ2: What are The Current Motivators and Deterrents to Developers Paying Attention to Security? 145
6.6.3 RQ3: Does the Development Methodology, Company Size, or Adopting Test-Driven Development Influence Software Security? 146
6.7 Limitations 146
6.8 Conclusion 147

Chapter 7 Discussion, Future work, and Conclusions 148

7.1 Thesis Contributions 148
7.2 Insights on Conducting Studies with Developers 150

7.3 Answering the Thesis Research Question: Recommendations for Supporting Developers . . . 152
7.4 Future Research Directions . . . 156
7.5 Conclusion . . . 158

Bibliography **159**

Appendix A Interview Script **175**

Appendix B Motivations and Amotivations for Software Security **177**

Appendix C Developers' Survey **184**

Appendix D Types of Software Developed by Survey Participants **194**

List of Tables

4.1 Qualitative study participant demographics 56
4.2 The degree of security in the SDLC. 60
4.3 Summary of themes emerging from the qualitative analysis. . . 76

5.1 Knowledge Acquisition Taxonomy 90
5.2 Distribution of participants mentioning learning opportunities fitting in each cell of the Knowledge Acquisition Taxonomy . . 92

6.1 Summary of survey study participant demographics 120
6.2 Summary of statistical tests for the survey study 121
6.3 Within subject statistical analysis comparing security efforts in SDLC stages 126
6.4 Number of participants indicating that security is important and that their software is an interesting target for attackers 127
6.5 Factor analysis for software security strategies 131
6.6 Factor analysis for motivation 134
6.7 Factor analysis for security deterrents 137
6.8 Between subject statistical analysis of the effect of development methodology, company size, and adopting TDD on software security 141

B.1 Motivations and Amotivations of software security 178

List of Figures

2.1 Microsoft Security Development Lifecycle (SDL) 10
2.2 Example of a call graph visualization 23
2.3 Example visualization for class dependency 23
2.4 Engeström's triangle 26
2.5 Third generation activity theory with two interacting activity systems 27
2.6 The self-determination continuum 28

3.1 Screenshot of FindBugs interface 35
3.2 The setup for usability studies in Chapter 3 36
3.3 Cesar's treemap visualizing select defect categories 40
3.4 Cesar's visualization, details, and source code panes. 41
3.5 Example for a secondary visualization for Cesar 48
3.6 The relation between Cognitive Dimensions and four select objectives of a Collaborative Security Code Review (CSCR) tool. 50

4.1 Security adopters: developer testing abstraction 58

5.1 Axial coding process for interview data. 88
5.2 Analyzing motivations and amotivations for software security . 105
5.3 The self-determination continuum of software security 107
5.4 Internalizing software security model 112

6.1 Explanation differentiating between software security and security functions in the survey. 118
6.2 Software security efforts in the Software Development Lifecycle (SDLC) 125
6.3 Participants' opinion of their teams. 126
6.4 Satisfaction with teams' procedures 127
6.5 Likelihood of the existence of vulnerabilities in team's code . . 127

6.6 Types of security issues experienced by participants' companies. 128
6.7 Long term effect of experiencing security issues on awareness and concern for security 128
6.8 Strategies for handling software security 130
6.9 Strategies for handling software security after factor analysis . 132
6.10 Software security motivators 133
6.11 Motivations for software security after factor analysis 135
6.12 Deterrents to software security. 136
6.13 Software security deterrents after factor analysis. 140

D.1 Types of software developed by survey participants 194

List of Acronyms

API Application Programming Interface.

BSIMM Building Security In Maturity Model.

CSCR Collaborative Security Code Review.

CTF Capture The Flag.

CVE Common Vulnerabilities and Exposures.

HCI Human-Computer Interaction.

IDE Integrated Development Environment.

IoT Internet of Things.

IT Information Technology.

KMO Kaiser Meyer-Olkin.

LE Large Enterprise.

NASA National Aeronautics and Space Administration.

NIST National Institute of Standards and Technology.

NVD National Vulnerability Database.

OOP Object-oriented Programming.

OWASP Open Web Application Security Project.

QA Quality Analysis.

SAMM Software Assurance Maturity Model.

SAT Static-code Analysis Tool.

SDL Security Development Lifecycle.

SDLC Software Development Lifecycle.

SDT Self-Determination Theory.

SME Small and Medium Enterprise.

TDD Test-Driven Development.

UI User Interface.

US-CERT Department of Homeland Security's United States Computer Emergency Readiness Team.

W-SDI Work Self-Determination Index.

WEIMS Work Extrinsic and Intrinsic Motivation Scale.

Chapter 1

Introduction

Usable security is an interdisciplinary field that combines various fields of research including computer security, Human-Computer Interaction (HCI), cognitive science, and psychology. The goal of usable security can be described as: designing security systems that users can use comfortably and without making dangerous errors—designing systems that are usable without compromising on security [44, 174]. To achieve this, usable security focuses on the human factors of computer security, including human behaviour and cognition.

Typical end-users have been the primary focus of usable security research [40,130]. These users have little computer security knowledge and may be reluctant to perform security tasks. However, recent work has begun to focus on software developers as users who also need support when dealing with the implementation of software that adequately addresses privacy and security [13, 71, 132]. Developers, although considered experts in their own domain, are typically not security experts [71]. They sometimes make mistakes that affect the privacy and security of their whole user-base [13, 71].

This thesis advances the body of usable security research for software developers, through proposing a new approach for source code security analysis that encourages collaboration and exploration; identifying discrepancies between real-life security practices and best practices, and identifying reasons for such discrepancies; and exploring factors that could influence developers' security processes, such as knowledge and motivation.

1.1 Scope

Detecting software vulnerabilities is a classic problem in computer security. A software vulnerability can be defined as "*a flaw or weakness in system security procedures,*

design, implementation, or internal controls that could be exercised (accidentally triggered or intentionally exploited) and result in a security breach or a violation of the system's security policy" [157]. Thus, exploiting vulnerabilities negatively affects one or more of the pivotal components of security: confidentiality, integrity, and availability [87]. We note that vulnerabilities could be unintentional or could be introduced to a system out of malice. For the work presented herein, we focus on supporting developers avoid unintentional vulnerabilities; malicious developers are thus out of the scope of this work.

Software security focuses on the resistance of applications to vulnerability exploitation. This is different from security functions, which can be expressed as functional requirements, such as authentication [182]. In this thesis, we focus on ensuring *software security* with special focus on the human in the development loop. Security functions are out of the scope of this thesis. Thus, terms such as "security" and "secure" used herein refer to software security (how secure a software application is against unintentional triggers or malicious exploitations of vulnerabilities).

In addition, this thesis focuses on the software security process and understanding how the human actors (*e.g.*, developers) deal with, and influence, this process. Although we do not focus on technologies to support secure software development, this thesis can help inform the design of these technologies.

1.2 Motivation

Historically, security has been an afterthought in software development, where the focus was mainly on functionality [166]. However, increasing threats led to acknowledging the importance of addressing security in the development lifecycle [66, 166]. Major software companies are taking the initiative to integrate security in the SDLC, starting from the early stages. For example, Google has an independent *Security Team* responsible for aiding security reviews during the design and implementation phases, as well as providing ongoing consultation on relevant security risks and their remedies [6]. Microsoft has been following a security-oriented software development process since 2004. The Microsoft Security Development Lifecycle (SDL) introduces

security early in the development and throughout the different stages of the traditional SDLC [104]. In addition, several initiatives for implementing a *secure* SDLC have been proposed [62] (discussed more in Section 2.2). Integrating security in the SDLC from the early stages when vulnerabilities are less expensive to fix [39] has shown improved security outcomes compared to when security was viewed as an additional task [104].

Despite these efforts, software vulnerabilities persist [116]. With increasing connectivity and progress towards the Internet of Things (IoT), threats have changed [77] and software security is often critical. In addition to vulnerabilities in traditional computing systems (*e.g.*, Heartbleed [41] and Shellshock [160]), vulnerabilities are found in devices and applications that are not necessarily considered security-sensitive, such as baby monitors [136], children's toys [155], and medical devices [74, 107, 135]. Also, the threat is no longer limited to large enterprises, even Small and Medium Enterprises (SMEs) are increasingly becoming targets of cyberattacks [153].

The majority of existing usable security research for software security mainly focuses on a single aspect of the process: proposing and improving security tools [13]. Little research [30] addresses the human factors of usable security, such as developers' attitudes, behaviours, and motivations towards software security. This thesis highlights the role usable security research could play towards supporting security throughout the SDLC by focusing on the human factors. Existing usable security literature is discussed in Section 2.4, and the research gap is discussed in Section 2.6. In this thesis, rather than focusing on a specific SDLC stage, we look at the development lifecycle as a whole. We explore current approaches taken towards software security throughout the development lifecycle and how they compare to security best practices. In addition, we explore developers' approaches to performing their tasks, factors that motivate addressing software security, factors impeding the implementation of best practices, and barriers to security efforts.

1.3 Research Question

The research began with a focus on improving the usability of source code analyzers to support developers analyze their code security. However, we recognized that the problem of software security extends beyond security tools. Thus, we then focused our efforts on the overall security process. In particular, this thesis explores the process of handling software security, with a focus on the human aspect, rather than the technological aspect.

We aim to improve the state of software security by understanding and supporting the human in the development loop. The main research question is:

> *How can we support the human dimension of software security throughout the SDLC by better understanding factors that motivate, or impede, security efforts?*

To address this question, we take an empirical approach that includes both qualitative and quantitative methods. This thesis has the following four main objectives.

Objective 1 Supporting collaboration and exploration during source code security analysis.

Objective 2 Exploring how software security fits in the development lifecycle in real-life.

Objective 3 Identifying the means through which developers acquire security knowledge.

Objective 4 Exploring motivations and deterrents to software security.

1.4 Contributions

This thesis addresses a research direction that is currently understudied by the usable security community. We contribute to the understanding of security practices, how they compare to best practices, and factors that influence the process. Our approach

is user-centered with the aim to understand, encourage, and support the integration of security throughout the different stages of the SDLC. The main contributions of this thesis are as follows.

- We studied usability issues facing developers while using code analyzers by evaluating a popular open-source Static-code Analysis Tool (SAT). We then designed a prototype, which offers developers and testers a visual environment to support their analysis of source code vulnerabilities. The usability evaluation of our prototype showed promising results in terms of supporting collaboration amongst developers and encouraging discussion and exploration of potential issues. We also provided general recommendations to guide future designs of code review tools to enhance their usability.

- We conducted an interview study and an online survey study, with developers currently employed in industry, to study different aspects of software security, including the degree of integration of security in the SDLC in real life. We found that developers have varying approaches to software security. In addition, the interviews revealed considerable deviations from best practices in real life. The survey data implied that developers are not explicitly ignoring security, but rather consider it part of their responsibilities and in some cases, they even find their teams' security processes unsatisfactory. Through our analyses, we identified factors that influence the adoption of security best practices, *e.g.*, company culture, security knowledge, external pressure, and experiencing security issues.

- We further analyzed our interview data to explore how developers acquire security knowledge. Through our analysis, we developed a taxonomy of the different opportunities for acquiring security knowledge identified in our data. The learning opportunities varied in their formality and the developer's motivation in initiating them. Learning as a by-product of developers' tasks was more common than other SDLC-independent learning opportunities. We also discuss how some of the learning opportunities we have identified could help harmonize project teams and bridge the gap between developers and security experts.

- We explored factors that could influence developers motivation towards software

security. This was done through analyzing our interview and online survey data. Several factors led developers to become amotivated towards software security, such as being unequipped to handle security tasks. In addition, we identified different motivations to software security that vary in their degree of internalization (whether they are self- or externally-driven). We discuss the importance of focusing on internal motivations to promote software security.

- We propose a human-oriented model to describe how external motivations to software security can be transformed into internal motivations. The model represents successful strategies for motivating software security, while overcoming factors that could lead to amotivation.

1.5 Thesis Outline

This thesis uses the first person plural prose as the work presented herein was supervised by Prof. Sonia Chiasson. This thesis is organized as follows. Chapter 2 presents current security initiatives for supporting the integration of security in the SDLC. It also provides background and discusses related work addressing human factors in software engineering and software security, as well relevant software visualizations research. The gap in research is also discussed in this chapter. Chapter 3 presents our usability evaluation of a popular open-source SAT. We design and prototype our visual analysis environment to support collaboration between developers and testers when analyzing the security of their code. This chapter also presents general guidelines that could help toolsmiths design more usable code analysis tools. Chapter 4 presents our qualitative study investigating software security in practice. We explore security practices employed by development teams and identify pitfalls in the process. In Chapter 5, we further analyze the interviews to explore how developers acquire security knowledge and what motivates them to address software security. We also introduce our model for internalizing software security. Chapter 6 introduces our online survey study, where we further explore our findings from the previous chapters with a bigger participant sample. In Chapter 7, we present our final discussion, conclusion, and future work.

1.6 Related Publications

Work presented in this thesis has been published in peer-reviewed academic venues.

- **Hala Assal** and Sonia Chiasson. Security in the Software Development Lifecycle. In *The 14th Symposium on Usable Privacy and Security (SOUPS)*, pages 281–296, 2018. USENIX. [full paper]
- **Hala Assal** and Sonia Chiasson. Motivations and Amotivations for Software Security. In *SOUPS Workshop on Security Information Workers (WSIW)*, 2018. USENIX. [workshop paper]
- **Hala Assal**, Sonia Chiasson, and Robert Biddle. Cesar: Visual Representation of Source Code Vulnerabilities. In *IEEE Symposium on Visualization for Cyber Security (VizSec)*, pages 1–8, Oct 2016. [full paper]
- **Hala Assal**. Collaborative Security Code Review. In *Proceedings of the 14th International Conference on Mobile and Ubiquitous Multimedia (MUM)*, pages 439–444, 2015. ACM. [extended abstract]
- **Hala Assal**, Jeff Wilson, Sonia Chiasson, and Robert Biddle. Collaborative Security Code Review: Towards Aiding Developers Ensure Software-Security. In *The 11th Symposium on Usable Privacy and Security (SOUPS)*, 2015. USENIX. [extended abstract]

Chapter 2

Background and Related Work

This chapter presents background and relevant research on software engineering, software security, and human factors. We also discuss the gap in research in Section 2.6. Additionally, Section 2.7 provides brief background on theories used in this thesis to explain our results.

2.1 Software Engineering

Software engineering is "*the process of solving customers' problems by the systematic development and evolution of large, high-quality software systems within cost, time and other contraints*" [94]. The Software Development Lifecycle (SDLC) can be defined as: "*a conceptual framework or process that considers the structure of the stages involved in the development of an application from its initial feasibility study through to its deployment in the field and maintenance*" [140]. Different processes, or SDLC models, have been created, such as *waterfall*, *rapid prototyping*, and *agile software development* [140,152], each with its own approach of structuring the SDLC stages. Regardless of the software process, the SDLC must include fundamental design, implementation, and testing activities, including "software validation" [152].

As it is the focus of Chapter 3, we will now briefly discuss some software validation methods that help reveal code defects.

Software testing can be categorized as *Black-box* (*e.g.*, fuzz testing) or *White-box* (*e.g.*, static analysis) testing [152]. Testers using the black-box method investigate the functionality of the software without knowledge of its internal structure or having access to its source code. White-box testing aims to uncover hidden errors by examining every visible path of the source code.

Inspection is a white-box method of software validation. The formal code inspection method proposed by Fagan [58] involved manually inspecting source code line

by line. Despite being a cumbersome and expensive process, evidence has shown the merit of code inspection on the quality of software [28]. Nowadays, a more lightweight version of this method is adopted by organizations such as Microsoft and Google to make use of its benefits while avoiding the shortcomings. This informal tool-based inspection method is referred to as *Modern Code Review* [28] (henceforth, code review).

Static code analysis is sometimes performed during code review [127]. Static-code Analysis Tools (SATs) automate the process of finding defects using static-code analysis. However, existing SATs are not *sound*; they build a non-exact model of the code to be analyzed and this approximation inevitably leads to false negatives. A false negative is when a vulnerability exists in the program, yet the tool fails to detect it. On the other hand, false positives, another problem with SATs, is when the tool mistakenly reports a vulnerability. For example, these could occur when the program interacts with an external system through which the tool cannot trace the flow of data [17]. Thus, SATs require human inspection of potential defects; SAT users need to inspect analysis reports produced by the tool in order to determine whether the discovered defects are true problems or false positives.

Security has historically been outside the focus of the development lifecycle [166]. Recently, with increasing and constantly changing threats, the community recognized the importance of integrating security in the SDLC [66,166]. In the following section, we discuss initiatives proposing the integration of security throughout the SDLC.

2.2 Security Initiatives

Processes and recommendations for integrating security in the SDLC have been proposed by major software companies and other organizations. We now give a brief background on the most notable ones.

Security Development Lifecycle (SDL).

The first initiative encouraging the integration of security in the SDLC from the early stages was introduced by Microsoft. In 2004, Microsoft's SDL [104] was released [66]. It has 7 stages (Figure 2.1). The first stage focuses on providing security training to entities involved in the development process (*e.g.*, developers and testers),

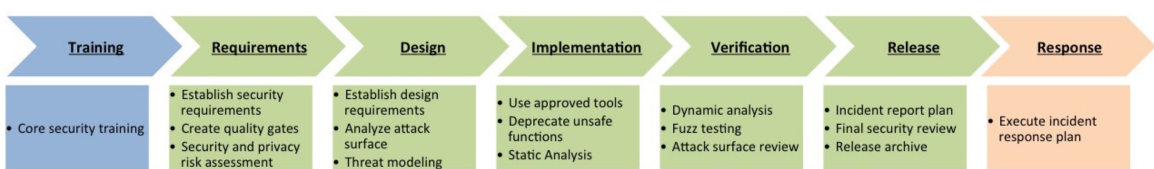


Figure 2.1: Microsoft Security Development Lifecycle [104].

whereas the final stage focuses on response in case of a security incident. The remaining five stages cover the main stages of the development lifecycle; they consists of 16 security practices and can be employed regardless of the platform. The SDL recommends specific references, tools, and sometimes provides training material.

Building Security In Maturity Model (BSIMM). First introduced in 2009 and currently maintained by Cigital [1], the Building Security In Maturity Model (BSIMM) [2] is in its 8th iteration. The model was built by analyzing security initiatives employed by 95 software companies. BSIMM recommends 12 main security practices. However, it does not provide explicit instructions to follow; rather, it provides high-level insights to help companies plan their own secure development initiative and assess their security practices in comparison to other organizations.

These two initiatives are intended for use by development companies of any size. They provide security best practices throughout the different stages of the SDLC. Microsoft divides its security practices into 7 phases similar to the stages of the traditional SDLC (*e.g.*, design and implementation), whereas BSIMM divides them into 4 main categories (*e.g.*, security touch points and deployment).

Open Web Application Security Project (OWASP) initiatives. Focusing mainly on web development, OWASP published several initiatives addressing security [121, 123–125]. The current Software Assurance Maturity Model (SAMM) [124] supersedes an earlier initiative [121]. It recognizes 4 main classes of activities in the SDLC and provides 3 security best practices for each. This model can assist companies in integrating security in their development process, and in evaluating their security initiatives. Additionally, the Developer Guide [123] provides architects and developers with high-level information on security best practices, as well as specific advice. The Testing Guide [125] focuses on best practices for testing and evaluating

[1] https://www.cigital.com

SDLC security activities.

Others. Additional resources for security best practices are also available. Notable examples include: National Aeronautics and Space Administration (NASA)'s *Software Assurance Guidebook* [113], National Institute of Standards and Technology (NIST)'s *special publication 800-64* [86], Department of Homeland Security's United States Computer Emergency Readiness Team (US-CERT)'s *Top 10 Secure Coding Practices* [145], as well as various articles emphasizing the importance of secure development [38, 100, 101, 176].

Resources for software security best practices vary in their organization, as well as their presentation style, *e.g.*, some might focus on technical details more than others. Previous research [108, 133, 166] emphasized the importance of devising methods to help developers and development companies choose the set of best practices they would follow and help them establish the security process within their organizations. The success or failure of these processes, or any human-directed process for the matter, depends at least partially on the human user. We discuss research looking at the human aspect in software engineering and software security in Sections 2.3 and 2.4, respectively.

2.3 Human Factors in Software Engineering

In this section, we present an overview of recent research focusing on specific aspects of human factors in software engineering, namely developers' goals, behaviours, and personalities. Security is usually out of scope in this research, so the findings provide context but may not necessarily apply directly to security. We present this work for insight and completeness.

2.3.1 Factors Influencing Developers' Practices

The software development lifecycle involves multiple stages that require different cognitive processes [82]. Understanding how developers approach their different tasks could provide insight on challenges they face, which could be useful in designing and evaluating development tools [91, 103]. For example, developers' frequently switch between different tasks, thus tools and methodologies should allow reducing untimely

interruptions and support developers in focusing on the task at hand [103].

Previous research found multiple factors that affect developers' practices and the strategies they employ throughout their work day. For example, individual characteristics such as "work style" could influence the developer's strategy for implementing some functionality [91]. Some developers try to get the code to work as soon as possible, regardless of whether they understand its implications; others implement only after acquiring a full understanding of the problem and the solution.

Other factors that influence developers' approaches to their activities include: the type of development process, the context of use of the application they are implementing, and the developers' knowledge of the application and the programming language [91]. Developers rely on different knowledge resources, to gain specific knowledge they need to perform their tasks; *e.g.*, they rely on their colleagues to answer their questions [91] or search for information online [103]. In fact, a quarter of developers' work day is spent on coding-related activities (*e.g.*, browsing for code-related information), and another quarter is spend on collaborative activities (*e.g.*, collaborating with colleagues, answering emails) [103].

2.3.2 Motivations for Conducting Code Reviews

Evaluating the effectiveness of code reviews could be done using various methods, *e.g.*, based on the number of errors detected during the code review, or the losses prevented relative to the code review costs [42]. In a different direction, addressing the human aspect, Bacchelli and Bird [28] focused on evaluating the effectiveness of code reviews in fulfilling the developers' goals. They explored developers' motivations for conducting (non-security focused) code reviews, as well as the actual outcomes of code reviews. By comparing the motivations to the outcomes, the efficiency of the code review could be deduced. Their study included observing Microsoft developers in code review sessions, carrying out semi-structured interviews with these developers, and manually inspecting discussion comments contained within code reviews.

The two most prominent motivations for developers to perform code reviews were to find defects and to improve the code (in terms of readability, adding comments, and removing dead code). However, developers also discussed additional motivations.

They explained that code reviews provided a good opportunity for learning (*e.g.*, developers followed code reviews in order to learn about the different areas of the code they need to modify to implement certain features), allowed for knowledge transfer between developers and reviewers, and helped increase team awareness (*e.g.*, developers use code reviews to keep team members aware of changes done to the code) and enforced the concept of shared code ownership between members of the developing team.

However, after analyzing and categorizing code review comments, the authors [28] found that developers' motivations do not match code review outcomes. Developers did not spend as much time finding defects as they had expected; instead more time was spent on "code improvement" (the most prominent category of code review comments). "Finding code defects", the top motivation for performing code reviews, was only the fourth most frequent category of comments. Comments relating to "knowledge transfer" were also found, where reviewers[2] pointed developers to external resources to gain the knowledge necessary to handle detected issues.

To increase the efficiency of code reviews in fulfilling developers' objectives, Bacchelli and Bird [28] recommended integrating code analysis tools in code reviews to automate defect-finding tasks. Static analysis tools can support developers during code reviews by reducing the time and effort spent on reviews [129] and allowing developers to focus on less obvious defects [28]. In addition, since their study provided evidence that code reviews involve much more than merely finding defects, the authors emphasized the importance of providing proper support for all the different aspects of code reviews (*e.g.*, knowledge transfer), rather than exclusively focusing on finding code defects.

2.3.3 Reasons for Use and Under-Use of SATs

Despite their benefits [22, 35, 39], SATs are not widely accepted by the software development community [22, 79]. Previous research [79, 95] worked towards identifying

[2] The developer who carries out the code review in this study was not necessarily the author of the code.

tools' advantages and disadvantages that influence developers' decision to use or dismiss them. Existing and potential features that were described by developers as the main reasons that would motivate them to use SATs were [79]:

- *Automatic defect-detection.* Some developers find SATs a convenient method to identify defects in their code. In comparison to manual line-by-line code inspection, SATs offer an automated process, making fault detection faster and less effortful.

- *Providing support to team development efforts.* Some development teams use SATs to raise awareness of potential problems in the early stages of the development, as well as to enforce coding standards and maintain coding styles across teams.

- *Availability in Integrated Development Environment (IDE).* Developers expressed their willingness to use SATs if they were part of their existing tools. Developers are more likely to use these tools if they were built-in, or could be integrated in their existing IDE.

- *Customizable output.* Developers consider the ability to customize SATs, to produce an output that is most relevant to them, as one of the main factors influencing their willingness to use these tools. Developers explained that by customizing the tool to focus on relevant issues, the quality and volume of its defect predictions can be significantly improved.

The main factors leading to the underuse of SATs are [79, 95]:

- *The quality of tool output.* SATs' false positives are one of the main reasons behind their underuse. Some developers see these tools as useless, especially when they find that the number of false positives exceeds the number of true positives. In addition, with large projects, SATs produce a large volume of warnings that could reach thousands. Thus, with many false positives and up to thousands of potential defects, developers find it inefficient to use these tools. In order to accept SATs, developers need to trust that the tools are providing them with accurate information in an efficient manner [92].

- *Support for collaboration.* Developers are reluctant to use SATs because they do not adequately support collaboration between team members. Although these tools could be used to enforce coding standards and styles, development teams find that setting up and sharing tools' settings is an unintuitive process that usually results in confusion when the standards need to be changed. Even though some tools (*e.g.*, FindBugs [5]) allow developers to exchange defect reports, this feature takes the developer out of the development environment, and thus out of context. This causes developers to be reluctant to use this feature, and subsequently the static-analysis tool altogether.

- *Tool customization.* Developers believe the quality of tool output could be improved (*e.g.*, to show less false positives) if they were able to customize the tools to look for defects they care about the most. However, many existing tools require complicated steps to customize, and yet they do not allow developers make the customizations they want. Additionally, many tools do not allow developers to temporarily dismiss certain warnings, or categories of warnings; these tools only allow developers to completely turn off certain filters on a project level. This setting could later cause problems if the developer forgets to turn the filter back on.

- *Result understandability.* Developers feel that SATs do not provide enough information to assist them in assessing whether the warning is a true or false positive. The tools fail to explain their reasons for triggering a warning; they usually do not clearly explain what the problem is, why it is considered a problem, and how it could be fixed. Without obvious reasoning why a warning was issued, developers could not develop trust for the analysis tool, thus lowering the likelihood of using it for defect-detection.

- *Actionable tool output.* Code suggestions and quick fixes are two features many developers miss in most existing SATs. Developers expect to be able to take clear steps, guided by the SAT, to fix the defects it has detected. However, there is some skepticism to the true merit of quick fixes. For example, developers might wrongfully accept a quick fix to a false positive, without thoroughly

inspecting the code and the bug report, potentially leading to more problems. If the tool does not provide actionable output, developers see more benefit if the SAT focuses on defects discovered in newer versions of the project, especially in large projects.

2.4 Human Factors in Software Security

Garfinkel and Lipford [65] highlighted the lack of human factors research focusing on software developers. Green and Smith [71] discussed how developers are often viewed as "the weakest link"—mirroring the early attitude towards end-users before usable security research gained prominence. While developers are more technically-experienced than typical end-users, they should not be mistaken for security experts [13, 71]. They need support when dealing with security tasks, *e.g.*, through developer-friendly security tools [178], programming languages that prevent security errors [71], or guidelines to help developers avoid introducing vulnerabilities, such as Microsoft's NEAT and SPRUCE [65, 105].

Acar *et al.* [13] outlined a research agenda for usable security research for software security—a research area that is in its early stages. Their agenda includes understanding developers' attitudes and security knowledge, exploring the usability of available security development tools, and proposing tools and methodologies to support developers in building secure applications. Pieczul *et al.* [132] discussed challenges facing developer-centred research and highlighted the need for deeper understanding of the continuously evolving field of software development. Recruiting developers and ensuring ecological validity are examples of challenges facing studies in this area. Developers are busy and must often comply with organizational restrictions on what can be shared publicly. To partially address these issues, Stransky *et al.* [158] designed a platform to facilitate distributed online programming studies with developers.

We now discuss relevant research focusing on the human factors of software security and investigating security tool adoption, their benefits, and improving and proposing new tools.

2.4.1 Security Tool Adoption

While investigating the integration of security tools in companies' processes, previous research found that developers generally exhibited a *security is not my responsibility* attitude [175,179,182]. Moreover, only a small number of large companies in the study by Xiao *et al.* [179] were using security tools to ensure the security of their code, and in these companies security best practices were informal. Developers were expected to follow best practices without any specific policies or guidelines. Small companies in the same study did not consider security as a dimension in the development process; in these companies developers were not required to follow secure coding practices. Across relevant research, several factors influencing the adoption of new security tools were identified; these could also be potential influencing factors to the integration of security in general, not just the adoption of security tools. We will now highlight some prominent factors influencing developers' adoption of security tools.

Company Policies and Company Culture. The development company's policies and the overall company culture towards security were found to be among the main deciding factors in motivating security in development and encouraging developers' decision to adopt new security tools [179,182]. Developers who were required by their company policy to use security tools to test their code, did indeed use these tools. Companies that were keen on security considered it as a shared-responsibility; these companies mandate the use of security tools and following security best practices, encourage collaboration between developers and security experts, and offer security education opportunities to developers [179]. On the other hand, developers who faced bureaucracy when trying to introduce a new tool to their toolbox, dismissed the idea of using new security tools. To encourage developers to use security tools, Wurster and van Oorschot [178] suggest mandating their use and rewarding developers who code securely through external means.

Knowing that even developers who have security knowledge and those recognizing the importance of tending to security throughout all the stages of the development lifecycle exhibit the same behaviour [175, 179, 182], we believe that the *security is not my responsibility* attitude directly stems from company policies and culture. Developers with this attitude relied on other entities to ensure the security of their

applications, be it the design team (by identifying security requirements), or the code review or testing teams (by identifying vulnerabilities in the code). However, despite this reliance, collaboration between developers and testers was lacking [179]. This dissociation implies that developers will not learn about security and will continue to make the same vulnerability-causing programming mistakes.

Application Domain. Participants' perception of the usefulness of security to their applications influences their intentions to practice secure development [177]. The domain and context of use of the application was found to be prominent factor in adopting security tools [179, 182]. Developers assume that if end-users do not interact with their code or if they are building an internal application for a known set of users, then their application is not security-sensitive and thus there is no need to use security tools. Some developers assume that their application is secure, since the design stage considers any security features that need to be included. Even though this is a plausible assumption, it only applies to security functions; the design stage does not account for vulnerabilities resulting from implementation mistakes. This highlights some developers' misconceptions of software security; they only attribute it to specific functions, such as "operational security, software access control, and user authentication" [179], missing vulnerabilities—a major reason for security breaches.

Security Tools Complexity. The factors described above were extracted from an exploration focusing on adopting security tools, however, they could be considered general factors affecting the use of security tools (whether new or already part of the developers toolbox). The complexity of security tools, however, is more relevant to adopting *new* tools.

Some developers are reluctant to use security tools because they are complex and require special security knowledge [175, 179]. For many, installing and learning how to use and interpret the output of a new tool was too steep a cost that sometimes outweighs the benefits [175]. Some developers also mentioned that the high rate of false positives increased the "cost" of using the tool without any apparent benefits. Unless mandated by their company, developers are less likely to take time out of their allocated development time to try using a new security tool.

2.4.2 Developers' Abilities and Expertise

As security vulnerabilities increasingly spread [114, 119], developers and their lack of security education have been criticized. The assumption was that if developers learned and cared about security, they could avoid vulnerabilities [24,178]. Conflicting opinions argue the reason might be because security guidelines do not exist or are not mandated by the companies [175, 179, 182], or that developers might lack the ability [119] or proper expertise [27] to identify vulnerabilities despite having general security knowledge.

Baca *et al.* [27] found that developers' general experience (*e.g.*, the number of years of development experience, or experience with the programming language) did not have the expected positive impact on the correctness of classifying vulnerabilities as true or false positives. The two main influential factors were developers' experience with code analysis tools and prior security knowledge. Developers with specific experience in using SATs and prior security knowledge were significantly better at properly analyzing security vulnerabilities.

Oliveira *et al.* [119] argued that developers and security education are not the root causes of security vulnerabilities. They explained that developers' decision-making processes are "heuristics-based"; throughout their tasks, developers are consumed with solving problems that assume common cases, and are not necessarily considering all the available information. Vulnerabilities are usually unexpected corner cases [119] and identifying them is a cognitively-demanding task [149], hence they are not included in developers' set of heuristics. Oliveira *et al.* showed that security vulnerabilities are "blind spots" in developers' decision-making processes; developers are not commonly thinking in terms of security while coding. The study showed that developers are mainly focused on functionality and performance, and that security is not commonly included in their programming tasks. Even participants with security knowledge were unable to identify security issues, because they were not currently in the security mindset. However, when explicitly alerted of potential security issues, participants were more conscious of security in their tasks.

Baca *et al.* and Oliveira *et al.* argued for a more customized security experience for developers. Baca *et al.* suggested that to achieve best results in analyzing security

vulnerabilities, developers need to gain practical experience in using static-analysis tools with a focus on security aspects. Whereas, Oliveira *et al.* recommended in-context security education. They explained that solely relying on teaching developers about security vulnerabilities in general and then expecting them to identify these vulnerabilities while coding is not ideal. Rather, they recommended priming developers about potential vulnerabilities in-context, in their IDE, while they are writing code. In addition, Thomas *et al.* [165] call for better training opportunities that target the specific security issues that developers encounter in their code, as well as tailoring the training to address weaknesses in developers' security knowledge. Weir *et al.* [172] found that direct interaction with security issues can in fact lead to better security.

In general, developers often have trouble finding and understanding security information [29, 110], and usually turn to their peers or web searches for help [29].

Existing research has noted a gap in communication and security knowledge between developers and security experts (sometimes referred to as auditors) [110, 111, 165]. Some teams employ a developer who is interested in, or knowledgeable about, security to act as a liaison between the development team and security experts [165]. Nafees *et al.* [110] proposed "Vulnerability Anti-Patterns" which encapsulates recurring errors that lead to vulnerabilities in a template that is intended to captures knowledge of existing vulnerabilities in a usable manner for developers. Nafees *et al.* hypothesize that using this pattern-based approach familiar to developers would help bridge the communication gap between security experts and developers, and help developers understand how attackers can exploit vulnerabilities in their code.

2.4.3 Improving and Introducing New Security Tools and Methodologies

Approaches to improve security include advocating for the use of static analysis tools [35, 78, 81], reducing the number of false positives in security tools [118, 168], and using innovative approaches, such as machine learning to assist in the discovery of vulnerabilities [73, 168, 183]. For example, Perl *et al.* [131] recently used machine learning techniques to develop a code analysis tool. Their tool has significantly fewer false-positives compared to previous work. In the reminder of this section, we discuss relevant work that focuses on the human aspect.

Smith *et al.* [149] proposed an approach to help toolsmiths build tools that support developers' information needs while analyzing vulnerabilities. They identified 17 categories of information that developers seek during the analysis of SAT warnings. These categories included questions regarding understanding vulnerabilities, attacks that might exploit these vulnerabilities, alternative fixes, the intended functionality and the context of the code, how to interpret the tool's output, how to find more details on the detected vulnerabilities, and whether the vulnerability was worth spending time to fix it [149, 150]. Their study, however, included students who do not necessarily have the same information needs or apply the same strategies as professional developers.

Xie *et al.* [180] proposed a tool that follows the in-context approach recommended by Oliveira *et al.* [119]. Xie *et al.* prototyped a tool to remind web developers of secure programming practices in their IDE. The tool performs static analysis of the code, and alerts developers of potential issues on-the-spot. Although the tool does not cover all vulnerability types, usability evaluations (*e.g.*, [96, 163, 164, 181]) showed promising results in encouraging developers to be more attentive to security. Focusing on mobile applications, Nguyen *et al.* [115] developed an IDE plugin to support Android application developers adhere to, and learn about, security best practices without distributing their workflow. Studies evaluating the plugin suggest that its usage significantly improves the security of code created by both professional developers and hobbyists [115].

In an alternative approach, Wurster and van Oorschot [178] recommend taking developers out of the loop through the use of Application Programming Interfaces (APIs) to improve code security. Trüpe [169] proposed a research direction for improving the design of APIs to reduce vulnerability-causing mistakes. Acar *et al.* [10] evaluated five cryptographic APIs and found usability issues that sometimes led to insecure code. They also found that API documentation that provided working examples was significantly better at guiding developers to write secure code. Focusing on software security resources in general, Acar *et al.* [14] found that some available security advice is outdated and most resources lack concrete examples. In addition, they identified some under-represented topics, including program analysis tools.

2.5 Software Visualizations

Visualizations is one of the main demanded features of SATs [79]. In this section we give a brief background on software visualizations and visualizations specifically for software security.

Visualizations transform information into a visual form that reveals hidden information in the data, supports and encourages data exploration and analysis [49]. Visualizing data can lead to discoveries that were otherwise inconceivable [151].

Software visualization is a form of information visualization. Specifically, it is "*the art and science of generating visual representations of various aspects of software and its development process*" with the goal to "*help to comprehend software systems and to improve the productivity of the software development process*" [49].

The majority of the software visualization literature focuses on supporting software development tasks [69]. Visualization systems exist for exploring the source code and its structure, such as call graphs and metrics [162], the relations between Object-oriented Programming (OOP) classes [15, 128], and class dependencies [50]. Other work focuses on visualizations to support performance analysis [46, 138, 147]. As examples, Figure 2.2 shows a call graph for the *libgklayout* Mozilla plugin, whereas Figure 2.3 visualizes which class depends on which for a Java graphics framework.

Other research focused on using visualizations to support collaboration during SDLC activities. Müller *et al.* [109] explored the use of source code visualizations, among other methods, to encourage and support collaborative code reviews using multitouch interfaces. Anslow *et al.* [18] developed a source code visualization system using a large multitouch table as the interface. The aim of their system was to help developers collaborate in exploring the structure and evolution of the different versions of the software systems. The visualization system supports multiple visualizations that provide an overview of the system being analyzed with the ability to dig deeper for more details, as well as discover problematic entities such as particularly small or large classes. Anslow *et al.* found their system did indeed encourage collaboration and team discussion.

As for visualizations for software security, we found little work addressing this

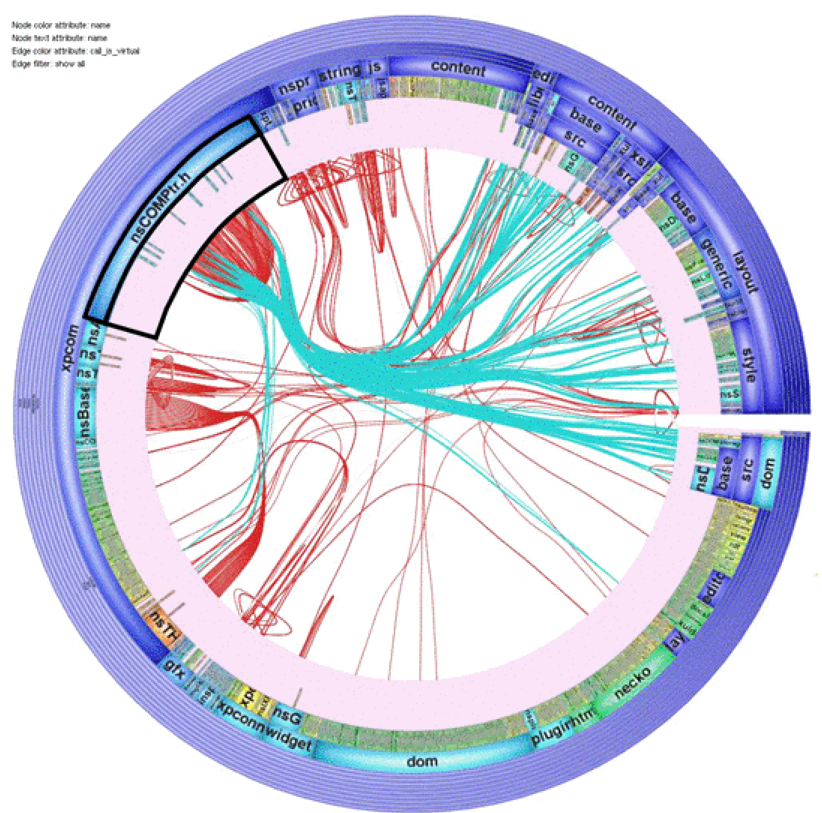


Figure 2.2: Call graph of a Mozilla plugin [162].

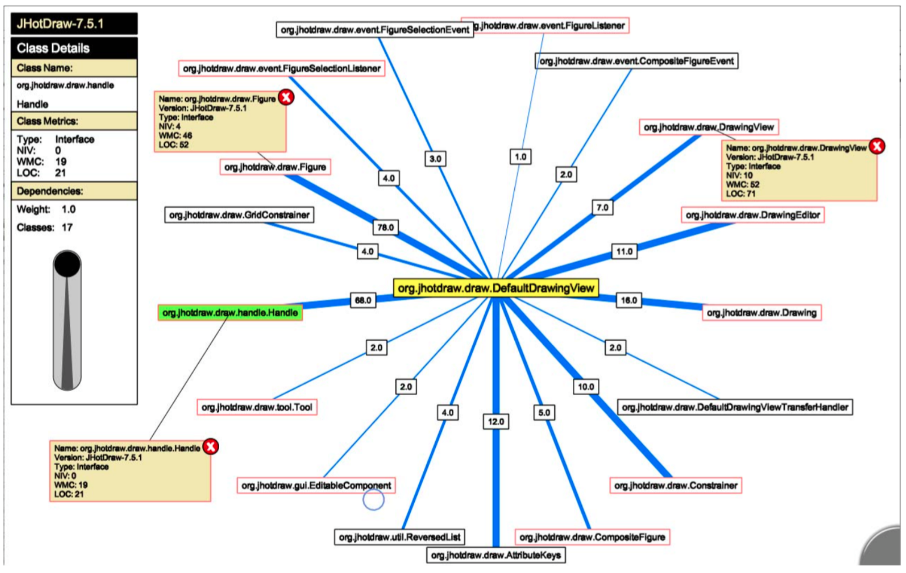


Figure 2.3: SourceViz Class dependency view [18]

area. We will now present two notable examples.

Fang *et al.* [59] proposed a tool that automatically produces diagrams necessary for software security analysis tasks, such as threat modelling. The detailed diagrams are automatically generated by the tool after it collects trace data from the system during run time. The tool allows security analysts to explore the diagrams through time and with different levels of detail. Automatically generating the diagrams using this tool is significantly faster than the manual method; the proposed tool reduced the time taken to the initial diagrams from months to hours. This tool, however, requires a fully implemented program, so it cannot be used in the early stages of the SDLC.

Goodall *et al.* [69] developed a visual analysis system allowing developers to explore vulnerabilities detected in their source code. They aimed to help developers gain a better understanding of the security of their code by providing a visual representation of the aggregate results from different code analysis tools. The visualization presented each source code file as a block, where the block's width depends on the number of potential vulnerabilities detected in the file it represents. Although the authors explain different use cases of their proposal, there was no user testing or usability evaluation done to evaluate its efficacy.

The majority of the work addressing visualizations for software security fails to address the human aspect. The cybersecurity visualization research community has acknowledged the need for using user-centered design methodologies and evaluation through the entirety of the design process of visualization tools [102]. Despite this, most work published at VizSec, [3] the main conference for security visualizations, rarely includes user studies or evaluations of usability with target users of the system. In Chapter 3, we present our user-centered approach for designing and evaluating a visual analysis prototype where developers/testers collaborate to reduce risks of security vulnerabilities in their code.

[3] http://vizsec.org

2.6 Research Gap Analysis

In this chapter, we reviewed relevant research on the human factors in software engineering and software security. As evidenced, several research gaps remain in addressing the human aspects of software security. Usable security research for software developers has been an under-investigated area [65,71] and is still in its early stages [13] despite recent advancements.

In this thesis, we are taking a holistic approach to explore software security in real life. Rather than focusing on specific tools or a specific development stage, we are addressing security in the SDLC overall. We are looking into real-life practices to determine the status quo, and identify positives and negatives in existing processes. This allows for more informed approaches towards supporting the positives and addressing barriers to security. Knowing where we stand in terms of the overall security can also inform approaches focusing on specific stages of development. In general, we focus on the interplay between the developer and software security. Specifically, we explore developers' motivations, abilities, expertise, attitudes, and behaviour towards software security. We take a human-centric approach to address the following aspects of the research gap:

- Focusing on the overall process of software security.
- Identifying and evaluating current strategies and processes addressing security throughout the SDLC in practice.
- Exploring developers' motivations towards software security and identifying factors that influence their motivation.
- Identifying opportunities for acquiring security knowledge, and investigating how it influences security practices and motivation.
- Supporting exploration of security vulnerabilities and collaboration between team members and between teams.

2.7 Background on Activity Theory and Self-Determination Theory

We will now give brief background on two main theories that we have used to explain our results in Chapter 5.

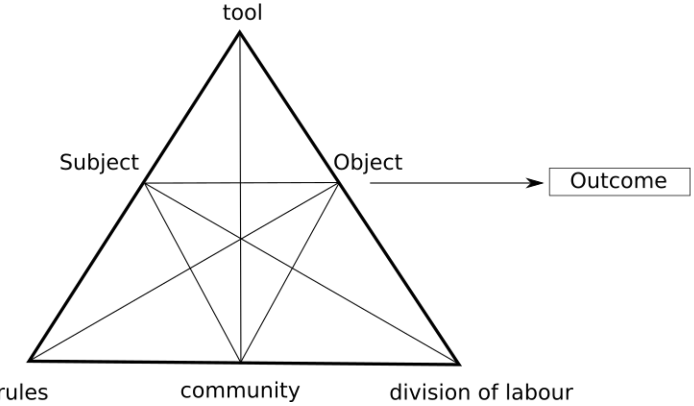


Figure 2.4: Engeström's triangle [55]

2.7.1 Activity Theory

Activity theory [53,55] can be defined as "a philosophical and cross-disciplinary framework for studying different forms of human practices as development processes, with both individual and social levels interlinked at the same time" [90]. Activity is the unit of analysis and is the key to understanding human functioning in activity theory [55, 112]. An activity consists of a *subject* (someone or a group involved in the activity) who uses a *tool* as a mediator to transform the *object* (their goal or objective) into an outcome [112]. Engeström extended this model by proposing the "activity system model (Engeström's triangle)" [55, 90]. As shown in Figure 2.4, *tools* mediate the relation between *subject* and *object*, *rules* (both explicit and implicit) mediate the relation between *subject* and *community*, and finally the mediation between *object* and *community* is done through *division of labour* (explicit and implicit organization of the community involved in the activity) [55, 90].

The "third generation of activity theory" takes two activity systems as the minimal unit of analysis [54]. It aims to understand discussions, perspectives, and interaction between multiple activity systems. As shown in Figure 2.5, each activity system transforms the raw object (object 1) to their desired outcome (object 2) and potentially to a shared outcome (object 3).

One of the most relevant principles of activity theory is the principle of *multi-voicedness* [54]. Rather than a homogeneous system, activity theory views an activity

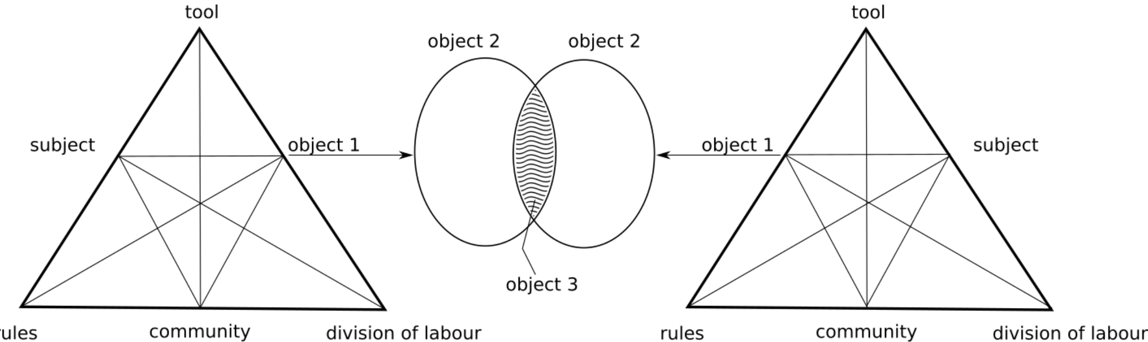


Figure 2.5: Third generation activity theory with two interacting activity systems [54]

system as consisting of multiple perspectives, traditions, and interests [54, 117]. The multi-voicedness is magnified when multiple activity systems interact. Even though multi-voicedness can be a source of trouble, it can also be a source of innovation, when the participants from the interacting activity systems are involved in acts of communication and negotiation to amalgamate their (possibly) conflicting perspectives [54]. These acts of communication and negotiation are what result in reaching the shared outcome (object 3 in Figure 2.5).

Contradictions may arise due to "historically accumulating structural tensions within and between activity systems" [54]. Such contradictions are considered by activity theory as the central source of the activity's change and development that occurs through attempts to resolve these conflicts.

In Chapter 5, we use Activity Theory to describe the interaction between the different teams working on developing a software product, their different objectives and perspectives, and how they can benefit from this multi-voicedness and resolve their contradictions to enhance the security of their software.

2.7.2 Self-Determination Theory (SDT)

In Chapter 5, we use the Self-Determination Theory (SDT) [47, 141] to explain what motivates developers and their teams to address software security, as well as reasons for the lack of motivation with respect to software security. SDT identified distinct types of motivation, each with clear consequences on human potentials to thrive [142], specifically for learning, performance, personal experience, and well-being [141]. SDT

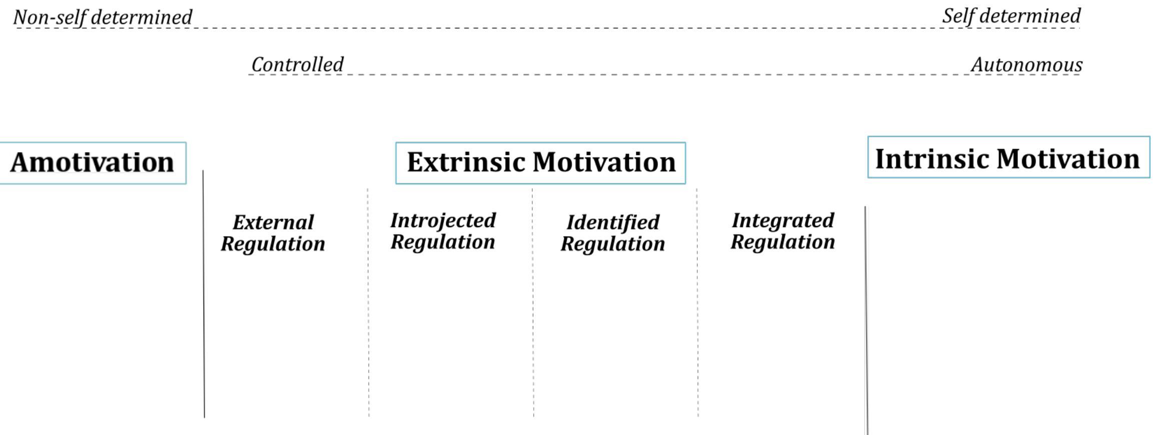


Figure 2.6: The self-determination continuum [64, 141]

uses the *autonomy-control continuum* to differentiate the different types of motivation with respect to their regulation [142]. Behaviours are autonomously motivated when they are fully self-determined, whereas controlled behaviours are those driven by external or internal pressures or an obligation to act [141, 142]. Figure 2.6 shows the different types of motivation: intrinsic, extrinsic, and amotivation.

Intrinsic motivation is when an activity is voluntarily performed for the pleasure and enjoyment it causes. Intrinsic motivations are driven by humans' "inherent tendency to seek out novelty and challenges, to extend and exercise one's capacities, to explore, and to learn." [141].

In contrast, extrinsic motivation is when a person is engaged in an activity for outcomes separate from those innate to the activity itself [141]. SDT classifies four different types of extrinsic motivation: external, introjected, identified, and integrated regulation. Although extrinsic, these motivations vary in their degree of autonomy and self-determination (Figure 2.6). Along the autonomy-control continuum, *external regulation* is the least automomous extrinsic motivation, where behaviours are performed purely to satisfy an external demand, to receive an external reward, or to avoid negative consequences. *Introjected regulation* behaviours are less externally-regulated (less controlled), however, behaviours have not been fully internalized or accepted as one's own. They are rather internally-imposed to maintain self-esteem, *e.g.*, to avoid the feeling of guilt or to boost ego. *Identified regulation* is a more autonomous form

of extrinsic motivation, where the person consciously evaluates the behavioural goal, identifies with it, and accepts it as personally important [142]. Although internally-regulated, the behaviour is seen as chosen by oneself, thus self-determined [170]. The most autonomous form of extrinsic motivation is *integrated regulation*, where the person fully owns the behaviour; she has internalized the behavioural goal and fully identifies with it. Integrated regulation is considered extrinsic, despite the person's autonomy and self-willingness to act, because the behaviour is performed for separate outcomes and not for the pure enjoyment from the activity itself [170].

Based on their levels of control versus autonomy, intrinsic motivation, integrated regulation, and identified regulation form an *autonomous motivation composite* [141]. External and introjected regulation form a *controlled motivation composite* [141].

Amotivation is the lack of motivation to act, where the person does not act at all or acts without intent [141]. Amotivation has three forms. The first form is when people feel they cannot effectively achieve the desired outcome, *e.g.*, because they are not competent to do it [141, 142, 170]. The second form of amotivation occurs when the action lacks interest, relevance, or value to the person [141, 142, 170]. Finally, the third form is when amotivation to a behaviour is actually a defiance and motivation to oppose said behaviour [142].

SDT shows that compared to controlled motivation, the more autonomous motivation styles are associated with various positive outcomes, such as increased engagement, improved performance, more creativity, more cognitive flexibility, and better learning [141, 170].

To facilitate the integration of extrinsically-motivated behaviours, the person should feel more autonomy and freedom which allows them to internalize these behaviours [141]. This leads this person to experience more volitional persistence and self-driven interest in the activity [170]. In addition to autonomy, SDT postulates that the degree of internalization depends on the person's perceived self-competence to perform the act [141], and their feeling of *relatedness* (*i.e.*, feeling connected to others and being a significant member of the group [142]).

In Chapter 5, we use SDT to explain motivations and amotivations to software security identified in our interview data.

Chapter 3

Visual Representation of Source Code Vulnerabilities

Software security initiatives recommend using SATs to evaluate source code in the early stages of development. However, despite their benefits [39], SATs are not widespread in the Software Engineering community [95], for a variety of reasons including the lack of support for collaboration [79].

In this chapter, we explore the human factors aspect of code analysis to understand the reasons for the low-adoption of SATs. We first study the usability issues faced by software developers while using code analyzers by evaluating FindBugs[1] [5], a popular open-source SAT. Then, to address serious usability issues uncovered by evaluating FindBugs, we take a user-centered approach in designing a visual analysis environment to support developers analyze the security of their code. We design and implement an initial prototype, Cesar, to support what we call CSCR, where developers/testers collaborate to reduce risks of security vulnerabilities in the code under review. Next, we evaluate the usability and effectiveness of the prototype, and propose additional features for future design iterations. The usability evaluation studies, for both FindBugs and Cesar, use the Cognitive Dimensions framework and are informed by the Cognitive Walkthrough methodology. As a final contribution of this chapter, we provide general recommendations for designing collaborative code review tools based on the lessons learned from evaluating FindBugs and Cesar.

This chapter is published in IEEE Symposium on Visualization for Cyber Security (VizSec), 2016 [21]

[1] At the time of writing this thesis, SpotBugs (`https://spotbugs.github.io`) had been created as a successor for FindBugs.

3.1 Using the Cognitive Dimensions Framework for Usability Evaluation

The Cognitive Dimensions framework [37] seeks to determine whether users' intended activities are appropriately supported by the system in question. In cases where there are a deficiencies, the designer explores how the system can be fixed and the trade-offs of different design alternatives guided by the framework. The Cognitive Dimensions framework was designed to aid, even the non-Human Computer Interaction specialists, in evaluating the usability of their systems. Blackwell and Green [36] developed a Cognitive Dimensions questionnaire for use by prospective users for evaluating system usability. One of the aims of this framework is to improve the quality of discussion between designers and those evaluating the design by providing a common vocabulary—the cognitive dimensions. Evaluating the usability of a system using the Cognitive Dimensions framework consists of three main steps: (1) classifying users' intended activities, (2) analyzing the Cognitive Dimensions, and (3) determining whether the system appropriately supports the users.

The Cognitive Dimensions framework classifies six generic activities when dealing with information structures: incrementation (adding more information without changing the existing structure), transcription (copying information from one information structure to another), modification (changing the structure of information), exploratory design (exploring with changing the existing structure and adding further information to it), searching (looking for specific information), and exploratory understanding (discovering the structure and the basis of information). Activities in Cesar fall under searching, and exploratory understanding. There are 13 main Cognitive Dimensions, and more have been proposed in literature [37]. Each dimension gives a reasonably general description of an information structure. The purpose of the system being evaluated defines whether it is desirable to be rated high/low on each Cognitive Dimension. For example, consider the *Viscosity* Cognitive Dimension; it describes the system's resistance to change. A viscous system requires users to perform many actions to fulfill a single task. Viscosity could be useful in preventing accidental errors; the redundancy in steps allows users to notice errors and also forces them to think about their actions before acting. However, viscosity is harmful for modification and exploratory activities, as it places a cognitive load on the users.

When using the framework for the purpose of our studies, it was confusing and disruptive to the discussion to have some Cognitive Dimensions that the system is supposed to fulfill (*i.e.*, Cognitive Dimension is high) and some which it should avoid (*i.e.*, Cognitive Dimension is low). As a solution, we present all Cognitive Dimensions as desired dimensions, rewording as needed to frame each positively.

We provide a brief description the five Cognitive Dimensions [37] selected for the usability evaluations presented herein. They are all particularly desirable cognitive dimensions for systems enabling exploration.

- *Fluidity (FLUI)* is the desirable counterpart of *Viscosity*. Being at the opposite end of the viscosity spectrum, fluidity is useful for modification activities and exploratory design. Due to the nature of our application (code review tools), we include this dimension relating to changes to how defect and vulnerability information is represented.
- *Low Cognitive Load (LCOG)* is the desirable counterpart of *Hard Mental Operations*. It described a system that does not place a high cognitive load on the user.
- *Abstraction (ABST)* describes a system that uses abstraction mechanisms and the types of abstractions used. Abstractions help make information structures more succinct and could reduce viscosity. Enabling different levels of abstraction is useful for exploration activities.
- *Closeness of Mapping (CLOS)* is providing a match between representations of information and its domain in a way that allows users to build on their domain knowledge to solve problems.
- *Visibility and Juxtaposability (VIJU)* is the ability to view components easily and to view any two components side-by-side. This Cognitive Dimension is useful for transcription and incremental activities, and is especially useful for exploratory design.

3.2 Using the Cognitive Walkthrough Methodology for Usability Evaluation

The Cognitive Walkthrough [134] is a method used for evaluating the usability of systems from the perspective of users. It focuses on evaluating the system learnability by focusing on users' cognitive activities to ensure the ease of system learning through exploring the interface. Users' tasks are identified, and one or more evaluators work through the steps to perform these tasks from prospective users' perspective. Evaluators thus identify potential usability issues and provide suggestions to improve the system learnability. This method has been used to evaluate the usability of different systems, including visualization systems [16].

At each step during the walkthrough, evaluators typically ask the following four main questions. We present the questions herein essentially verbatim from the original paper [173].

- Will the user try to achieve the right effect?
  - For the current step, does the user realize that this step is required to achieve their goal? For example, if the user's objective is to identify potential security vulnerabilities in their code using a static analysis tool, do they realize that they first need to run the tool to analyze the codebase?
- Will the user notice that the correct action is available?
  - For example, is the button visible? or is the the required action intuitive?
- Will the user associate the correct action with the effect they are trying to achieve?
  - Will the user realize that the visible action actually helps them achieve their goal? For example, if the user needs to click a button to run the SAT, is the button text clear? Does it help the user associate it to their goal?
- If the correct action is performed, will the user see that progress is being made toward solution of their task?

– Will the system provide the user with proper feedback showing their progress towards completing their task?

In this chapter, the usability evaluations of FindBugs and Cesar used the Cognitive Dimensions framework and was informed by the Cognitive Walkthrough methodology. Whereas a typical Cognitive Walkthrough focuses on system learnability from the perspective of novice users, our method focuses on the overall usability of the system, whether the system helps users understand and explore the information it presents, and whether it influences the interaction between users. However, similar to a Cognitive Walkthrough, we observe evaluators' in-context discussion while they perform the different tasks provided by the interface. Next, we reflect on the results of the these discussions, and evaluate the interface using the Cognitive Dimensions framework.

3.3 FindBugs' Study

After surveying different open source and proprietary SATs [118], we chose to evaluate the usability of FindBugs.[2] It is a popular open source tool used in the Microsoft Security Development Lifecycle (SDL) and is widely used in similar research projects [23, 79, 185]. This study does not focus on FindBugs' underlying defect detection mechanisms, but rather on the tool's User Interface (UI) as it is the element with which developers interact. FindBugs analyzes Java code to detect potential defects, and divides them into nine categories, *e.g.*, Security, Malicious Code, and Performance. Each defect has two metrics: "Rank" indicating its severity and "Confidence" indicating the tool's confidence that it is an actual issue. Figure 3.1 shows a screenshot of FindBugs interface and the output of its analysis of an open-source codebase.

3.3.1 Study Design

Informed by the Cognitive Walkthrough methodology, we evaluated the usability of FindBugs v.2.0 with a group of four evaluators who are experts in the fields of security

[2] `http://findbugs.sourceforge.net/findbugs2.html`

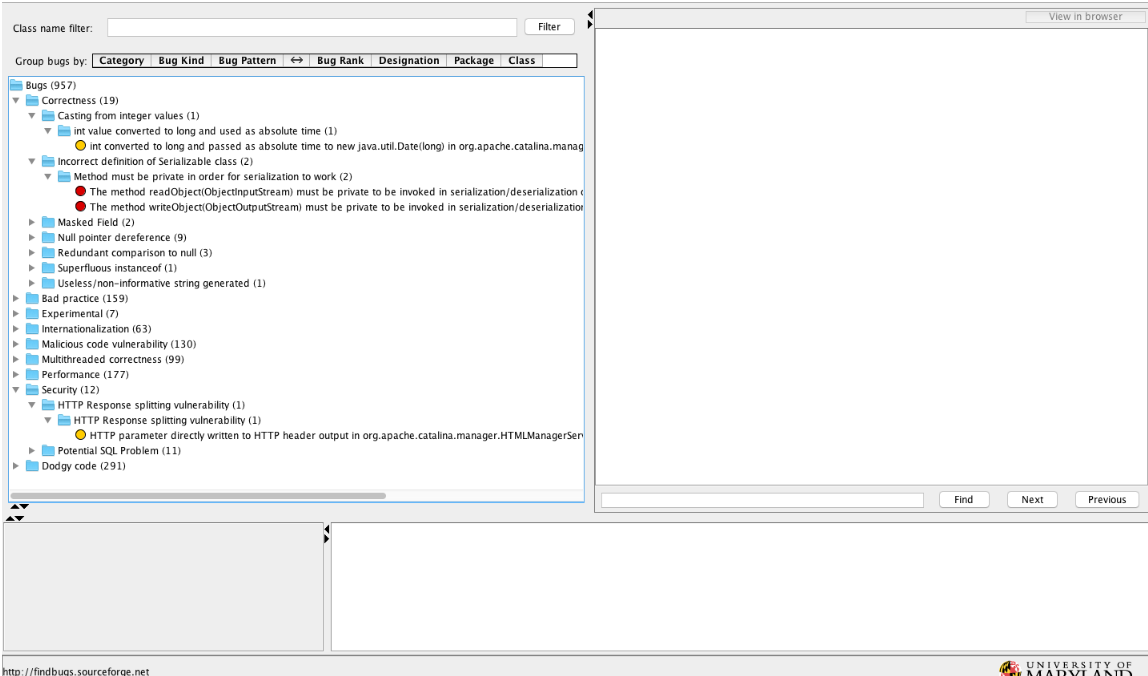


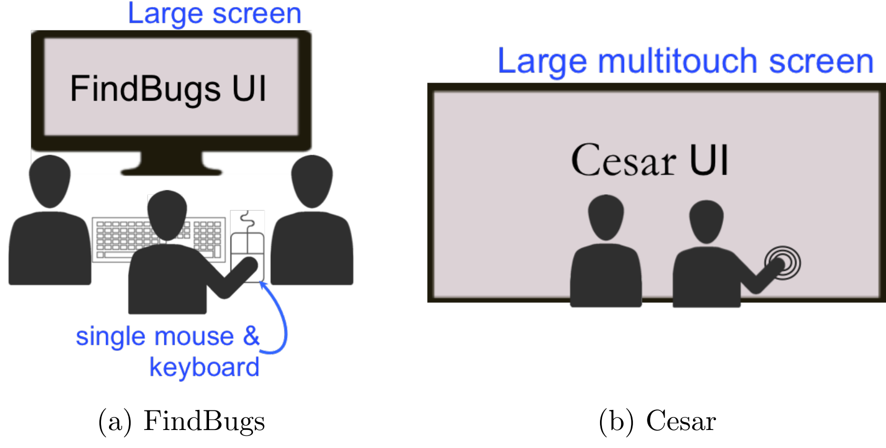
Figure 3.1: Screenshot of FindBugs interface.

and usable security. We refer to FindBugs' evaluator i as $\text{E}i_{FB}$. The session lasted 90 minutes and was voice recorded. Members of the research team were present to observe and take notes during the session. FindBugs UI was displayed on a 47-inch screen using a single mouse and keyboard for interaction (see Figure 3.2a). The session started with running FindBugs analysis on the source code of Apache Tomcat v.6.0.41.[3] Next, the evaluators performed some tasks to explore FindBugs' UI and its warnings of potential vulnerabilities. These tasks include:

- T_1 Choose a package and view its vulnerabilities.
- T_2 How many Security vulnerabilities are there in the codebase?
- T_3 Choose a class and view its number of vulnerabilities.

The evaluators then focused on some warnings and worked towards classifying them as false positives or true vulnerabilities. Finally, the evaluators discussed different UI features they wished to have been available in FindBugs.

[3] `http://tomcat.apache.org/download-60.cgi`



(a) FindBugs (b) Cesar

Figure 3.2: Usability studies session setup.

3.3.2 Results

We found, also aligned with previous research [79], that FindBugs' UI does not adequately support collaboration. The evaluator managing the input devices, $E2_{FB}$, was more engaged in exploring the interface and vulnerabilities than the rest of the evaluators. Communicating ideas was problematic; evaluators tried to draw each other's attention by pointing to the screen, and they sometimes resorted to asking $E2_{FB}$ to point the mouse to what they wanted to discuss by describing its position (*e.g.*, top right of the screen).

The UI has poor fluidity ($-FLUI$),[4] and was hard to navigate. Often the evaluators would be silent while trying to determine how to perform a certain task or how to make sense of the information presented on the screen. In addition, to complete some tasks, the evaluators needed to perform many steps. For example, FindBugs presents potential defects as a tree structure, where the details of the individual defects are accessible via the leaf nodes. For every vulnerability that the evaluators wanted to inspect in detail, they had to click through all the nodes down the tree branch containing that vulnerability. This deterred the evaluators from extensively exploring vulnerability details, and likewise had a negative effect on collaboration between them, as they were very consumed in the steps that they forget to discuss the information they were shown.

[4] A + or − sign before a Cognitive Dimension indicates that the interface is high or low on this dimension, respectively. A + sign throughout this chapter indicates an advantage of the interface.

The default setting of FindBugs does not maintain the codebase hierarchy ($-CLOS$); the tree structure focuses on individual defects rather than on their distribution in the codebase. Granted, the UI allows users to structure the tree by the codebase packages, however this option is hidden in the UI in a way that the evaluators did not discover throughout the entire evaluation session. Although the default hierarchy of the tree structure is adjustable, it was not clear for the evaluators how to perform this task and the role of some elements of the UI was not clear. For example, while trying to adjust the structure of the tree, $E1_{FB}$ said, "*I don't know what the arrow part is. [Does] the arrow means ignore everything to the right?*"

The structure of the tree, focusing on the defects rather than the codebase, swayed the evaluators to become too consumed in the first vulnerability they viewed. This was exacerbated by the fact that in order to view other vulnerabilities, they would have to go through many steps and clicks. This led the evaluators to become absorbed by their attempt to assess a vulnerability without assessing the overall quality of the code, or thinking first about their strategy to evaluate the codebase.

There was confusion related to other aspects of the UI as well, such as the *Rank* and *Confidence* metrics. The Rank of a defect is presented as an integer ranging from 1 to 20, while the Confidence is presented by colours (*e.g.*, if a defect was detected with high confidence, it is marked red). After inspecting the interface, the evaluators deduced that the lower the rank, the more severe the vulnerability. They were inspecting a low rank vulnerability that was marked red. $E2_{FB}$ said, "*So lower [rank] is higher [severity]. Because it's red? and red means bad.*" However, towards the end of the evaluation session, the evaluators noticed another vulnerability that did not match their reasoning. At this point $E2_{FB}$ exclaimed, "*Wait, hold on! I thought lower was higher [severity]! So, maybe the colour and the thing [rank] [aren't] related.*" This confusion resulted from the inconspicuousness of the Confidence metric ($-VIJU$). FindBugs' UI does not explain what the integer values or colours mean, thus the evaluators erroneously linked the colours (*i.e.*, Confidence) to the integer values (*i.e.*, Rank), and attributed them both to the vulnerability's severity. The discussion trying to decipher the meaning of, and relation between, the colours and the integer values ended by $E1_{FB}$ saying, "*I think the rank needs some kind of explanation. What's a 7*

mean? [...] Is there anything in the documentation that would help? [all laughing]."

On the other hand, FindBugs' UI allows users to adjust the size of its components, *e.g.*, they could increase the size of the code pane when they want to focus on the source code. In addition, when a user clicks on the defect leaf node in the tree structure, the UI displays the details of this defect in a separate pane, as well as the source code with the defect lines highlighted. FindBugs also provides the ability to add and save textual annotations to the detected defects.

Although FindBugs allows users to focus on specific defects, it fails to encourage users to develop a strategy for evaluating the overall quality of the code. In addition, it does not adequately support collaboration nor exploration activities.

3.4 Cesar

We designed and implemented Cesar, a prototype aiming to leverage the benefits of SATs (*e.g.*, FindBugs), while addressing their shortcomings. We chose FindBugs for demonstration purposes, however, our general approach could be applied to the results of any SAT. The prototype was developed as a web application using JavaScript and D3.[5] This allows it to be used on many platforms needing only a browser, thus eliminating the need to install more software applications. Cesar offers a visual representation of the output of FindBugs in the form of a treemap [148], where the codebase (sub)packages are interior nodes and the classes are the leaf nodes. The size of a leaf node depends on the number of potential vulnerabilities in the class represented by this leaf node relative to the total number of vulnerabilities in the codebase. In contrast to Goodall *et al.*'s proposal [69], Cesar's treemap maintains the codebase hierarchy to which developers are accustomed to maintain closeness to the programming environment. Cesar uses a large multitouch vertical surface as an interface. Large multitouch displays have shown promising results in supporting and promoting collaboration between team members [18].

Cesar is designed to satisfy the following objectives:

O_1 Support and encourage collaboration

O_2 Encourage exploration

[5] `https://d3js.org`

O_3 Support focusing on the quality of the code as a whole

O_4 Support focusing on details of specific vulnerabilities

The first step to building the prototype was to run FindBugs analysis on a codebase. We analyzed the `Catalina` package of Apache Tomcat written in Java. However, our implementation is extensible to source code in any programming language that is organized in a hierarchical structure, either implicitly through the language (*e.g.*, OOP) or through the programmers' file organization. The result of FindBugs' analysis is an XML file of potential defects in the software analyzed; the file contains each defect's name, its severity and priority, the category under which it falls, and information about its location in the codebase. However, the file is arranged by defect, ignoring the codebase hierarchy. Thus, we extracted the XML file and built a JSON file in the proper format for use by Cesar, maintaining the codebase hierarchy. We also added the description[6] of every detected defect to the JSON file. The current implementation of Cesar visualizes the different types of defects as categorized by FindBugs, however, the same settings apply for visualizing only security vulnerabilities.

Through checkboxes above the treemap visualization pane, Cesar enables developers to choose the categories of issues they want to include in the visualization. Figure 3.3 shows Cesar with all the categories included in the treemap visualization. Gray rectangles show package names, and rectangles in a coloured block below each package represent all the subpackages and classes in this package. Each class is represented by a rectangle. Classes belonging to one package have the same colour. [7] In the presented implementation of Cesar, colours were chosen randomly and only serve to show classes that belong to the same package. In Section 3.6, we discuss how rectangle colour could be used to present more information. The area of a rectangle (*i.e.* a class) in the treemap represents the number of potential issues in the class relative to the total number of issues detected in the codebase. The visualization is interactive, *i.e.*, the treemap view is adjusted in real-time to add/remove each category the user selects/unselects. The relative area of each rectangle is adjusted to show *only* the

[6] `http://findbugs.sourceforge.net/bugDescriptions.html`

[7] Rectangles having the same colour that are not adjacent to each other do not belong to the same package.

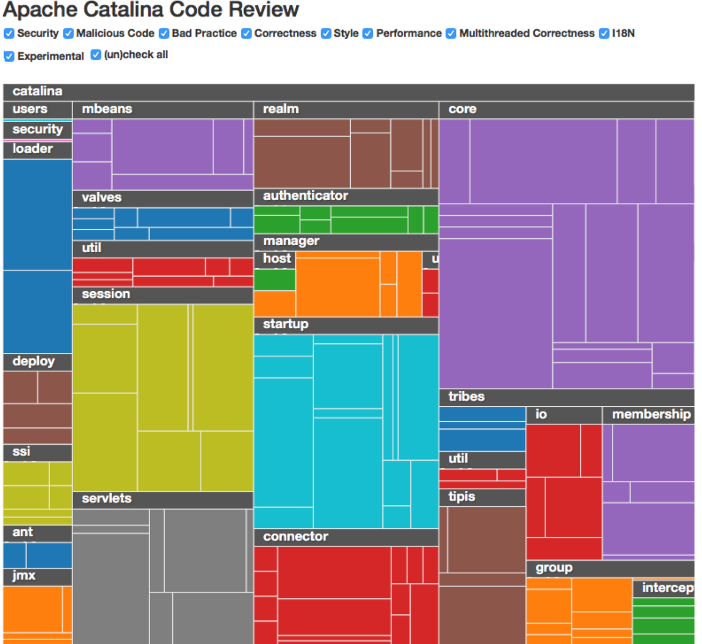


Figure 3.3: Cesar's treemap showing the distribution of selected defect categories in package `Catalina`.

number of issues belonging to the selected categories. Although the relative size of the treemap rectangles change with changing the categories, the rectangles remain in the general vicinity to help developers maintain perspective.

Users can change the view of the treemap to focus on a sub-package by tapping it. The visualization thus zooms-in on that particular package, filling the entire treemap pane with it, as in Figure 3.4 showing sub-package `catalina.realm`. When the user zooms to the class level, they can view the number of defects in each class. Each rectangle is labelled with x of y, where y is the total number of issues in a class and x shows how many of those belong to the selected categories. For example, in Figure 3.4, class `catalina.realm.RealmBase` is labelled (1 of 9), indicating that it has a total of 9 defects, and only one of them is of the selected category (Malicious Code).

When the user performs a long tap on a class, two additional panes appear next to the visualization pane: "details" and "source code". The former lists all defects (in the tapped class) that belong to the selected categories. Defects are grouped by categories, and a brief description of each defect is available. The description is collapsable to

Figure 3.4: Cesar's visualization, details, and source code panes.

save screen space. The "source code" pane displays the class's code, highlighting defect lines. As shown in Figure 3.4, the "details" pane lists the only Malicious Code vulnerability in class `catalina.realm.RealmBase`, whereas the "source code" pane displays its code with the vulnerability line highlighted.

The current version of Cesar created the treemap using FindBugs' set of categories (*e.g.*, Security, Performance). However, this set could be customized to best fit the code review goals. For a security-focused code review tool, the set (Cesar's checkboxes) could include exclusively security vulnerability categories (*e.g.*, Buffer overflow, Cross-site scripting).

3.5 Cesar's Study

To analyze how well Cesar fulfills its objectives, we conducted a usability evaluation of its UI informed by the Cognitive Walkthrough methodology, and used the Cognitive Dimensions as the evaluation framework.

3.5.1 Study Design

We held two independent usability evaluation sessions, each with two evaluators. The four evaluators were recruited participants with industry programming experience and high experience in Java programming. The evaluators performed all the tasks on the

prototype, while members of the research team observed and took notes. The sessions were audio recorded and structured as follows. First, a researcher introduced the study to participants and explained how a Cognitive Walkthrough is conducted, then the participants started working together interacting with the prototype on a 75-inch vertical multitouch screen (see Figure 3.2b) to get an idea of how it works. The researcher then gave participants some tasks to perform. After all the tasks were done, each participant independently filled out a short survey soliciting their opinion of the prototype. The researcher then interviewed the pairs to discuss their opinion of the prototype and their recommendations. Both the survey and interview questions were discussing the Cognitive Dimensions, and were adapted from the Cognitive Dimensions questionnaire [36].

We followed the Cognitive Dimensions framework when designing the study. We categorized activities done using the prototype according to the Cognitive Dimensions framework into *exploratory understanding* aiming to advance developers' understanding of the overall security-level of the program (*e.g.*, identifying packages and classes that contain the most security vulnerabilities) and *searching activities* (*e.g.*, looking for the explanation of the vulnerability type). We divided the tasks into "get to know your system" tasks where participants spend some time exploring how Cesar works, followed by some "specific" tasks that are representative of all that could be conducted using the prototype, and finally some "general" tasks which allows them to reflect more on Cesar's overall purpose.

The "Get to know your system" tasks included:

KT_1 Select (one or more) categories of bugs to see their distribution in the code base

KT_2 Zoom in one package to see the number of bugs

KT_3 Change the bug categories selected to see the difference in the number of bugs

KT_4 Display the source code for a class

KT_5 Find the different bug types present in a class

The "Specific" tasks included:

Find out:

ST_1 how many security vulnerabilities in `catalina.core` package

ST_2 how many packages have security vulnerabilities?

- ST_3 which (sub)package is the least/most vulnerable?
- ST_4 which line of code has Malicious code vulnerabilities in class `catalina.realm.RealBase`?
- ST_5 what does "MS: Field isn't final but should be (MS_SHOULD_BE_FINAL)" mean?

For the "General" tasks, we asked the evaluators to imagine they are in charge of approving the visualized Apache codebase for deployment. We asked them to describe how they would do an appraisal of this software and how would they prioritize which vulnerabilities to fix.

3.5.2 Cesar's Strengths

In this section, we analyze the sessions' outcomes based on the Cognitive Dimensions, and discuss how the prototype fulfills each of its objectives (Section 3.4). In Section 3.6, we discuss improvements to address some of Cesar's weaknesses and enhance the user experience. Cesar's usability evaluator i is referred to as E_i.

Participants rated how Cesar fulfills objectives 2-to-4 on a scale of 1 (*very well*) to 4 (*not at all*).[8] Survey responses were positive, with mean scores of 1 for O_2, 1.75 for O_3, and 1.25 for O_4.

O_1 **Support and encourage collaboration.** In contrast to a personal computer with a mouse and keyboard, the large multitouch interface offered all evaluators the same view and level of control over the interface. In FindBugs, the evaluator managing the input devices was more engaged than the others (Section 3.3.2), whereas Cesar's evaluators were almost equally engaged in interacting with the interface. They were actively discussing their understanding of the different parts of the interface, as well as discussing the steps they thought were necessary to perform the different tasks and the implications of the different types of vulnerabilities. At no point during Cesar's usability evaluation sessions was one evaluator monopolizing the interface while the other stood silent. E_3 mentioned, "*since it's touch, the control is accessible. You don't have to hand over the mouse or anything, so that's good.*" When an evaluator

[8] Participants did not numerically rate Cesar with respect to O_1, rather we rely on the verbal discussion of their opinion and our observations.

successfully reached a desired view, they were keen to return to the main view to show their teammate the steps followed to reach that view. As an evaluator was waiting for their teammate to complete interacting with the interface, they were focusing on the steps taken by their teammate and they would sometimes say words like "aha" expressing that they have discovered something. The evaluators attributed this to the natural feel of the interface and how easy it is to move from one view to another. This implies that Cesar has high fluidity ($+FLUI$). In addition, because the interface did not exhaust the evaluators' working memories ($+LCOG$), *e.g.*, it does not require them to carry information from one step to the next, the evaluators could discuss the steps with their teammates without breaking their train of thought, and so were able to discuss and share information before moving to the next step.

O_2 **Encourage Exploration.** Due to the abstract nature of the visualization ($+ABST$) and the way the treemap structure maintained the codebase hierarchy familiar to developers ($+CLOS$), the evaluators were stimulated to explore the different aspects of the interface as soon as they started interacting with the prototype. In addition, E_2 mentioned, " *[the interface] supports exploration pretty well, just by the nature of touching the things that are on the screen, like that encourages you to go back and look at other categories and compare.*" The flexibility of the system and the fluidity of switching from one view to another ($+FLUI$) helped the evaluators feel comfortable to change a view and explore more. For example, when the task was to find the line number that contains a "Malicious Code" vulnerability in a specific class (ST_4 above), the evaluators returned to the main view after inspecting that class, and looked for the package that had the most "Malicious Code" vulnerabilities even though this was not an assigned task.

The interface reduces cognitive load ($+LCOG$) when performing exploration tasks. For example, the evaluators would explore which packages were most/least vulnerable by looking for the visually biggest/smallest areas in the treemap when only the "Security" category was selected. The prototype thus provides a quick overview as opposed to Findbugs where users would have to go through each package in the tree structure and look for the number of security vulnerabilities in each one. Cesar's evaluators also mentioned that the treemap allowed them to easily discover areas where

there are many problems and to look for different trends in the code. The structure of the interface guided the evaluators to continuously consider the overview along with the detailed view, and the interface induced them to continuously explore available information and ask themselves questions such as "*What happens if we uncheck this vulnerability category?*", "*Why is this package so big?*", and "*Why does this class have all these security vulnerabilities?*".

O_3 **Support focusing on the quality of the code as a whole.** The different packages and classes in the codebase are represented with rectangles in the treemap and the relative area of the rectangles represents the number of potential defects relative to the total number of vulnerabilities in the codebase. This form of abstraction ($+ABST$) helped promote the focus on the overall quality of the codebase. For example, the evaluators were not drawn into individual vulnerabilities before acquiring an overall view of the quality of the codebase. Instead, they initially developed a strategy for appraising the software, and then compared the areas of the rectangles to each other and to the overall area of the visualization in order to begin exploring the most problematic packages. E_1 said, "*I think we did take a step back to think, 'okay what [is] our approach, do we stay here in a details view or do we go back to the overview, or you know should we filter things more, filter things less'. So, we kind of took a step back I think before every task to kind of figure out what our strategy is.*"

We mentioned earlier that the interface guided the evaluators to continuously consider the overview of the code base along with the detailed view. This was accomplished by the ease of starting from the main view containing the package in question and the ability to zoom-in on it to get more details, where the level of zoom depends on the detail depth. For example, in order to perform task ST_4, evaluators started from the main view, showing the `catalina` package, zoom in on the `realm` package, and then display the vulnerabilities and source code of the `RealBase` class. By doing so, the evaluators were able to evaluate the state of the `realm` package with respect to the codebase before digging deep for the more detailed information. The ease of switching between views ($+FLUI$) prevented distraction from the evaluators' objective and focused their attention on the overall code quality. Nevertheless, the interface does not force users to dig through the interface for every task, as it allows for general

exploration tasks, such as finding the most vulnerable package, without delving into details. The evaluators mentioned that the interface was helpful in allowing them to gain a general understanding of how vulnerable the codebase is, and that it does this in a more interesting way than going through a list of potential vulnerabilities. E_4 said, "*I think it's good that it's pictorial and that people can discuss on it [...] It's not a pile of text that I'm going through. It's not a list of errors.*"

The interface allows for some shortcuts, *e.g.*, it allows users to display defect details and source code of any class by a long tap on its leaf node's rectangle from any view. This could be useful, for example, in case a user wants to explore the class that has the most vulnerabilities, which would be represented by the biggest rectangle in the treemap, or for someone who memorized the location of a class on the treemap. We discuss more shortcuts that we anticipate to be useful for advanced users in Section 3.6.

O_4 **Support focusing on details of specific vulnerabilities.** When discussing their strategy for appraising the codebase, the evaluators mentioned how they would use filters provided by the interface and zoom-in on important packages to assess how well the codebase fulfills their coding standards. E_3 mentioned, "*Depends on what our standards are, or what we consider a bug that must be fixed, or one that's maybe not that important,*" whereas E_4 said, "*Maybe I would start with the security issues, maybe after I get all the security [issues] fixed, I would look at another [category], say performance,*" while tapping on the respective checkboxes. By providing filters that allowed them to inspect vulnerability categories that are more critical to them, and by adjusting the visualization in realtime ($+FLUI$) as more categories are (un)checked, the interface allowed the evaluators to narrow down their focus to those critical vulnerabilities. For example, when the evaluators did not want to inspect "Style" issues, they unchecked it to disregard it from their analysis. Maintaining the structure of the codebase hierarchy in the treemap visualization ($+CLOS$) aligned with the evaluators' familiarity with the hierarchical nature of the codebase. This helped the evaluators focus their attention on specific vulnerabilities, rather than trying to resolve to which package a class belonged.

In addition to filtering out irrelevant defects, evaluators consistently checked the

description of the vulnerabilities available through Cesar. However, their behaviour varied; some started looking at the source code, discussing why a specific line was problematic before looking at the provided explanation,while others started with the explanation before checking the code. In either case, they all inspected the vulnerability description area, mentioned that it was useful, and that its placement close to the source code window invited them to consistently refer to it ($+VIJU$).

3.6 Future Enhancements

In this section, we address how Cesar may be improved in future iterations. Although Cesar could be used to visualize any type of defect, our focus here is on security vulnerabilities.

Display the number of vulnerabilities in a package on the treemap. Minimizing vulnerability details displayed by the treemap visualization encouraged the evaluators to inspect the overall code quality, *e.g.*, by allowing them to compare the vulnerability of a package relative to other packages or to the codebase as a whole. However, our evaluators mentioned that displaying the absolute number of vulnerabilities in the package could help them perform some tasks faster ($+LCOG$, $+VIJU$). For example, evaluator E_1 said, "*It would be nice if the number of vulnerabilities that are in a package could be written next to the package [name] [...] When we were looking for the smallest package, we were like "is this the smallest?", "is this the smallest?", whereas if it was just a number, we would have been able to spot it quicker*". Thus, displaying the absolute number of detected vulnerabilities in a package beside its name would further augment users' focus on the overall code quality and speed up some tasks. For consistency, the number beside the package name should follow the same format as that displayed on class rectangles. It should be in the form of (x of y), where x is the number of issues from the selected categories, and y is the total number of vulnerabilities in the package from all categories.

Add a breadcrumbs trail. To further increase the fluidity of the interface, we would use a breadcrumbs trail [106] to trace and display the hierarchy of the user's

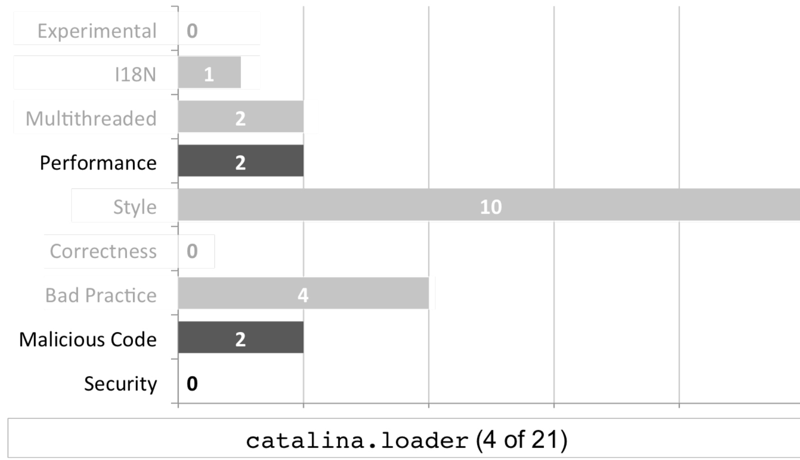


Figure 3.5: A secondary visualization showing the distribution of vulnerabilities in each category. Selected categories are darker.

current treemap view in relation to the codebase structure. This would allow users to switch easily between package levels with a single tap and to quickly identify the location of the package/class in the treemap's current view in relation to the codebase ($+FLUI$). For example, rather than having to tap twice on the screen to zoom-out of the fourth-level sub-package (*e.g.*, package `interceptors` with full path `catalina.tribes.group.interceptors`) to the second level (package `tribes`), the user would tap on the name of the second-level sub-package in the following breadcrumbs trail: *catalina* ▷ *tribes* ▷ *group* ▷ *interceptors*. Although the evaluators did not find it annoying to have to tap on the screen multiple times to reach a higher package, *e.g.*, evaluator E_2 thought that, "*Getting in [zooming-in] is really quick and then getting back up [zooming-out] to the top is pretty quick as well*", we believe that this feature would be particularly useful as a shortcut for more advanced users and especially for large projects with deep package levels.

Use colours to represent data. In addition, as was suggested by the evaluators during both usability evaluation sessions, we could use colours to introduce another dimension of the data visualized by the treemap. For example, rather than using random colours for the rectangles, packages and classes that are more critical to the application, or those that need to be thoroughly investigated against security vulnerabilities (*e.g.*, classes handling databases), would be in warm colours, and the others

in cool colours. Alternatively, packages/classes could be coloured according to the severity of their vulnerabilities, the warmer colours indicting more severe vulnerabilities. This mechanism would provide yet another way for filtering information and minimizing the cognitive load on users (+$LCOG$).

Use a secondary visualization. Finally, to increase the visibility of the vulnerability state of the code, we propose a secondary visualization to show the distribution of different vulnerability categories in the package/class in the current treemap (+$VIJU$). Initially, both visualizations would show the distribution of all vulnerability categories in the codebase as a whole (package `catalina` in our example). In addition, we could employ the *linking and brushing* techniques [33]. The secondary visualization and the treemap could be linked, such that whenever the treemap changes, the secondary visualization could be updated in realtime to match the current treemap. For example, if the user zooms-in on a sub-package, the secondary visualization would be updated to show its distribution. Figure 3.5 depicts the secondary visualization as a bar graph showing the distribution of all the vulnerability categories in the `catalina.loader` sub-package. It shows the absolute number of vulnerabilities in each category on its respective bar, as well as the package name, the number of selected vulnerabilities, and the total number of vulnerabilities (+$LCOG$). The colour of the bars will match that used in the treemap—grey bars for packages and the same colour used in the treemap for classes. In Figure 3.5, the user selected only a subset of vulnerability categories (Security, Malicious code, and Performance); unselected categories are faded out on the bar graph. Focusing on categories can be done by selecting their checkboxes, or using a *shadow highlight brushing* operation on the secondary visualization (+$FLUI$). Brushing can be done by tapping on the category's bar, its name, or by finger-tracing around them (*e.g.*, drawing a circle). This will highlight the selected categories, and will update the linked treemap accordingly. Highlighting information about the selected vulnerability categories allows users to focus on them, while the background information helps users maintain cognizance of the overall code quality. We note that in some cases, a category with a particularly large number of vulnerabilities will occupy most of the bar, thus making it hard to see the rest of the categories. Thus, the secondary visualization will enable users to

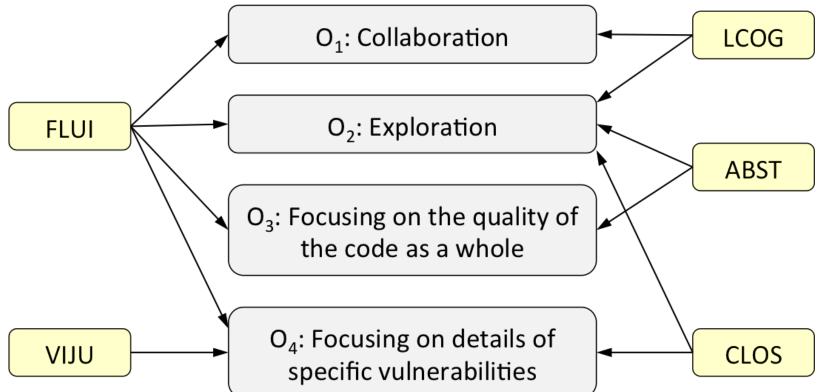


Figure 3.6: Relation between Cognitive Dimensions and four select objectives of a CSCR tool. The figure can be read following the arrows, *e.g.*, the arrow from FLUI to Collaboration indicates that Fluidity supports Collaboration.

zoom-in/out to focus on the desired categories.

3.7 Discussion

In this section, we reflect on the results from our evaluations to discuss the effect of five Cognitive Dimensions on achieving Cesar's objectives (see Figure 3.6), and provide general recommendations for collaborative code review tools based on the Cognitive Dimensions framework.

Fluidity ($FLUI$). Fluidity is useful for *all* four objectives of a collaborative code review tool. A fluid interface does not compel the user to perform many actions to fulfill a certain task, thus reduces disruptions to the code review workflow. In particular, without fluidity, the user would be consumed in the many steps required to perform a task, negatively affecting collaboration between team members, and distracting them from the overall code quality and from analyzing specific vulnerabilities. A user absorbed by the task at hand is less likely to have discussions with their teammates that might break their line of thought, and they might be reluctant to redo all these steps to teach their teammate how to perform a certain task. In addition, being distracted by the steps they need to perform takes the user's focus off the original objective of assessing the quality of the code and inspecting vulnerabilities. A system with poor fluidity would also deter users from exploring—there is only so many times a user

may be willing to go through multiple steps to explore information. As evident by Cesar's usability evaluations, the UI of a collaborative code review tool should be fluid enough to invite users to explore the different aspects of the data presented to discover hidden trends, rather than settling for the most obvious explanations.

Recommendation. It is important for a code review tool to have fluidity. The tool should act as a transparent interaction medium; users should feel as if they are directly interacting with their codebase, rather than the burden of learning a new code review tool.

Low Cognitive Load (*LCOG*). A tool that reduces the cognitive load on users is valuable for collaboration and exploration. Useful information should be available to the user throughout the different steps taken to perform a task. Reducing the load on the user's working memory allows the user to be more open to discussion and exploration. A tool that provides users with opportunities for exploration without exerting too much cognitive effort is more likely to succeed in persuading them to perform deeper analyses.

Recommendation. A code review tool should conserve the user's cognitive resources to the actual objective of reviewing their codebase, determining how to secure it, and identifying critical issues.

Abstraction (*ABST*). Using the correct level of abstraction to present vulnerability information supports exploration and helps the user assess the overall quality of the codebase. Providing vulnerability information in an abstract form, while maintaining closeness of mapping as discussed below, encourages the user to explore available information and invokes their analysis mindset. Additionally, the difference in the approach taken by the Findbugs and Cesar's evaluators demonstrates the usefulness of abstraction in preventing users from getting engrossed in the details of specific vulnerabilities without being mindful of the big picture.

Recommendation. A collaborative code review tool should use abstractions to invite the user to explore available information and to apprise the user of the overall quality of the codebase.

Closeness of Mapping ($CLOS$). Closeness of mapping of the interface elements to the domain motivates users to explore information provided by the code review tool, and concentrate on vulnerability details. Thus, a visualization that maintains the codebase hierarchy familiar to the developers allows them to build on their existing knowledge of the codebase to review their code, looking for hidden patterns and trends. In addition, it allows them to focus on the details of vulnerabilities, rather than *deciphering* how the vulnerability information presented to them relates to their code.

Recommendation. A code review tool should maintain closeness of representation to the domain of software development, *e.g.*, by maintaining the codebase hierarchy in its visualizations. Users who are familiar with the structure of the information do not have to go through the first steps of determining how to make sense of the presented data, and could delve right into analyzing this data utilizing their existing knowledge.

Visibility and Juxtaposability ($VIJU$). This dimension supports focusing on details of specific vulnerabilities. As noted by Cesar's evaluators, when vulnerability information (such as its description and location in the codebase) is easily accessible to the user without cognitive effort, the user can focus their attention on the specifics of the vulnerability, identifying its criticality and how to solve it. Components that are relevant to each other should be placed side-by-side to allow the user to easily access the required information, making comparisons and inferences.

Recommendation. Components' visibility and their placement juxtaposed allows users to direct their cognitive resources to investigating vulnerability details, making inferences and decisions, rather than on finding information in disparate parts of the interface.

3.8 Limitations

Our usability evaluations of FindBugs and Cesar were done in a lab setting, which may not necessarily reflect real-life scenarios. Participants evaluating the usability of FindBugs and Cesar were not currently employed as developers, however, all of

our participants had industry programming experience. Finally, participants were not familiar with the codebase analyzed, which could have influenced our results. However, we did not focus on analyzing specific vulnerabilities, rather our study explored participants' general behaviour using the tools analyzed and how these tools supported collaboration and exploration. In future studies, it would be interesting to use a codebase with which participants are familiar.

3.9 Summary

We applied a user-centered approach to address the issue of usability of source code analyzers. Usability evaluation was two fold: using the Cognitive Dimensions framework and informed by the Cognitive Walkthrough method. We evaluated the usability of the UI of FindBugs, one of the most popular open source code analyzers. To address some of FindBugs' usability issues we designed Cesar, which provides developers and testers with a visual analysis environment to help them reduce risks of source code security vulnerabilities. Cesar uses a vertical multitouch display as an interface, and a treemap as its primary visualization element. The treemap presents vulnerability information while maintaining the codebase hierarchy familiar to developers. We evaluated the usability of an initial Cesar prototype, and discussed additional potentially useful features based on the evaluation. Our analysis shows that the prototype is promising in promoting collaboration, exploration, and enabling developers to focus on the overall quality of their code as well as inspect individual vulnerabilities. Finally, we presented general recommendations for designing collaborative code review tools.

While working on the research presented in this chapter, we recognized that the problem of software security is a larger one, relating to the whole process of integrating security in the development lifecycle. We recognized that usable security research is mainly focused on security tools and methodologies, which is but a single aspect of the process. In the following chapter, we present our work focusing on the overall security process.

Chapter 4

Security in the Software Development Lifecycle

Despite current efforts, security vulnerabilities are discovered daily and threats are increasing and changing [77], extending even to small companies [153]. Developers are often viewed as “the weakest link in the chain” and are blamed for security vulnerabilities [71,178]. However, simply expecting developers to keep investing more efforts in security is unrealistic and unlikely to be fruitful [13].

To understand reasons for the persistence of software vulnerabilities, we designed a qualitative study to explore steps that teams are taking to ensure the security of their applications, how developers’ security knowledge influences the process, and how security fits in (and sometimes conflicts with) the development workflow. We interviewed 13 developers who described their tasks, their priorities, as well as tools they use. During the data analysis, we recognized that our participants’ practices and attitudes towards security formed two groups, each with trends distinguishable from the other group. On comparing real-life security practices to best practices, we also found significant deviations.

This chapter makes the following contributions.

- We present a qualitative study looking at real-life practices employed towards software security.
- We amalgamate software security best practices extracted from the literature into a concise list to assist further research in this area.
- We reflect on how well current security practices follow best practices, identify significant pitfalls, and explore why these occur.
- Finally, we discuss opportunities for future research.

This work is published at the Symposium on Usable Privacy and Security (SOUPS). USENIX Association, 2018. [20]

4.1 Study Design and Methodology

In this section, we present the study design, participant demographics, analysis methodology, and limitations.

4.1.1 Interview Study Design

We designed a semi-structured interview study and received REB clearance. The interviews targeted 5 main topics: general development activities, attitude towards security, security knowledge, security processes, and software testing activities (see Appendix A for the interview script). To recruit participants, we posted on development forums and relevant social media groups, and announced the study to professional acquaintances. We recruited 13 participants; each received a $20 Amazon gift card for participation. Before the one-on-one interview, participants filled out a demographics questionnaire. Each interview lasted approximately 1 hour, was audio recorded, and later transcribed for analysis. Interviews were conducted in person ($n = 3$) or through VOIP/video-conferencing ($n = 10$). Data collection was done in 3 waves, each followed by preliminary analysis and preliminary conclusions [67]. We followed Glaser and Strauss's [67] recommendation by concluding recruitment on saturation (*i.e.*, when new data collection does not add new themes or insights to the analysis).

4.1.2 Participant Demographics

A project team consist of teams of developers, testers, and others involved in the SDLC. Smaller companies may have only one project team, while bigger companies may have different project teams for different projects. We refer to participants with respect to their project teams; team i is referred to as Ti and P-Ti is the participant from this team. We did not have multiple volunteers from the same company. Our data contains information from 15 teams in 15 different companies all based in North America; one participant discussed work in his current (T7) and previous (T8) teams, another discussed his current work in T10 and his previous work in T11. In our dataset, seven project teams build web applications and services,

Table 4.1: Participant demographics

	Participant					Company and team	
Participant ID	Gender	Age	Years	Title	SK	Company size	Team size[1]
P-T1	F	30	1	Software engineer	4	Large enterprise	20
P-T2	M	34	15	Software engineer	5	Large enterprise	12
P-T3	M	33	10	Software engineer	4	Large enterprise	10
P-T4	M	38	21	Software developer	4	SME	7
P-T5	M	34	12	Product manager	5	Large enterprise	7
P-T6	F	26	3	Software engineering analyst	3	Large enterprise	12
P-T7, P-T8⋆	M	33	4	Senior web engineer	4	SME – n/a⋆	3
P-T9	M	34	5	Software developer	3	Large enterprise	20
P-T10, P-T11⋆	M	33	8	Software engineer	2	SME – SME⋆	5
P-T12	M	37	20	Principal software engineer	5	SME	10
P-T13	M	38	15	Senior software developer	2	SME	8
P-T14	M	26	3	Software developer	2	SME	4
P-T15	F	27	5	Junior software developer	4	Large enterprise	7

Years: years of experience in development
SK: self-rating of security knowledge 1(no knowledge) - 5(expert). SME: Small-Medium Enterprise
⋆: indicates participant's previous company. [1] Team size for the current company

such as e-finance, online productivity, online booking, website content management, and social networking. Eight teams deliver other types of software,*e.g.*, embedded software, kernels, design and engineering software, support utilities, and information management and support systems. This classification is based on participants' self-identified role and products with which they are involved, and using Forward and Lethbridge's [63] software taxonomy. Categorizing the companies to which our teams belong by number of employees [34], seven teams belong to SMEs (T4, T7, T10–T14) and eight teams belong to large enterprises (T1–T3, T5, T6, T8, T9, T15). All participants hold university degrees which included courses in software programming, and are currently employed in development with an average of 9.35 years experience ($Md = 8$). We did not recruit for specific software development methodologies. Some participants indicated following a waterfall model or variations of Agile. See Table 4.1 for participant demographics.

4.1.3 Analysis

Data was analyzed using the Qualitative Content Analysis methodology [52, 80]. It can be deductive, inductive, or a combination thereof. For the deductive approach, the researcher uses her knowledge of the subject to build an analysis matrix and

codes data using this matrix [80]. The inductive method, used when there is no existing knowledge of the topic, includes open coding, identifying categories, and abstraction [80].

We employed both the deductive and inductive methods of content analysis. The deductive method was used to structure our analysis according to the different development stages. We built an initial analysis matrix of the main SDLC stages [152]. After a preliminary stage of categorizing interview data and discussions between the researchers, the matrix was refined. The final analysis matrix defines the stages of development as follows. *Design* is the stage where the implementation is conceptualized and design decisions are taken; *Implementation* is where coding takes place; *Developer testing* is where testing is performed by the developer; *Code analysis* is where code is analyzed using automated tools, such as SATs; *Code review* is where code is examined by an entity other than the developer; *Post-development testing* is where testing and analysis processes taking place after the developer has committed their code.

We coded interview data with their corresponding category from the final analysis matrix, resulting in 264 unique excerpts. Participants talked about specific tasks that we could map to the matrix stages, despite the variance in development methodologies. We then followed an inductive analysis method to explore practices and behaviours within each category (development stage) as recommended by the content analysis methodology. We performed open coding of the excerpts where we looked for interesting themes and common patterns in the data. This resulted in 96 codes. Next, data and concepts that belonged together were grouped, forming sub-categories. Further abstraction of the data was performed by grouping sub-categories into generic categories, and those into main categories. The abstraction process was repeated for each stage of development. As mentioned earlier, during our analysis we found distinct differences in attitudes and behaviours that were easily distinguishable into two groups, we call them *the security adopters* and *the security inattentive*. We thus present the emerging themes and our analysis of the two groups independently. Figure 4.1 shows an example of the abstraction process for developer testing data for the security adopters. While all coding was done by a single researcher, two researchers

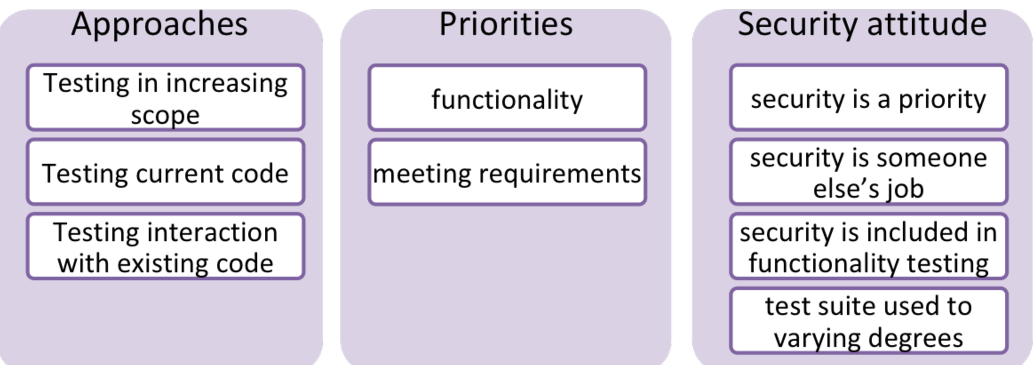


Figure 4.1: Security adopters: developer testing abstraction

met regularly to thoroughly and collaboratively review and edit codes, and group and interpret the data. To verify the reliability of our coding, we followed best practices by inviting a researcher who has not been involved with the project to act as a second coder, individually coding 30% of the data. We calculated Krippendorff's alpha [88] to assess inter-rater reliability, and $\alpha = 0.89$ (percentage of agreement = 91%). According to Krippendorff [89], $alpha \geq 0.80$ indicates that coding is highly reliable and that data is "similarly interpretable by researchers". In case of disagreements, we had a discussion and came to an agreement on the codes.

4.1.4 Limitations

Our study included a relatively small sample size, thus generalizations cannot be made. However, our sample size followed the concept of saturation [67]; participant recruitment continued until no new themes were emerging. Additionally, recruiting participants through personal contacts could result in biasing the results. While we cannot guarantee representativeness of a larger population, the interviewer previously knew only 3/13 participants. The remaining ten participants were previously unknown to the researcher and each represented a different company. While interviews allowed us to explore topics in depth, they presented one perspective on the team. Our data may thus be influenced by participants' personal attitudes and perspectives, and may not necessarily reflect the whole team's opinions. However, we found that participants mainly described practices as encouraged by their teams.

4.2 Results: Security in Practice

We assess the degree of security integration in each stage of the SDLC as defined by our final analysis matrix. As mentioned earlier, we found differences in participants' attitudes and behaviours towards security that naturally fell into two distinct groups. We call the first group the *security adopters*: those who consider security in the majority of development stages (at least four stages out of six[1]). The second group who barely considered security or did not consider it at all form the *security inattentive*. We chose the term *inattentive*, as it encompasses different scenarios that led up to poor security approaches. These could be that security was considered and dismissed or it was not considered at all, whether deliberately or erroneously. Table 4.2 presents two tables, one for each group identified in our dataset. We classified practices during a development stage as:

(●) *secure*: when security is actively considered, *e.g.*, when developers avoid using deprecated functions during the implementation stage.

(○) *somewhat secure*: when security is not consistently considered, *e.g.*, when threat analysis is performed only if someone raises the subject.

(×) *not secure*: when security is not part of this stage, *e.g.*, when developers do not perform security testing.

(⊗) *not performed*: when a stage is not part of their SDLC (*i.e.*, considered not secure).

(?): when a participant did not discuss a stage during their interview, therefore denoting missing data.

Table 4.2 highlights the distinction in terms of security between practices described by participants from the security adopters and the security inattentive groups. The overwhelming red and orange table for the security inattentive group visually demonstrates their minimal security integration in the SDLC. Particularly, comparing each stage across all teams shows that even though the security adopters are not consistently secure throughout the SDLC, they are generally more attentive to security than the other group. The worst stage for the security inattentive group is *Code*

[1] At least three stages in cases where we have information about four stages only. Note that this is just a numeric representation and the split actually emerged from the data.

Table 4.2: The degree of security in the SDLC. The tables show participants' teams and the applications they develop according to the taxonomy in [63].

●: secure, ○: somewhat secure, ×: not secure, ⊗: not performed, ? : no data

(a) The Security Adopters

Application		Design	Implementation	Developer testing	Code analysis	Code review	Post-dev testing
embedded software	T1	×	●	×	●	●	●
design and engineering software	T3	?	●	?	●	●	●
design and engineering software	T5	●	●	○	●	●	●
info. management & decision support	T11	?	●	○	●	●	?
support utilities	T12	×	●	○	●	●	●
support utilities	T14	×	●	●	⊗	●	●

(b) The Security Inattentive

Application		Design	Implementation	Developer testing	Code analysis	Code review	Post-dev testing
kernels	T2	×	●	×	○	○	●
website content management	T4	○	○	○	⊗	○	○
e-finance	T6	×	○	×	×	○	○
online productivity	T7	×	×	×	⊗	×	○
social networking	T8	×	×	×	⊗	×	●
embedded software	T9	●	●	○	⊗	○	○
online booking	T10	○	○	×	⊗	○	⊗
online productivity	T13	×	●	×	⊗	○	●
online productivity	T15	×	×	×	×	×	×

analysis, which is either not performed or lacks security, followed by the *developer testing* stage, where security consideration is virtually non-existent.

We initially suspected that the degree of security integration in the SDLC would be directly proportional to the company size. However, our data suggests that it is not necessarily an influential factor. In fact, T14, the team from the smallest company in our dataset, is performing much better than T6, the team from the largest company in the security inattentive group. Additionally, we did not find evidence that development methodology influenced security practices.

Although our dataset does not allow us to make conclusive inferences, it shows an alarming trend of low security adoption in many of our project teams. We now discuss data analysis results organized by the six SDLC stages defined in our analysis

matrix. All participants discussed their teams' security policies, as experienced from their perspectives, and not their personal preferences. Results, therefore, represent the reported perspectives of the developers in each team.

4.2.1 Exploring Practices by Development Stage

We found that the prioritization of security falls along a spectrum: at one end, security is a main priority, or it is completely ignored at the other extreme. For each SDLC stage, we discuss how security was prioritized, present common trends, and highlight key messages from the interviews. Next to each theme we indicate which group contributed to its emergence: `(SA)` for the security adopters, `(SI)` for the security inattentive, and `(SA/SI)` for both groups. Table 4.3 provides a summary of the themes.

I. Design Stage

We found a large gap in security practices described by our participants in the design stage. This stage saw teams at all points on the security prioritization spectrum, however, most participants indicated that their teams did not view security as part of this stage. Our inductive analysis revealed three emerging themes reflecting security prioritization, with one theme common to both the security adopters and the security inattentive, and one exclusive to each group.

Security is not considered in the design stage. `(SA/SI)` Most participants indicated that their teams did not apply security best practices in the design stage. Although they did not give reasons, we can infer from our data (as discussed in other stages) that this may be because developers mainly focus on their functional design task and often miss security [119], or because they lack the expertise to address security. As an example of the disregard for security, practices described by one participant from the security inattentive group violates the recommendation of simple design; they intentionally introduce complexity to avoid rewriting existing code, and misuse frameworks to fit their existing codebase without worrying about introducing vulnerabilities. P-T10 explained how this behaviour resulted in highly complex code, "*Everything is so convoluted and it's like going down rabbit holes, you see their code*

and you are like 'why did you write it this way?' [...] It's too much different custom code that only those guys understand." Such complexity increases the potential for vulnerabilities and complicates subsequent stages [145]; efforts towards evaluating code security may be hindered by poor readability and complex design choices.

Security consideration in the design stage is adhoc. `(SI)` Two developers said their teams identify security considerations within the design process. In both cases, the design is done by developers who are not necessarily formally trained in security. Security issue identification is adhoc, *e.g.*, if a developer identifies a component handling sensitive information, this triggers some form of threat modelling. In T10, this takes the form of discussion in a team meeting to consider worst case scenarios and strategies for dealing with them. In T4, the team self-organizes with the developers with most security competence taking the responsibility for designing sensitive components. P-T4 said, "*Some developers are assigned the tasks that deal with authorization and authentication, for the specific purpose that they'll do the security testing properly and they have the background to do it.*" In these two teams, security consideration in the design stage lies in the hands of the developer with security expertise; this implies that the process is not very robust. If this developer fails to identify the feature as security-sensitive, security might not be considered at all in this stage.

Security design is very important. `(SA)` Contrary to all others, one team formally considers security in this stage with a good degree of care. P-T5 indicated that his team considers the design stage as their first line of defense. Developers from his team follow software security best practices [104, 123, 145], *e.g.*, they perform formal threat modelling to generate security requirements, focus on relevant threats, and inform subsequent SDLC stages. P-T5 explains the advantages of considering security from this early stage, "*When we go to do a further security analysis, we have a lot more context in terms of what we're thinking, and people aren't running around sort of defending threats that aren't there.*"

II. Implementation Stage

Most participants showed general awareness of security during this stage. However, many stated that they are not responsible for security and they are not required to secure their applications. In fact, some developers reported that their companies do not expect them to have any software security knowledge. Our inductive analysis revealed three themes regarding security prioritization in this stage.

Security is a priority during implementation. `(SA/SI)` All security adopters and two participants from the security inattentive group discussed the importance of security during the implementation stage. They discussed how the general company culture encourages following secure implementation best practices and using reliable tools. Security is considered a developer's responsibility during implementation, and participants explained they are conscious about vulnerabilities introduced by errors when writing code.

Developers' awareness of security is expected when implementing. `(SA/SI)` For those prioritizing security, the majority of security adopters and one participant from the security inattentive group are expected to stay up-to-date on vulnerabilities, especially those reported in libraries or third-party code they use. The manner of information dissemination differs and corroborates previous research findings [179]. Some have a structured approach, such as that described by P-T1, "*We have a whole system. Whenever security vulnerability information comes from a third-party, [a special team follows] this process: they create an incident, so that whoever is using the third-party code gets alerted that, 'okay, your code has security vulnerability', and immediately you need to address it.*" Others rely on general discussions between developers, *e.g.*, when they read about a new vulnerability. Participants did not elaborate on if and how they assess the credibility and reliability of information sources. The source of information could have a considerable effect on security; previous research found that relying on informal programming forums might lead to insecure code [11]. In Xiao *et al.*'s [179] study, developers reported taking the information source's thoroughness and reputation into consideration to ensure trustworthiness.

Security is not a priority during implementation. `(SI)` On the other end of the security prioritization spectrum, developers from the security inattentive group

prioritize functionality and coding standards over security. Their primary goal is to satisfy business requirements of building new applications or integrating new features into existing ones. Some developers also follow standards for code readability and efficiency. However, security is not typically considered a developer's responsibility, to the extent that there are no consequences if a developer introduces a security vulnerability in their code. P-T7 explained, "*If I write a bad code that, let's say, introduced SQL injection, I can just [say] 'well I didn't know that this one introduces SQL injection' or 'I don't even know what SQL injection is'. [...] I didn't have to actually know about this stuff [and] nobody told me that I need to focus on this stuff.*" This statement is particularly troubling given that P-T7 has security background, but feels powerless in changing the perceived state of affairs in his team.

Our analysis also revealed that some developers in the security inattentive group have incomplete mental models of security. This led to the following problematic manifestations, which could explain their poor security practices.

Developers take security for granted. `(SI)` We found, aligning with previous research [119], that developers fully trust existing frameworks with their applications' security and thus take security for granted. Our study revealed that these teams do not consider security when adopting frameworks, and it is unclear if, and how, these frameworks' security is ever tested. To partially address this issue, T4 built their own frameworks to handle common security features to relieve developers of the burden of security. This approach may improve security, however verifying frameworks' security is an important, yet missing, preliminary step.

Developers misuse frameworks. `(SI)` Despite their extreme reliance on frameworks for security, developers in T10 do not always follow their recommended practices. For example, although P-T10 tries to follow them, other developers in his team do not; they occasionally overlook or work-around framework features. P-T10 explains, "*I have expressed to [the team] why I am doing things the way I am, because it's correct, it's the right way to do it with this framework. They chose to do things a completely different way, it's completely messed up the framework and their code. They don't care, they just want something that they feel is right, and you know, whatever.*" Such framework misuse may result in messy code and could lead to potential

vulnerabilities [145]. Although frameworks have shown security benefits [143], it is evident that the manner by which some teams are currently using and relying on them is problematic.

Developers lack security knowledge. `(SI)` Developers from the security inattentive group vary greatly in their security knowledge. Some have haphazard knowledge; they only know what they happen to hear or read about in the news. Others have formed their knowledge entirely from practical experience; they only know what they happen to come across in their work. Developers' lack of software security knowledge could explain why some teams are reluctant to rely on developers for secure implementation. P-T7 said, "*I think they kind of assume that if you're a developer, you're not necessarily responsible for the security of the system, and you [do] not necessarily have to have the knowledge to deal with it.*" On the other hand, some developers have security background, but do not apply their knowledge in practice, as it is neither considered their responsibility nor a priority. P-T7 said, "*I recently took an online course on web application security to refresh my knowledge on what were the common attacks on web applications [...] So, I gained that theoretical aspect of it recently and play[ed] around with a bunch of tools, but in practice I didn't actually use those tools to test my software to see if I can find any vulnerability in my own code because it's not that much of a priority.*"

Developers perceive their security knowledge inaccurately. `(SI)` We identified a mismatch between developers' perception of their security knowledge and their actual knowledge. Some developers do not recognize their secure practices as such. When asked about secure coding methods, P-T6 said, "*[The] one where we stop [cross-site scripting]. That's the only one I remember I explicitly used. Maybe I used a couple of other things without knowing they were security stuff.*" In some instances, our participants said they are not addressing security in any way. However, after probing and asking more specific questions, we identified security practices they perform which they did not relate to security.

Furthermore, we found that some developers' mental model of security revolves mainly around security functions, such as using the proper client-server communication protocol. However, conforming with previous research [179], it does not include

software security. For example, P-T9 assumes that following requirements generated from the design stage guarantees security, saying "*if you follow the requirements, the code is secure. They take those into consideration.*" However, he mentioned that requirements do not always include security. In this case, and especially by describing requirements as a definite security guarantee, the developer may be referring to security functions (*e.g.*, using passwords for authentication) that he would implement as identified by the requirements. However, the developer did not discuss vulnerabilities due to implementation mistakes that are not necessarily preventable by security requirements.

Our study also revealed the following incident which illustrates how **Vulnerability discovery can motivate security** `(SI)` and improve mental models. Developers in T13 became more security conscious after discovering a vulnerability in their application. P-T13 said, "*We started making sure all of our URLs couldn't be manipulated. [..] If you change the URL and the information you are looking at, [at the] server side, we'd verify that the information belongs to the site or the account you are logged in for.*" Discovering this vulnerability was eye-opening to the team; our participant said that they started thinking about their code from a perspective they had not been considering and they became aware that their code can have undesirable security consequences. In addition, this first-hand experience led them to the knowledge of how to avoid and prevent similar threats.

III. Developer Testing Stage

Across the vast majority of our participants, whether adopters or inattentive, security is lacking in the developer testing stage. Functionality is developers' main objective; they are blamed if they do not properly fulfil functional requirements, but their companies do not hold them accountable if a security vulnerability is discovered. P-T7 said, "*I can get away with [introducing security bugs] but with other things like just your day-to-day developer tasks where you develop a feature and you introduce a bug, that kind of falls under your responsibility. Security doesn't.*" Thus, any security-related efforts by developers are viewed as doing something extraordinary. For example, P-T2 explained, "*If I want to be the hero of the day [and] I know there's*

a slight possible chance that these can be security vulnerabilities, [then] I write a test and submit it to the test team." We grouped participants' approaches to security during this stage into four categories.

Developers do not test for security. `(SA/SI)` The priority at this stage is almost exclusively functionality; it increases in scope until the developer is satisfied that their code is fulfilling functional requirements and does not break any existing code. And even then, these tests vary in quality. Some developers perform adhoc testing or simply test as a sanity check where they only verify positive test cases with valid input. Others erroneously, and at times deliberately, test only ideal-case scenarios and fail to recognize worst-case scenarios. The majority of developers do not view security as their responsibility in this stage; instead they are relying on the later SDLC stages. P-T2 said, "*I usually don't as a developer go to the extreme of testing vulnerability in my feature, that's someone else's to do. Honestly, I have to say, I don't do security testing. I do functional testing.*" The participant acknowledged the importance of security testing, however, this task was considered the testing team's responsibility as they have more knowledge in this area.

Security is a priority during developer testing. `(SA)` As an exception, our analysis of P-T14's interview indicates that his company culture emphasizes the importance of addressing security in this stage. His team uses both automated and manual tests to ensure that their application is secure and is behaving as expected. P-T14's explained that the reason why they prefer to incorporate security in this stage was that it is more cost efficient to address security issues early in the SDLC. He explained, "*We have a small company, so it's very hard to catch all the bugs after release.*"

Developers test for security fortuitously. `(SA)` In other cases, security is not completely dismissed, yet it is not an explicit priority. Some security adopters run existing test suites that may include security at varying degrees. These test suites include test cases that any application is expected to pass, however, there is not necessarily a differentiation between security and non-security tests. Some developers run these tests because they are required to, without actual knowledge of their purpose. For example, P-T3 presumes that since his company did not have

security breaches, security must be incorporated in existing test suites. He explained, "*[Security] has to be there because basically, if it wasn't, then our company would have lots of problems.*"

Developers' security testing is feature-driven. `(SI)` In another example where security is not dismissed, yet not prioritized, one participant from the security inattentive group (out of the only two who perform security testing), considers that security is not a concern as his application is not outward facing, *i.e.*, it does not involve direct user interaction. P-T9 explained, "*Security testing [pause] I would say less than 5%. Because we're doing embedded systems, so security [is] pretty low in this kind of work.*" While this may have been true in the past, the IoT is increasingly connecting embedded systems to the Internet and attacks against these systems are increasing [72]. Moreover, classifying embedded systems as relatively low-risk is particularly interesting as it echoes what Schneier [25] described as a road towards "a security disaster". On the other hand, P-T4 explained that only features that are classified as sensitive in the design stage are tested, due to the shortage in security expertise. As the company's only developer with security background, these features are assigned to P-T4. Other developers in T4 do not have security experience, thus they do not security-test their code and they are not expected to.

IV. Code Analysis Stage

Eight developers reported that their teams have a mandatory code analysis stage. Participants from the security adopters group mentioned that the main objectives in this stage is to verify the code's conformity to standards and in-house rules, as well as detect security issues. On the other hand, participants from the security inattentive group generally do not perform this stage, and rarely for security.

Security is a priority during code analysis. `(SA)` All security adopters who perform this stage reported that security is a main component of code analysis in their team. T5 mandates analysis using multiple commercial tools and in-house tools before the code is passed to the next stage. T3 has an in-house tool that automates the process of analysis to help developers with the burden of security. P-T3 explained, "*[Our tool] automatically does a lot of that for us, which is nice, it does static analysis,*

things like that and won't even let the code compile if there are certain requirements that are not met." One of the advantages of automating security analysis is that security is off-loaded to the tools; P-T3 explains that security "*sort of comes for free*".

Security is a secondary objective during code analysis. `(SI)` P-T2 explained that in his team, developers' main objective when using a SAT is to verify conformity to industry standards. Although they might check security warnings, other security testing methods are considered more powerful. P-T2 explained, "*[SAT name] doesn't really look at the whole picture. [...] In terms of: is it similar to a security vulnerability testing? No. Pen testers? No. It's very weak.*" In addition to the lack of trust in SATs' ability to identify security issues, and similar to previous research (*e.g.*, [79]), our participants complained about the overwhelming number false positives and irrelevant warnings.

Developers rarely perform code analysis, never for security. `(SI)` Code analysis is not commonly part of the development process for the security inattentive group. According to their developers, T2, T6, and T15 use SATs, but not for security. Code analysis is performed as a preliminary step to optimize code and ensure readability before the code review stage, with no consideration to security.

Reasons for underusing SATs were explored in other contexts [79]. The two main reasons in our interviews were that their use was not mandated or that developers were unaware of their existence. We found that **Developers vary in awareness of analysis tools.** `(SI)` In addition to those unaware, some developers use SATs without fully understanding their functionality. P-T10 does not use such tools since it is not mandated and his teammates are unlikely to do so. He said, "*I know that there's tools out there that can scan your code to see if there's any vulnerability risks [...] We are not running anything like that and I don't see these guys doing that. I don't really trust them to run any kind of source code scanners or anything like that. I know I'm certainly not going to.*" Despite his awareness of the potential benefits, he is basically saying *no one else is doing it, so why should I?* Since it is not mandatory or common practice, running and analyzing SATs reports would add to the developer's workload without recognition for his efforts.

V. Code Review Stage

Most security adopters say that security is a primary component in this stage. Reviewers examine the code to verify functionality and to look for potential security vulnerabilities. P-T14 explained, "*We usually look for common mistakes or bad practices that may induce attack vectors for hackers such as, not clearing buffers after they've been used. On top of that, it's also [about the] efficiency of the code.*"

Contrarily, the security inattentive discount security in this stage—security is either not considered, or is considered in an informal and adhoc way and by unqualified reviewers. Code review can be as simple as a sanity check, or a walkthrough, where developers explain their code to other developers in their team. Some reviewers are thorough, while others consider reviews a secondary task, and are more inclined to accept the code and return to their own tasks. P-T10 explained, "*Sometimes they just accept the code because maybe they are busy and they don't want to sit around and criticize or critically think through everything.*" Moreover, reviewers in T9 examine vulnerabilities to assess their impact on performance. P-T9 explained, "*[Security in code review is] minimum, I'd say less than 5%. So, yeah you might have like buffer overflow, but then for us, that's more of the stability than security issue.*" We grouped participants' descriptions of the code review stage into four distinct approaches.

Code review is a formal process that includes security. `(SA)` All security adopters mentioned that their teams include security in this stage. For some teams, it is a structured process informed by security activities in previous stages. For example, security-related warnings flagged during the code analysis phase are re-examined during code reviews. Reviewers can be senior developers, or an independent team. Being independent, reviewers bring in a new perspective, without being influenced by prior knowledge, such as expected user input. P-T5 said, "*We do require that all the code goes through a security code review that's disconnected from the developing team, so that they're not suffered by that burden of knowledge of 'no one will do this', uh, they will.*" Sometimes reviewers might not have adequate knowledge of the applications. In such cases, T1 requires developers to explain the requirements and their implementation to the reviewers. P-T1 said, "*You have to explain what you have done and why. [...] so that they need not invest so much time to understand what is*

the problem [...] Then they will do a comparative study and they will take some time to go over every line and think whether it is required or not, or can it be done in some other way." Although cooperation between different teams is a healthy attitude, there might be a risk of developers influencing the reviewers by their explanation. P-T13 indicated the possibility of creating a bias when reviewers are walked-through the code rather than looking at it with a fresh set of eyes. He said, "*umm, I have not really thought about [the possibility of influencing the reviewers.] [...] Maybe. Maybe there is a bit.*"

Preliminary code review is done as a checkpoint before the formal review. `(SA)` This is an interesting example of developers collaborating with reviewers. P-T1 mentioned that reviewers sometimes quickly inspect the code prior to the formal review process and in case of a potential issue, they provide the developer with specific testing to do before the code proceeds to the review stage. This saves reviewers time and effort during the formal code review, and it could help focus the formal process on intricate issues, rather than being overwhelmed with simple ones.

Security is not considered during code review. `(SI)` The majority of the security inattentive participants explained that their teams' main focus for code review is assessing code efficiency and style, and verifying how well new features fulfill functional requirements and fit within the rest of the application. In fact, some participants indicated that their teams pay no attention to security during this stage. It is either not the reviewers' responsibility, or is not an overall priority for the team. P-T7 explained that because reviewers are developers, they are not required to focus on security. In addition to not being mandated, our participants explained that most developers in their teams do not have the necessary expertise to comment on security. P-T7 said, "*Probably in the two years that I've been working, I never got feedback [on] the security of my code [...] [Developers] don't pay attention to the security aspect and they can't basically make a comment about the security of your code.*"

Security consideration in code review is minimal. `(SI)` According to developers from the security inattentive group, some of their teams pay little attention to security during code review only by looking for obvious vulnerabilities. Additionally, this may only be performed if the feature is security-sensitive. In either case,

teams do not have a formal method or plan, and reviewers do not necessarily have the expertise to identify vulnerabilities [119]. Our participants explained that reviewers are either assigned or chosen by the developer, based on the reviewer's qualifications and familiarity with the application. However, this can have serious implications, *e.g.*, those who have security expertise will carry the burden of security reviews in addition to their regular development tasks. P-T12 explained that this caused the individuals who had security knowledge to become "*overloaded*". Although our data does not allow us to make such explorations, it is important to investigate the effect of workload on the quality of code reviews, and whether it has an effect on developers' willingness to gain security knowledge. For example, does being the person designated to do security code reviews motivate developers to gain security knowledge? Or would they rather avoid being assigned extra reviewing workload?

VI. Post-Development Testing Stage

Security is a priority during post-development testing. `(SA)` Three participants from the security adopters group mentioned that their project teams have their own testers that evaluate different aspects, including security. The general expectation is that the testers would have some security knowledge. Additionally, P-T12 mentioned that his company hires external security consultants for further security testing of their applications. However, because the testing process by such experts is usually "*more expensive and more thorough,*" (P-T12), they usually postpone this step until just before releasing the application. We identified two distinct motivations for performing security testing at this stage: **Post-development testing is used to discover security vulnerabilities, or for final verification.** `(SA)` Unsurprisingly, the majority of security adopters rely on post-development testing as an additional opportunity to identify and discover security vulnerabilities before their applications are put out to production. T1, on the other hand, expects security post-development testing to reveal zero vulnerabilities. P-T1 explained, "*If they find a security issue, then you will be in trouble. Everybody will be at your back, and you have to fix it as soon as possible.*" Thus, this stage is used as a final verification that security practices in the previous stages were indeed successful in producing a vulnerability-free

application.

Similar to the code review stage, we found evidence of collaboration between the development and the testing team, however, **Testers have the final approval.** `(SA)`. Testers would usually discuss with developers to verify that they understand the requirements properly, since they do not have the same familiarity with the application as developers. However, P-T5 explained that although developers can challenge the testing team's analysis, they cannot dismiss their comments without justification. Addressing security issues is consistently a priority. P-T5 said, "*[The testing team will] talk to the development teams and say, 'here's what we think of this', and the development team will sometimes point out and say, 'oh, you missed this section over here' [...] but one of the things is, we don't let the development teams just say, 'oh, you can't do that because we don't want you to'. So the security teams can do whatever they want.*" Cooperation between developers and testers could help clear ambiguities or misunderstandings. In T5 testers have some privilege over developers; issues raised by testers have to be addressed by developers, either by solving them or justifying why they can be ignored. P-T5 hinted that disagreements may arise between different teams, but did not detail how they are resolved. Further exploration of this subject is needed, taking into consideration the level of security knowledge of the development team compared to the testing team.

Security is prioritized in post-development testing for all of our security adopters, where they rely on an independent team to test the application as a whole. On the other hand, although post-development testing appears to be common to all teams from the security inattentive group (with the exception of T10), it often focuses primarily on functionality, performance and quality analysis, with little to no regard for security. Our analysis revealed the following insights and approaches to post-development security testing.

Security is not considered in post-development testing. `(SI)` According to their developers, two teams (T10, T15) do not consider security during this stage. T10 does not perform any testing, security or otherwise. The company to which T15 belongs has its own Quality Analysis (QA) team, though they do not perform security testing. P-T15 said, "*I've never seen a bug related to security raised by QA.*"

The case of T15 is particularly concerning; many teams rely on this stage to address software security, while T15 does not. According to our data, security is not part of the development lifecycle in T15. It would be interesting to further explore why some teams completely ignore software security, and what factors could encourage them to adopt a security initiative.

Post-development testing plans include a security dimension. `(SI)` As mentioned earlier, P-T2 relies mainly on this stage for security testing, In addition, P-T6, and P-T13 say that their teams consider security during this stage. However, there seems to be a disconnect between developers and testers in T6; developers are unaware of the testing process and consider security testing out of scope. Despite her knowledge that security is included in this stage, P-T6 mentioned, "*I don't remember any tester coming back and telling [me] there are [any] kinds of vulnerability issues.*" T13 started integrating security in their post-development testing after a newly hired tester who decided to approach the application from a different perspective discovered a serious security issue. P-T13 explained, "*No one had really been thinking about looking at the product from security standpoint and so the new tester we had hired, he really went at it from 'how can I really break this thing?' [..] and found quite a few problems with the product that way.*" The starting point of security testing in T13 was a matter of chance. When an actual security issue was discovered in their code, security was brought to the surface and post-development testing started addressing security.

Through our analysis, we found that along the security prioritization spectrum, there are cases where security in this stage is driven by different factors, as explained below.

Some participants discussed that their team relies on a single person to handle security, thus security consideration is driven by specific factors. For example, in T4, **Post-development security testing is feature-driven.** `(SI)`. P-T4 is the only developer in his company with security expertise, thus he is responsible for security. He explained that his company has limited resources and few employees, thus they focus their security testing efforts only on security-sensitive features (*e.g.*, authentication processes), as flagged by the developers. Thus, the question is how reliable

are assessments in this case given that they are done by developers with limited security expertise? On the other hand, in T7, **Post-development security testing is adhoc.** `(SI)`. P-T7 explained that they rely on a single operations-level engineer who maintains the IT infrastructure and handles security testing. Thus, testing is unplanned and could happen whenever the engineer has time or "*whenever he decides.*" P-T7 erroneously [153] presumes their applications are risk-free since they are a "*small company*", and thus they are not an interesting target for cyberattacks. Company size was used by some of our participants to justify their practices in multiple instances. Although in our data we did not find evidence to support that company size affects actual security practices, it shows our participants' perception.

We also found that an external mandate to the company can be a driving factor for security consideration. For example, P-T8 reported that his company needs to comply with certain security standards, thus his team performs security testing when they are expecting an external audit "*to make sure the auditors can't find any issue during the penetration test.*" In this case, **Post-development security testing is externally-driven.** `(SI)` Such external pressure by an overseeing entity was described as "*the main*" driving factor to schedule security testing; P-T8 explained that if it were not for these audits, his team would not have bothered with security tests. Mandating security thus proved to be effective in encouraging security practices in a team that was not proactively considering it.

As evidenced by our data, the security inattentive group's security practices, if existent, are generally informal, unstructured, and not necessarily performed by those qualified. The main focus is delivering features to customers; security is not necessarily a priority unless triggered, *e.g.*, by experiencing a security breach or expecting an external audit.

4.2.2 The adopters vs. the Inattentive

In general, security practices appear to be encouraged in teams to which the security adopters belong. In contrast, as explained by participants from the security inattentive group, their teams' main priority is functionality; security is an afterthought.

Table 4.3: Summary of themes emerging from the security adopters and the security inattentive, and common themes between the two groups. Although common themes exist, driving factors for these themes may differ. See Section 4.2.2 for more details.

Security Adopters Themes	**Common Themes**	**Security Inattentive Themes**
	Design	
· Security design is very important	· Security is not considered in the design stage	· Security consideration in the design stage is adhoc
	Implementation	
	· Security is a priority during implementation · Developers' awareness of security is expected when implementing	· Security is not a priority during implementation · Developers take security for granted · Developers misuse frameworks · Developers lack security knowledge · Developers perceive their security knowledge inaccurately · Vulnerability discovery can motivate security
	Developer Testing	
· Developers test for security fortuitously · Security is a priority during developer testing	· Developers do not test for security	· Developers' security testing is feature-driven
	Code Analysis	
· Security is a priority during code analysis		· Security is a secondary objective during code analysis · Developers rarely perform code analysis, never for security · Developers vary in awareness of analysis tools
	Code Review	
· Code review is a formal process that includes security · Preliminary code review is done as a checkpoint before the formal review		· Security is not considered during code review · Security consideration in code review is minimal
	Post-development Testing	
· Security is a priority during post-development testing · Post-development testing is used to discover security vulnerabilities, or for final verification · Testers have the final approval		· Security is not considered in post-development testing · Post-development testing plans include a security dimension · Post-development security testing is feature-driven · Post-development security testing is adhoc · Post-development security testing is externally-driven

Contrary to a trend towards labelling developers as "the weakest link" [71], our analysis highlights that poor security practices is a rather complex problem that extends beyond the developer. Just as we have identified instances where developers lack security knowledge or lack motivation to address security, we have also identified instances

where security was ignored or dismissed by developers' supervisors, despite the developer's expertise and interest. It is especially concerning when security is dismissed by those high in the company hierarchy. As an extreme case, P-T15 reported zero security practices in their SDLC; she explained "*To be honest, I don't think anybody cares about [security]. I've never heard or seen people talk about security at work [...] I did ask about this to my managers, but they just said 'well, that's how the company is. Security is not something we focus on right now.'*"

It was interesting to find that all our participants who identified themselves as developers of web applications and services, *i.e.*, in their current daily duties, (namely, P-T4, P-T6, P-T7, P-T8, P-T10, P-T13, P-T15) fall in the security inattentive group. Specific reasons for this are unclear. It may be because web-development is generally less mature and has a quick pace [137], and teams are eager to roll-out functionality to beat their competitors. In such cases, functional requirements may be prioritized and security may be viewed as something that can be addressed as an update, essentially gambling that attackers will miss any vulnerabilities in the intervening time. Teams who have not yet become victims may view this as a reasonable strategy, especially since patching generally does not requires end-user involvement (*e.g.*, web server fixes do not require users to update their software), making it a less complicated process. However, since participants building other types of software also fall in the security inattentive group, it is hard to draw a generic conclusion that web-development is particularly insecure.

Table 4.3 summarizes the themes that emerged from our analysis. As expected, we found conflicting themes between the security adopters and the security inattentive group, where the more secure themes consistently belongs to the security adopters. However, our analysis also revealed common themes (see Table 4.3), some of which are promising while others are problematic for security. On the positive side, participants from both groups discussed developers' role in security during implementation. On the other hand, participants from both groups also indicated a lack of attention to security in the design stage. Reasons leading to these common themes sometimes vary. Consider the theme *Developers do not test for security*; the security inattentive group ignored security testing because developers often lack the knowledge necessary

to perform this task. Whereas for the security adopters, the reason is that security testing is not included in developers' tasks even if they have the required knowledge. In Section 4.4.2, we discuss factors that we identified as influential to security practices.

4.3 Software Security Best Practices

After exploring real life security practices, how do these compare to security best practices? To answer this question, we amalgamate best practices into a concise list of the most common recommendations. In Section 4.4, we discuss the relationship between practices found in our study and best practices. Recall, we offered background on popular sources of software security best practices in Section 2.2.

Available resources for security best practices vary in their organization and their presentation style, *e.g.*, they vary in technical details. Practitioners may find difficulty deciding on best practices to follow and establishing processes within their organizations [108, 133, 166]. To help frame security practices we identified, we collected recommendations from the sources discussed in Section 2.2 to compose a concise set of best practices. This resulted in an initial set of 57 unorganized recommendations varying in format and technical details. We then grouped related recommendations, organized them in high-level themes, and iterated this process to finally produce the following 12 best practices. Other amalgamations may be possible, but we found this list helpful to interpret our study results. The list could be of independent interest to complementary research in this area.

B1 **Identify security requirements.** Identify security requirements for your application during the initial planning stages. The security of the application throughout its different stages should be evaluated based on its compliance with security requirements.

B2 **Design for security.** Aim for simple designs because the likelihood of implementation errors increases with design complexity. Architect and design your software to implement security policies and comply with security principles such as: secure defaults, default deny, fail safe, and the principle of least privilege.

B3 **Perform threat modelling.** Use threat modelling to analyze potential threats

to your application. The result of threat modelling should inform security practices in the different SDLC stages, *e.g.*, for creating test plans.

B4 **Perform secure implementation.** Adopt secure coding standards for the programming language you use, *e.g.*, validate input and sanitize data sent to other systems, and avoid using unsafe or deprecated functions.

B5 **Use approved tools and analyze third-party tools' security.** Only use approved tools, APIs, and frameworks or those evaluated for security and effectiveness.

B6 **Include security in testing.** Integrate security testing in functional test plans to reduce redundancy.

B7 **Perform code analysis.** Leverage automated tools such as SATs to detect vulnerabilities like buffer overflows and improper user input validation.

B8 **Perform code review for security.** Include security in code reviews and look for common programming errors that can lead to security vulnerabilities.

B9 **Perform post-development testing.** Identify security issues further by using a combination of methods, *e.g.*, dynamic analysis, penetration testing, or hiring external security reviewers to bring in a new perspective.

B10 **Apply defense in depth.** Build security in all stages of the SDLC, so that if a vulnerability is missed in one stage, there is a chance to eliminate it through practices implemented in the remaining stages.

B11 **Recognize that defense is a shared responsibility.** Address software security as a collective responsibility of all SDLC entities, *e.g.*, developers, testers, and designers.

B12 **Apply security to all applications.** Secure low risk applications and high risk ones. The suggested effort spent on security can be derived from assessing the value of assets and the risks, however, security should not be ignored in even the lowest risk applications.

4.4 Interpretation of Results

In this section, we compare security practices from our study to best practices, present factors influencing those practices, and discuss future research directions. We comment on teams' practices as described by their developers (our participants), recognizing that we have only one perspective per team. Compliance (or lack thereof) to all best practices is not proof of a secure (or insecure) SDLC. However, this list of widely agreed upon best practices allows us to make preliminary deduction on the software security status quo.

4.4.1 Current Practices versus Best Practices

Our analysis showed different approaches to security and varying degrees of compliance with best practices. The best practice with most compliance is B9; almost all participants reported that their team performs security post-development testing (to varying degrees). Contrarily, most do not *apply defense in depth* (B10); the security adopters do not consistently integrate security throughout the SDLC and the security inattentive group relies mainly on specific stages to verify security (*e.g.*, post-development testing). In addition, security is generally not a part of the company culture for the security inattentive group and they commonly delegate a specific person or team to be solely responsible for security. This leads to adhoc processes and violates B11: *recognize that defense is a shared responsibility*. Moreover, the security inattentive group violates B12 by ignoring security in applications considered low-risk without evidence that they performed proper risk analysis.

Deviations from best practices are apparent even from the design stage. The majority of participants indicate that their teams do not address security during design, contradicting B1–B3. Some developers may even deliberately violate the *Design for security* best practice (B2) to achieve their business goals and avoid extra work. On the other hand, the two participants who discussed formal consideration of security in design claim the advantages of having more informed development processes, identifying all relevant threats and vulnerabilities, and not getting distracted by irrelevant ones [145].

The implementation stage is particularly interesting; it shows the contradictions

between the security adopters and the security inattentive. Participants from both groups *perform secure implementation* (B4), yet this only applied to three security inattentive participants. For most of the security inattentive group, *security is not a priority* and *developers take security for granted*, assuming that frameworks will handle security. While frameworks have security benefits [143], each has its own secure usage recommendations (*e.g.*, [1]), often buried in their documentations, and it is unclear if developers follow them. In fact, our study suggests that *developers misuse frameworks* by circumventing correct usage to more easily achieve their functional goals, another violation of B4. Moreover, despite their reliance on frameworks, participants report that security is not factored in their teams' framework choices (violating B5).

We found non-compliance with best practices in other development stages as well. For example, some teams do not include security in their functional testing plans, violating B6, and some teams do not perform code analysis, violating B7. Ignoring code analysis is a missed opportunity for automatic code quality analysis and detection of common programming errors [22]. Participants who said their teams use security code analysis tools, do so to focus subsequent development stages on the more unusual security issues. Others do not review their code for security (violating B8); rather code review is mainly functionality-focused. In some cases, participants said that reviewers do not have the expertise to conduct security reviews, in others they maybe overloaded with tasks, and sometimes code review plans simply do not include security.

4.4.2 Factors Affecting Security Practices

Through close inspection of our results and being immersed in participants' reported experiences, we recognized factors that appear to shape their practices and that may not be adequately considered by best practices. We present each factor and its conflict with best practices, if applicable.

Division of labour. Best practices conflict with some of our teams' division of labour styles. Participants explained that some teams violate the *Apply defense in depth* (B10) best practice because applying security in each SDLC stage conflicts

with their team members' roles and responsibilities. In some teams, developers are responsible for the functional aspect (*i.e.*, implementation and functional testing) and testers handle security testing. These teams are also violating B6, because integrating security in functional testing plans would conflict with the developers' assigned tasks. Complying with these best practices likely means they need to change the team's structure and re-distribute the assigned responsibilities. Teams may be reluctant to make such changes [133] that may conflict with their software development methodologies [99], especially since security is not their primary objective [71].

Security knowledge. We found that the expectation of security knowledge (or lack thereof) directly affects the degree of security integration in developers' tasks. When security knowledge was expected, participants said that developers were assigned security tasks (*e.g.*, performing security testing). On the other hand, we found that developers' (expected) lack of security knowledge resulted in lax security practices (*Security is not considered in the design stage*, *Security is not a priority during implementation*, *Developers do not test for security*, and *Security is not considered during code review*). While these violate best practices (*e.g.*, B1, B4 B6, B8), it is unrealistic to rely on developers to perform security tasks while lacking the expertise. From teams' perspective, they are relieving developers from the security burden. This may be a reasonable approach, loosely following recommendations of taking the developer out of the security loop when possible [13, 71]. Another obvious, yet complicated, answer would be to educate developers [104]. However, companies may lack the resources to offer security training, and there is evidence that developers remain focused mainly on their primary functional task and not security [119].

Company culture. Another influential factor indicated by participants is the teams' cognizance of security and whether it is part of the company culture. In teams where security was reportedly advocated, developers spoke of security as a shared responsibility (conforming with B11). In instances where security was dismissed, participants said that developers did not consider security, and even those with security knowledge were reluctant to apply it. For successful adoption of security, initiatives should emerge from upper management and security should be rooted in the company's policies and culture. Developers are more likely to follow security practices if

mandated by their company and its policies [179]. Integrating and rewarding security in the company culture can significantly motivate security practices [178, 179], compared to instances where security is being viewed as something that only “heroes” do if there is time.

Resource availability. Some participants said their team decides their security practices based on the available budget and/or employees who can perform security tasks. As reported, some teams violate B10 as they do not have enough employees who can perform all the recommended security tasks in addition to their original workload. Also, others reportedly violate B9, because they neither have the budget to hire external penetration testers, nor do their members have the expertise to perform such post-development tests. For such companies, the price for conforming with these best practice is too steep for little perceived gain. In other cases, participants said their team strains their resources in ways that can be detrimental. For example, the one developer with the most security knowledge is handed responsibility to identify security-sensitive features and to verify the security of the team’s code. This is a significant burden, yet with little support or guidance. Besides the obvious security risks of such an approach, it may also lead to employee fatigue and ultimately to the loss of valuable team members.

External pressure. Monitoring by an overseeing entity can drive teams to adopt security practices to ensure they comply with its standards. Encouraging security practices through external mandates is not new, *e.g.*, the UK government mandated that applications for the central government should be tested using the National Technical Authority for Information Assurance CHECK scheme [9]. As a result of this initiative, companies have improved their management and response to cyber threats [8]. It would be interesting to explore how to mandate security practices in companies, and how governments and not-for-profit agencies could support teams, particularly those from the security inattentive group, to become more secure.

Experiencing a security incident. Participants reported that discovering a

vulnerability or experiencing a security breach first-hand is another factor that encouraged security practices and awareness in their teams. Despite extensive publicity around security vulnerabilities, awareness of and commitment to security remains low [139]. Our analysis shows that direct vulnerability discovery influenced security practices more than hearing news-coverage of high-profile vulnerabilities (*e.g.*, [41, 160]). This can be explained by the optimistic bias [171]: the belief that "misfortune will not strike me" [139]. Rhee *et al.* [139] found that the optimistic bias strongly influences perception of security risks in Information Technology (IT). It is even greater when the misfortune seems distant, without a close comparison target. Thus, to overcome such bias, security training and awareness has to reach all levels–from upper management to those directly involved in the development process. Similar to Harbach and Smith's [75] personalized privacy warnings which led users to make more privacy-aware decisions, software security training should be personalized and provide concrete examples of the consequences of these threats to the company. We recommend that training should also not focus exclusively on threats; it should provide concrete proactive steps with expected outcomes. Additionally, it should include case studies and first-hand accounts of security incidents, and approaches to overcome them. Hence, security training moves from the theoretical world to the real world, aiding in avoiding the optimism bias.

4.4.3 Future Research Directions

Security best practices advocate for integrating security starting from the early SDLC stages. However, with limited resources and expertise, if a team can only address security in post-development testing, is this team insecure? Or might this testing be sufficient? Is the security inattentive group in our dataset really guilty of being insecure? Or did they just find the cost of following security best practices too steep? Our work highlights that best practices are lacking in terms of guiding development teams to choose which best practices to follow based on their limited resources and expertise.

For future research, we suggest devising a lightweight version of security best practices and evaluating its benefit for teams that do not have enough resources to

implement security throughout the SDLC, or when implementing traditional security practices would be too disruptive to their workflow. Additionally, teams that succeeded at building a security-oriented culture should be further explored to better understand how others can adopt their approach. Further exploration of how to incorporate security in the company culture and evaluating its benefits can be a starting point for more coherent security processes, since developers are more likely to follow security practices if mandated by their company and its policy [179]. Particularly, what lessons can be carried from the security adopters over to the security inattentive group? Our work explores some of the issues surrounding secure development practices. Surveys with a larger sample of companies and more stakeholders would be an interesting next step.

4.5 Conclusion

Through interviews with developers, we investigated SDLC practices relating to software security. Our analysis showed that real-life security practices differ markedly from best practices identified in the literature. Best practices are often ignored, simply since compliance would increase the burden on the team; in their view, teams are making a reasonable cost-benefit trade-off. Rather than blaming developers, our analysis shows that the problem extends up in company hierarchies. Our results highlight the need for new, lightweight best practices that take into account the realities and pressures of development. This may include additional automation or rethinking of secure programming practices to ease the burden on humans without sacrificing security.

Chapter 5

Security Knowledge and Motivation

In the previous chapter, we focused on teams' software security practices, how these compare to best practices in the literature, and factors that may influence security processes in real life. Thus, our analysis focused on quotations related to practices in the different stages of the development. These quotations represent only a subset of the interviews.

Through a second phase of analysis that includes the entirety of the interview script, we draw our attention more to the human in the development loop—the developer. In our interviews we initially set out to explore developers' knowledge of software security and how they acquire this knowledge, hence $RQ1$ below. However, our data analysis highlighted that even those with the necessary knowledge may lack motivation towards software security. This motivated us to explore a second research question, $RQ2$. The two research questions for this chapter are:

RQ1 How do developers acquire knowledge relating to software security?

RQ2 What are developers' motivations towards software security?

Our data analysis revealed that knowledge and motivation are two intertwined aspects that may influence security practices; motivation in itself is not enough if the developer lacks security knowledge and, as it turns out, knowledge itself affects motivation. In this chapter, by addressing our two research questions, we identify opportunities and strategies for acquiring security knowledge (Section 5.2), explore factors influencing developers' motivation towards security (Section 5.3), as well as the relation between motivation and security knowledge (Section 5.4).

Part of this chapter is published at SOUPS Workshop on Security Information Workers (WSIW). USENIX Association, 2018. [19].

5.1 Using Grounded Theory for Analysis

We used Strauss and Corbin's Grounded Theory methodology [159] to analyze our interviews. Following Grounded Theory, we did not start with a preconceived theory; rather we worked from the data to offer insights and enhance understanding of the phenomenon under study.

Strauss and Corbin's methodology involves three types of *coding* processes. Coding refers to abstracting and conceptualizing raw data [43]. *Open coding* is the first coding step in Grounded Theory, in which data is analyzed and textual passages are assigned descriptive codes. Next, through *axial coding*, the researcher looks for relationships and connections between open codes. This is followed by *selective coding* to integrate categories and refine the theory. During selective coding, the researcher identifies a central idea (*core category*) to which all the other categories relate. The core category could either evolve out of the existing list of categories, or the researcher may need to create a more abstract phrase that can include all the existing categories [43].

To analyze our qualitative data, we performed open-coding through examining the answer to each question in the interview script and assigning codes describing the main themes or ideas discussed. The main researcher performed the open-coding, however, codes were discussed with a second researcher whenever a new code was created. Open coding was done using Atlas.ti on 600 unique excerpts and resulted in a total of 170 open codes. We italicize and use a different *font* for our codes when reporting.

"*Learning from peers*" is an example of an open code that we created when participants indicated that they acquire security knowledge through interaction with their colleagues. In the following quote, P-T11[1] explains how all his security knowledge came from colleagues in the company where he works. He said, "*I guess up until now, any knowledge I have got of it has just been purely from peers, or anytime we bring in a new employee and they have more knowledge about it. That's where I kinda learn*

[1] In this chapter, we refer to participants with respect to their teams as in the previous chapter. However, participants who discussed multiple teams are referred to as P-Ti/Tj, where i is their current team, and j is their previous team.

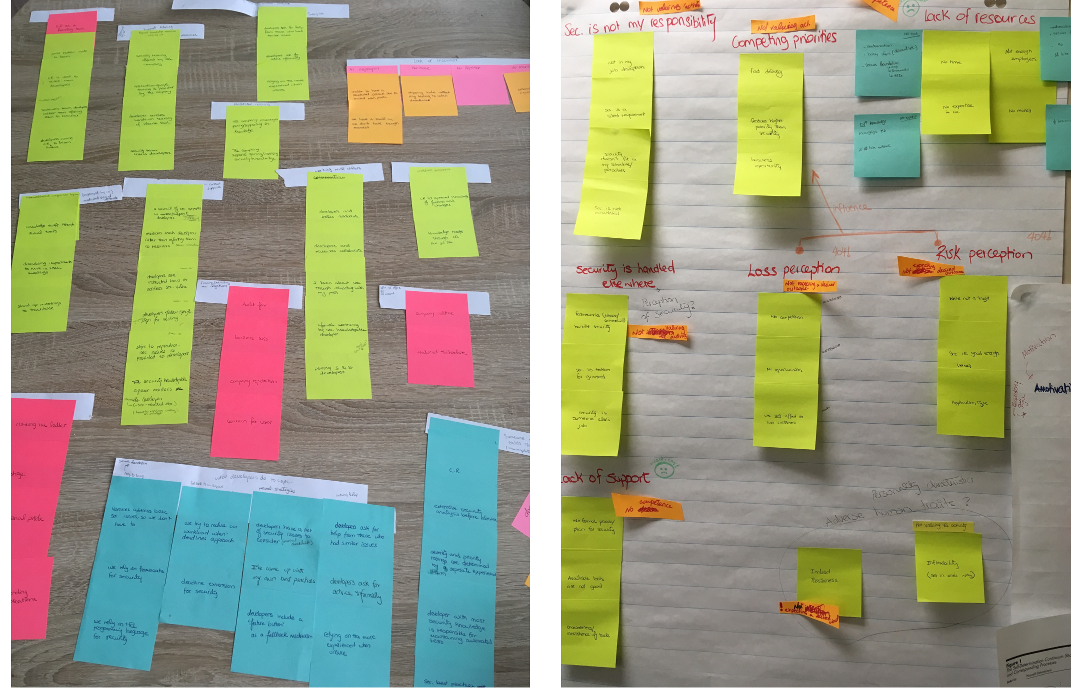


Figure 5.1: Axial coding process. Left: first round of axial coding, right: looking for relationships and connections

it from. But I have never really taken any courses or anything else in the security area."

Following open coding, we performed axial coding by looking for patterns, relationships, and connections between the open codes. During this process, we asked questions, such as, *why, where, how*, and *when*. We wrote each of the codes on a Post-It note, grouped similar ones, and looked for relationships, such as categorical or causality relationships. Even though this process was possible using Atlas.ti, we preferred using Post-It notes to allow us to be more immersed in the data, have an overview of the codes and categories, and have the ability to move them around as needed, as shown in Figure 5.1.

The last step in coding was selective coding, where we worked towards integrating and refining the categories, and identifying a *core category* that represents the overall theme of the research [159]. To achieve this, we examined the categories, while referring back to the interview scripts (raw data), abstracting the main issue and asking ourselves; "*what comes through although it might not be said directly?*" [159].

Through being immersed in the data and as the analysis continued, a core category relating to the concept of internalizing and accepting security activities and behaviours began to emerge. We will discuss this further in Section 5.4.

5.1.1 Researcher Bias

The researcher mainly performing the analysis has background in software engineering and software development, which may have influenced the analysis. However, the analysis did not start with any pre-conceived theories or ideas and the researcher tried to maintain objectivity throughout the analysis.

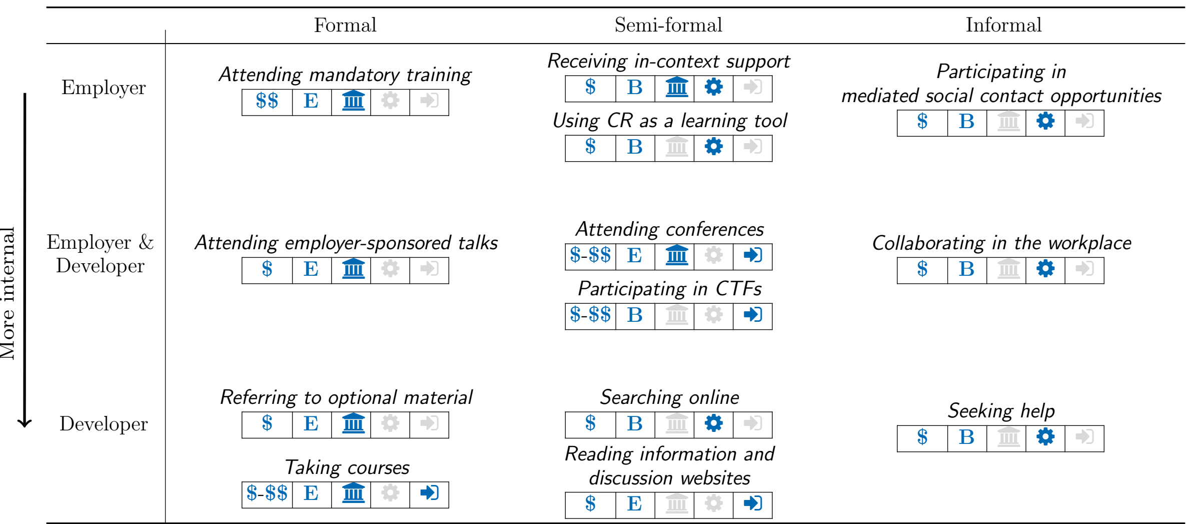
5.2 Knowledge Acquisition Taxonomy

Software security is not a trivial issue due to the adversarial nature of security, where software development teams and attackers are in a constant arms race. Additionally, the balance is tipped in the adversary's favour; to succeed, attackers may only need a single vulnerability to compromise a system, whereas for guaranteed security, software teams should eliminate *all* vulnerabilities. Failure on the part of developers could lead to catastrophic consequences [71] (*e.g.*, [41, 160]), and with evolving threats, it is important to stay updated on software security issues.

Through our analysis of interview data, we identified different opportunities for acquiring and sharing security knowledge. Some of these opportunities were not explicitly reported by our participants as learning methods, though our analysis revealed their potential for knowledge acquisition.

Types of Learning. We found that the learning opportunities identified in our data can be classified as "formal","semi-formal", and "informal". Formal learning is always organized and structured, has learning objectives, and is always intentional from the learner's perspective [48, 56, 57, 120]. On the other end of the spectrum, informal learning is never organized nor structured, does not have specific objectives, and is never intentional from the learner's perspective [48, 57, 120]. In other words, informal learning occurs through experience in everyday life. Semi-formal learning

Table 5.1: Knowledge Acquisition Taxonomy. The taxonomy presents knowledge acquisition opportunities and features associated with each opportunity. See inline for their description.



More internal ↓	Formal	Semi-formal	Informal
Employer	*Attending mandatory training* ($$, E)	*Receiving in-context support* ($, B); *Using CR as a learning tool* ($, B)	*Participating in mediated social contact opportunities* ($, B)
Employer & Developer	*Attending employer-sponsored talks* ($, E)	*Attending conferences* ($-$$, E); *Participating in CTFs* ($-$$, B)	*Collaborating in the workplace* ($, B)
Developer	*Referring to optional material* ($, E); *Taking courses* ($-$$, E)	*Searching online* ($, B); *Reading information and discussion websites* ($, E)	*Seeking help* ($, B)

falls between these two. Several sources [48, 57, 120] refer to it as "non-formal" learning, while others use this term to refer to "informal" learning (*e.g.*, [56]). We use the term "semi-formal" to avoid confusion. The definition for "semi-formal" learning has the least consensus [120], however, adapted from [48, 57, 120], we describe semi-formal learning as lacking one or more aspects of formal learning while being more organized and structured than informal learning.

Activity Initiator. The initiator of the activity is the entity with the motivation to start the activity, thus the learning opportunity. This can be the employer, *e.g.*, mandatory activities that the developer attends for compliance. On the other hand, developers themselves can initiate activities, *i.e.*, activities that the developer is self-motivated to initiate without direct encouragement or mandate. Some activities are initiated by *both* the employer and the developer, *e.g.*, non-mandatory activities set up by the employer that developers can choose to participate in, even though they would not have initiated the activity on their own. Thus, activity initiation is along a *employer - developer* spectrum, where initiation is more internal to the developer as we move towards the developer end of the spectrum.

In Table 5.1, we present a taxonomy of activities described by our participants

that we have identified as methods for knowledge acquisition. We represent the type of learning associated with the activity horizontally across the table, and the initiator of the activity (thus the initiator of the learning opportunity) vertically. Note that, unmotivated developers may still perform activities in the taxonomy's third row; they may benefit from learning opportunities that are a by-product of their tasks, however, our analysis shows that they are likely to procrastinate with respect to these activities.

Features. We have identified five different features for each learning opportunity/activity.

- *The relative cost (**$**, **$$**).* The symbol **$** indicates that the activity is relatively low cost and **$$** indicates higher cost. Obviously, the cost one company finds reasonable may be expensive for another. Thus, this characteristic should be used to compare activities relative to each other, rather than, *e.g.*, finding the most reasonably priced activity.

- *Fit in the developer's objective (**E**,**B**).* This feature describes how learning fits in the developer's objective. **E** indicates that learning is the developer's explicit objective of the activity, whereas **B** indicates that learning is a by-product.

- *Source expertise (🏛).* This describes whether the source of knowledge has high security expertise. A 🏛 in the taxonomy indicates that the source of information has high experience in the aspect being taught. In cases where it is greyed-out, the source of information varies in their level of expertise. Note that advancement in knowledge can occur even if the teacher does not have higher expertise than the learner through discussions and sharing of interpretations [154].

- *Fit in the SDLC (⚙).* This indicates whether the activity is part of the developer's tasks, thus part of the SDLC. A ⚙ in the taxonomy indicates that the activity is performed as part of the developer's tasks, whereas it is greyed-out when the activity is *not* part of the developer's tasks.

- *Knowledge source (➜]).* This feature describes the source of knowledge with respect to the company. A ➜] in the taxonomy indicates that knowledge is

Table 5.2: Distribution of participants mentioning learning opportunities fitting in each cell of the Knowledge Acquisition Taxonomy

More internal ↓	Formal	Semi-formal	Informal
Employer	*Attending mandatory training* P-T1, P-T2, P-T5	*Receiving in-context support* P-T1, P-T4, P-T6, P-T7/T8, P-T10, P-T11, P-T12 *Using CR as a learning tool* P-T1, P-T4, P-T6, P-T10	*Participating in mediated social contact opportunities* P-T2, P-T6, P-T7/T8, P-T10, P-T11, P-T12, P-T13
Employer & Developer	*Attending employer-sponsored talks* P-T2, P-T3	*Attending conferences* P-T2, P-T3, P-T5, P-T10, P-T11 Participating in CTFs —	*Collaborating in the workplace* P-T1, P-T2, P-T3, P-T5, P-T6, P-T7/T8, P-T10/T11, P-T10, P-T11
Developer	*Referring to optional material* P-T1, P-T3, P-T10 *Taking courses* P-T4, P-T6, P-T11	*Searching online* P-T3, P-T9, P-T10/T11, P-T10 *Reading information and discussion websites* P-T4, P-T7/T8, P-T10, P-T12	*Seeking help* P-T1, P-T2, P-T4, P-T7/T8, P-T9, P-T10, P-T11, P-T12

flowing to the company from external sources. When it is greyed-out, this implies that knowledge is shared within the company.

This taxonomy was built based on our analysis of interview data. Though other activities may exist that are not included in the taxonomy, the taxonomy allows for exploring and reasoning about learning-inducing activities in the context of software security.

We will now describe each of the activities presented in our taxonomy. Table 5.2 shows how the knowledge acquisition taxonomy categorizes the learning opportunities described by participants.

5.2.1 Formal Learning

For all formal learning opportunities that we have identified, learning is explicit (**E**) from the developer's perspective. In addition, the source of information, be it an instructor in a training session or an author of a training manual, has high expertise in the topic being taught (🏛). We will now discuss each of the identified formal learning opportunities.

Attending Mandatory Training ($$ E 🏛)

This formal training is usually expected as the first step for a secure SDLC. Most of our participants mentioned that they attended mandatory training as they started their job. In most cases, the training focuses on general security topics (*e.g.*, passwords, and phishing), and best practices while using company resources or sharing company code. However, some participants also reported that their training included aspects of software security (P-T1, P-T2, P-T5). In addition, P-T1 mentioned that her company requires that developers pass certain exams testing their knowledge of the training topics.

In addition to mandatory foundational training, P-T5 explained that his company also mandates that developers attend regularly scheduled training sessions. However, such mandatory training is often unwelcomed by developers. P-T5 explains, "*It's so often actually that people start to get annoyed at having to go to it*". Thus, the frequency of the sessions was reduced as "*the morale improved*" and the security company culture improved. P-T5 explains, "*We kept doing [regular training], and it's been quite effective. So as a result of it being effective, we scaled down the frequency at which it needs to be done. But it's not because it's less important, it's just because people started to get it more.*"

It appears that adapting the frequency of the training to the outcomes could lead developers to perceive the usefulness of the training and be more attentive to it. The rationale could be that the more attentive developers are, the less time they have to spend on additional training and the faster they can get go back to their development work. In Section 5.4, we discuss how valuing security can have an impact on developers' performance and on promoting software security.

Attending Employer-Sponsored Talks ($ E 🏛) *and* Referring to optional material ($ E 🏛)

These two explicit learning activities are initiated by *both* the company (for hosting the talk or providing the material), and the developer (for deciding to take this learning opportunity). The source of information for both activities are considered experts in the areas they discuss (🏛).

- *Attending employer-sponsored talks.* Two participants (P-T2, P-T3) mentioned that their companies sometimes arrange technical talks, *e.g.*, to introduce a new security API. We classified this activity under formal learning because it is structured, organized, and addresses a specific topic relevant to the company with specific learning objectives. These talks are usually given by employees within the company, thus knowledge sources are internal to the company. However, P-T3 mentioned that his company sometimes invites external experts to give talks.

- *Referring to optional material.* Three participants also mentioned that their companies provide optional security knowledge resources. Experts in the company prepared these resources in the form of reading material, or video lectures. P-T3 said, "*they do have an infrastructure, so that people can easily find actual courses taught by colleagues at this very institution. [...] They have a series of sort of self seminars, so these are like slideshows or online videos that we can look at.*" P-T1 mentioned that their material is accompanied by an assessment test, that allows the developer to self-test their knowledge. Additionally, these assessments allow the company to recognize the developer's level of security knowledge.

Taking Courses ($-$$ E 🏛 ➜)

To increase or maintain their security knowledge, some participants explained that they do so by taking courses or even acquiring a graduate degree. Their companies did not require or even encourage them to do so, thus the initiation of this learning opportunity is internal to the developer. P-T11 explained, "*For me, I like taking courses [...] I mean if something shows me all the types of vulnerabilities I need to really be thinking about when I am working on my applications.*" The cost of this activity varies, an online course is less expensive than an on-site course, and both are less expensive than a graduate degree. Some companies may reimburse the developer for the tuition fees. Knowledge in this case is flowing to the company from an external source (the institute offering the course or graduate degree).

5.2.2 Semi-Formal Learning

Activities listed under semi-formal learning are not as structured or organized as those listed under formal learning. The activity may (or may not) have specific learning goals. The source of knowledge does not necessarily have high security expertise, and learning from these activities may be the developer's explicit goal or a by-product of the activity.

Receiving In-Context Support ($ B 🏛 ✿) *and* Using CR as a learning tool ($ B ✿)

For both these activities, developers gain security knowledge in situ while working on their tasks (✿). In this case learning does not compete for time with development work, as learning from these activities is a by-product (**B**) of the development task.

- *Receiving in-context support.* In-context support comes in different forms. For example, through giving the developer specific steps to follow to reproduce an issue found in her code, along with an explanation of the issue and how to fix it. In another case, in-context support was done through pairing the developer with a more senior colleague to disseminate security knowledge across the development team and to support junior developers. P-T11 explained, "*[Pairing junior and senior developers intended] to make sure that the junior [developer] doesn't feel, you know, like they are left alone on the issue or they are frustrated or stuck, and have somebody to kinda guide them through the work that they are doing step by step.*" On the other hand, rather than relying on individuals with security expertise, P-T10 explained that his company formed a "security council" from experts within the office to provide security guidance to developers and to ensure developers' questions are answered by the council member with the most relevant expertise. The council informs developers of relevant issues to consider during implementation. Developers are expected to consult this council, *e.g.*, when they need advice. P-T10 explained, "*If there is anything that we flag up as 'ok this might have security implications', then it goes to them to say 'ok, do you guys find anything? [Do] you have any comments on the design? Is*

there anything maybe we didn't think of?'"

- *Using CR as a learning tool.* Code review, being part of the SDLC, is a suitable and convenient opportunity for developers to learn about security in context. P-T1 explained that upon finding a security issue, reviewers take this opportunity to teach the developer about its implications and how to fix it. This can happen face-to-face or through code review feedback. She said, "*[Reviewers] just come right away to your cubicle and explain [to] you [...] because they feel [that] going and taking a book and reading it would be, mmm, so much painful. So, they just come over to you and draw on the board and explain what you did and what you should not do.*" In some cases, the reviewer is not necessarily more experienced than the developer, however, the discussion that arrises during the review session can lead to better insights on code security. P-T1 also mentioned that junior developers can act as mock reviewers to learn about the process and types of issues to avoid in their code.

  Contrary to what would seem obvious on first sight, this learning method is categorized as **B** and not an **E**. Learning about security is not the developer's explicit objective from code reviews; developers either want to learn about the process of code reviews or they want to learn how to fix issues in their code.

Attending Conferences ($-$$ E 🏛 ➜) *and* Participating in CTFs ($ B ➜)

These activities are not mandatory; they are encouraged and sometimes sponsored by the employer. However, it is up to the developer to participate. Knowledge sources for these activities is usually external to the company (➜).

- *Attending conferences.* Three of our participants (P-T2, P-T5, P-T11) mentioned that they sometimes attend academic conferences to keep up with new technologies and new security attacks and defences. There is no mandate from their employers to attend such events, however, it is encouraged. Employers may even reimburse their developers for conference registration fees and/or other expenses. P-T11 explained, "*they do offer umm, they will pay for us to go. Like*

if you want go to a conference that's, you know, in town, they'll pay for the fee to go to the conference."

Thus, attending conferences is an activity initiated by developers and encouraged by employers, where learning is an explicit goal (**E**). Conference presenters are considered experts in the area they are presenting (🏛). The cost of this activity varies depending on the conference itself (conferences differ in their registration fees), and whether the expenses include travel and accommodation expenses.

- *Participating in CTFs.* Capture The Flag (CTF) [45] events are competitions where teams work together to solve challenges. These events offer hands-on experience to developers in an entertaining manner [122]. Developers participate in CTFs to socialize or to win prizes and bragging rights [122,161], thus learning about security is considered a by-product (**B**). However, CTFs are not part of the SDLC, hence the greyed-out ⚙. This is the only activity in our taxonomy where learning is a by-product while not being part of the developer's tasks.

  CTFs are fairly related to conferences [4] and have been increasing in popularity [161], however, none of our participants mentioned them. This implies a missed opportunity for learning in a manner that is shown to be amiable to developers. In addition to external events, employers can organize internal CTFs where teams would be formed from within the company. In such a way, employers can focus their events on the most relevant security aspects. Also, by forming teams from security experts and novices, this can improve collaboration and bridge the gap between developers and security experts [165].

Searching Online ($ B ➜ ⚙) ***and*** **Reading Information and Discussion Websites ($ E ➜)**

These two activities represent accessing online resources in general. Online resources vary in their organization and credibility. They range from personal blogs, knowledge markets (*e.g.*, Stack Overflow [7]), to more official resources (*e.g.*, National Vulnerability Database (NVD) [116] and Common Vulnerabilities and Exposures (CVE) [3]).

We present them as two different activities, as we have identified two different motives to accessing these resources, making them two distinct activities.

- *Searching online.* Developers often use online searches, *e.g.*, to find out how to implement a feature or how to fix an issue in their code [11,12]. If a developer is using online resources to fix a security issue, she may learn about that security issue while working on her task. Thus, learning from this activity is considered part of the SDLC (✿) and a byproduct of the activity (**B**). Two participants (P-T3 and P-T9) briefly discussed this activity. P-T3 explained, "*frankly, you know, Google search engine. I basically search things online. So, when there's something particular that I need to look into, I basically look it up online and see what the internet says.*" He also explained that he would prefer using "*official resources or more reputable sources*", rather than a blog or the such.

- *Reading information and discussion websites.* Contrary to searching online, learning from internet resources in this activity is explicit (**E**) and not part of the SDLC. Some developers explained that they use internet resources to stay up to date on security vulnerabilities. For example, P-T10 explained his strategy, "*[I follow a] couple of blogs, just general websites as well that might point out some new vulnerability. If I want to go in depth on something, then, you know, we can read about it CVEs for that thing.*" Our participants' recounts of their use of discussion websites indicates that they did not actively participate in discussions, rather they were passive learners reading about the security topic and the available discussion.

5.2.3 Informal Learning

Informal learning is sometimes referred to as "learning by experience" [120]. For all the activities listed here, learning is a by-product (**B**) of the activities, which are performed as part of the SDLC (✿).

Participating in Mediated Social Contact Opportunities ($ B ⚙)

Mediated social contact opportunities is a term that we use to describe social opportunities arranged by the employer. For example, some participants mentioned that they discuss work impediments during team meetings, including security issues they face and how they could be addressed. Others mentioned that they work in open-plan offices which often stimulates discussions. P-T10 said, "*We all sit relatively close together, so if someone finds something, they might just sort of say 'OK, does anyone know about this?' 'Why are we doing it this way?'*" P-T10 explained that they value these general discussions as they allow developers to stay informed about security vulnerabilities and prevent them in their code.

Collaborating in The Workplace ($ B ⚙)

In this activity, we focus on collaboration between different members of the project team; more specifically we focus on the interaction and the back-and-forth discussions between teams. In our interviews, participants described multiple instances of different teams working together, *e.g.*, testers working together with developers to better understand the purpose of the code, and thus being able to better analyze potential vulnerabilities. P-T2 explained, "*Usually, if [the testers] think there's a problem, they really wouldn't go ahead and publish the bug like this; they would work with us. They'd be like, 'do I understand this correctly? Is this the correct behaviour?" [...] So, it's a process before the bug actually gets submitted.*" Collaborating with other teams allows for "*information sharing*" between the different teams, as P-T11 described. P-T5 explained that one of the factors to rate the success of a code review session is by determining whether it resulted in information sharing among reviewers and developers. He explained, "*A good review is one where the development team gets a better understanding of the security of the application, and the security team gets a better understanding of how applications are constructed and how to interact with the development teams.*"

Workplace collaboration can help bridge the gap [165] and reduce conflicts between different teams. P-T9 explained that his development team's frustration with the testing team is mainly due to poor communication and the disconnect between the

two teams. He said, "*[The relationship between the testing and development team is] bad. [chuckles] I mean, usually, you just pass them the code and then, they run through test cases and, you know, if they fail, they'll just come back say 'fail'. And then they don't..., because people who [are] doing the testing, they have no, zero knowledge about the code itself.*"

Seeking Help ($ B ✿)

Participants also described multiple instances where they turned to their colleagues for help. This is different from "collaboration in the workplace", as help seeking here is informal and random, occurring only when the developer needs help while working on their tasks, rather than, *e.g.*, during a code review session or a follow-up on testing results. In addition, the developer seeking help is usually the main beneficiary of the knowledge shared. P-T4 explained, "*if I need advice from someone, I would usually ask, you know; 'I'm looking at this thing here, how would you go with doing it?' We kind of just talk back and forth, it's usually pretty free form and open.*" Our participants also mentioned that they sometimes seek help to answer more specific questions relating to specific security issues. For example, P-T1 explained that if she cannot fix a security issue in her code, she asks a teammate who faced a similar issue how they fixed it. Although this activity is initiated by the developer, it is sometimes performed by unmotivated developers albeit after procrastinating. For example, P-T2 mentioned, "*In my experience they would delay [asking for help]. They'd work on things for months and then over coffee they'd be telling me what they're looking at and I'd break it in 2 minutes [...]. And they would be like [chuckle] 'okay, let's do this again'.*"

5.2.4 Insights Based on the Taxonomy

As shown in Table 5.2, formal learning opportunities are the least commonly reported by our participants. It is unclear if our participants' companies are deliberately moving away from formal learning in favour of other types of learning. However, it is clear from Table 5.2 that security learning opportunities that result as a by-product from the developer's tasks and responsibilities are more common than others. We

will refer to those learning opportunities as *task-induced learning opportunities*.

The success of all the task-induced learning opportunities (except '*searching online*') requires interaction and collaboration within the project team. To explore the dialogues, interaction, and collaboration within a project team, we look at the project team through the lens of the third generation activity theory (*cf.* Section 2.7.1). Recall, a project team can consist of multiple teams (*e.g.*, development team, reviewers, and testing team), each of which constitutes an activity system. Thus, a project team can be seen as a network of multiple interacting activity systems [54]. Each team looks at the software from a different perspective and has background, points of views, and objectives different from the other teams (*multi-voicedness* [54]). For example, a development team would focus mainly on functionality, whereas the testing teams might focus on security.

Taking the interacting activity systems as our unit of analysis, we found that some project teams attempt to benefit from their multi-voicedness through communication, negotiations and avoiding conflict. This allows the activity system to grow [54]; the two teams bring together and harmonize their perspectives and objectives, functioning software for the development team and secure software for the security testing team. Breakdowns in this relationship can be detrimental. For example, P-T9 described the relationship between the developers and the testing team as "*bad*" due to the lack of collaboration and the lack of testers' understanding of the software functionality. On the other hand, P-T5 explained that their security testers "*have full access to the development team, so they can coordinate as much as they want*". This allows testers to have a good understanding of the features they are testing and minimizes conflicts between testers and developers. P-T2 gave another example for collaboration between developers and testers, he said, "*while testing they come back to [the developer] with questions and clarifications and then they go complete testing.*" He also gave an example of how the lack of collaboration almost caused a serious security issue to go unnoticed. He explained, "*a guy in [location] [...] was testing some memory issue in the kernel. While he was doing the test, he wanted to access kernel memory from the user process. So, his test actually succeeds to get to the kernel memory. He's [thinking] 'if I can get to the kernel memory, everything is*

fine, continue on.' So I look at the test and I'm like 'dude, [...] the test actually did discover a flaw, but you didn't tell me about it. This is a false negative.' [...] So, this is because of the lack of understanding."

The informal learning opportunities, identified from our interviews, are examples of how project teams benefiting from their multi-voicedness. Some participants indicated that these opportunities allow the developers to learn more about security while the testing team learns more about the functionality of the software. This in turn helped reduce conflicts between the different teams. Participants' description of such activities emphasized their sense of belonging to the team and portrayed their project teams as intertwined teams working together as one; a coherent network of activity systems, rather than disconnected systems.

This is particularly important for software security since the threats are constantly evolving and it is important to stay updated on new security issues [77, 135, 153]. Given that security is not the developer's primary objective, as evidenced by our data and previous work [13, 71, 178], it is unrealistic to expect that developers will be able to remain current on such issues on top of their regular tasks. In addition, security information is often presented in a manner that is unusable to developers [110, 111]. Thus, collaborating with those having high security expertise gives developers a chance to stay updated on security issues, and it could also lead to improving performance and motivation towards software security, as will be discussed later in this chapter.

Although we have some evidence that employers may be supporting some task-induced learning opportunities, our data does not contain enough in-depth evidence for all activities. Our participants did not necessarily describe these activities as security-learning methods, rather they were generally described as part of the developer's typical everyday work activities. It is also unclear whether employers recognize the potential for these activities in spreading security knowledge, and whether such task-induced learning opportunities receive appropriate support. Many developers prefer to learn while working [184], however, these may be missed security knowledge opportunities. Our analysis shows that participants remembered security issues they encountered in live systems more than others they may have learned about from

other sources. Thus, we believe that focusing on and promoting task-induced learning opportunities can be more beneficial for security than the on-size-fits-all mandatory training recommended by many security initiatives. Developers have different learning styles, like all humans [48], thus opportunities such as 'in-context support' provide a more personalized learning experience. Additionally, since developers are better at recognizing security vulnerabilities when primed [119], such in-situ learning opportunities—occurring at critical times when developers need help—are likely to be more effective, especially for security [165]. Employers should also encourage and support other activities that require collaboration within and between teams. This allows for spreading knowledge, reduces conflicts, and helps produce a software product where different points of view and priorities have been considered. For example, employers should find ways to help developers find the most knowledgable person to answer their security questions. As mentioned earlier, when trying to fix a vulnerability, developers may seek help from a colleague who had a similar issue. It may be the case that the developer does not know who that person is or she may postpone seeking help and procrastinate. In addition, the knowledgeable person may be overloaded with such questions on top of their regular tasks. Thus, employers may consider designing a security council, such as the one described by P-T10, or developing an accessible (developer-friendly) system that contains a database of previously detected vulnerabilities, how they were fixed (including code snippets), and who fixed them as potential contact persons. The system could also balance the number of help requests across different contact persons, to avoid overloading some of them. In addition, the time these contact persons spend collaborating with others should count towards their working hours and should not be an extra workload on top of their existing tasks.

5.2.5 Additional Use for the Knowledge Acquisition Taxonomy

Although we set out to explore how developers gain security knowledge, this taxonomy could be used to help induce security-learning opportunities, which not only affects the ability to address security, but as detailed in the following sections, it could also affect the motivation and willingness to do so.

This taxonomy can help employers and teams recognize existing activities adopted by their developers that may lead to advancing their security knowledge. In addition, it could help employers explore learning opportunities and decide on the best methods to promote security knowledge within their organization. Employers could also map their activities onto the taxonomy to determine whether they are a good fit for them and their developers, taking into consideration the different features for each activity and developers' initiative.

Based on our analysis, we have identified three main aspects to consider when deciding on a security knowledge promoting activity:

- The learner's initiative,
- The teacher's experience, and
- The available budget.

Initiative: If the developer (being the learner) is motivated to learn about security, then all the activities listed in the taxonomy are suitable. However, in case of an amotivated developer, it is unlikely that they would initiate an explicit learning activity. Thus, the employer could instead initiate (or at least partly initiate) the learning opportunity. Ergo, activities listed in the first, and perhaps the second row, of the taxonomy (Table 5.1) may be suitable. In addition, learning opportunities that are a by-product of the developer's main tasks (**B** ⚙), even from the third row, may be better received by such developers, especially those who do not have time to spare or those opposed to mandatory explicit learning, such as mandatory training. In Section 5.3, we focus more on motivations towards software security.

Experience: The teacher's experience is another important aspect to consider. If security expertise is unavailable in the company, the employer should avoid activities that require high internal security expertise. They could consider activities without that requirement (*i.e.*, those marked with a greyed-out 🏛), or activities with security expertise external to the company (*i.e.*, those marked with 🏛 and ➔). When relying on external expertise, it is important to check the source's credibility and to encourage developers to use more credible sources; developers often rely on external resources that are not necessarily ideal for security [11, 61].

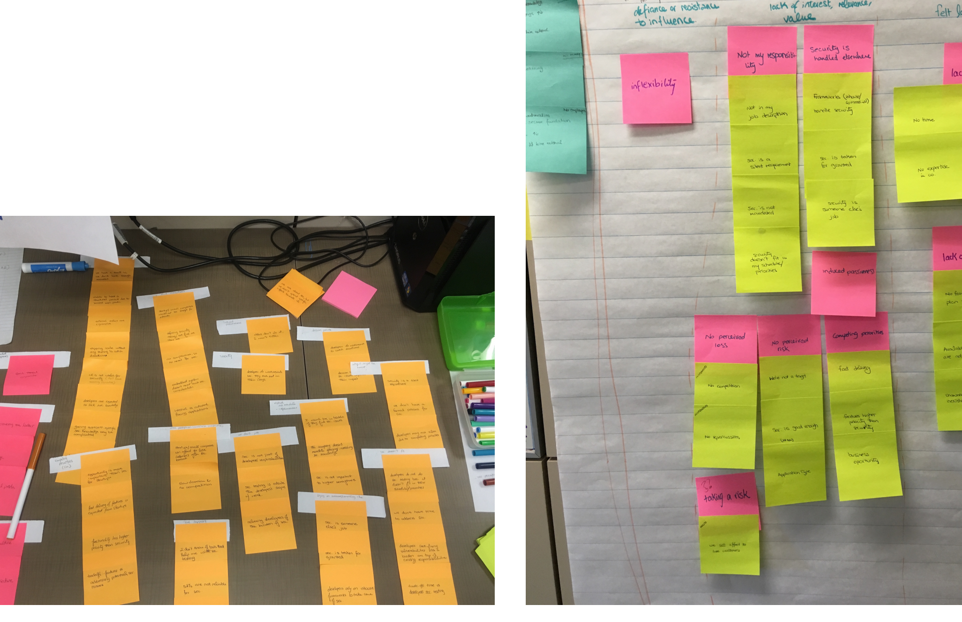


Figure 5.2: Analyzing motivations and amotivations for software security. Left: looking for patterns, right: identified patterns in amotivation.

Budget: The available budget that the employer is willing to allocate for promoting security is another aspect to consider. Fortunately, most learning opportunities derived from our interviews are relatively low cost. However, as mentioned earlier, the affordability of an activity varies between companies, thus employers would need to decide on the most useful activity that fits their budget. In addition, some employers may wish to invest in more expensive learning opportunities, such as offering security courses to their developers, or by hiring external security experts to provide in-context support to developers. Although the latter did not come up in our interviews, it has been reported elsewhere [98].

5.3 Motivation for Software Security

We now focus on motivations for performing security related tasks.

A recognized problem for security is *the unmotivated user property* [174]. This

concept also applies to software developers—security is rarely their primary objective [13, 71]. We analyzed the interviews to explore what motivates developers to adopt or not, security practices and perform security-related tasks. We assigned codes to interview data excerpts relating to developers' motivations, or lack thereof, for software security. We then used Post-It notes for further analysis as described in Section 5.1 (see Figure 5.2). Table B.1 in Appendix B, presents codes corresponding to the software security (a)motivations found in our data, explains each code, and presents a corresponding sample quotation. The table groups the codes following the types of motivations as will be discussed below and as shown in Figure 5.3.

Through our analysis, we found several factors that may induce developers' amotivation towards security, despite their knowledge and belief of its importance. In addition, we identified different motivations to software security. At first, we classified the motivations as intrinsic and extrinsic. However, this classification was too simplistic; the extrinsic motivations identified in our study varied in their driving forces, *e.g.*, an external mandate or the developer's sense of responsibility.

Thus, we found that SDT [47] was a good fit to represent the different motivations and deterrents (amotivations) identified in our study. As explained in Section 2.7.2, SDT [47] presents (a)motivations on a self-determination continuum. SDT [47] recognizes four types of extrinsic motivations that vary in the extent to which their regulation is autonomous [47].

- *External regulation* is the least autonomous and most external to oneself.
- *Introjected regulation* refers to motivations resulting from self-pressure and ego.
- *Identified regulation* is when the goal of the activity was evaluated by the actor and deemed as personally important.
- *Integrated regulation* is when the actor fully accepts the goal of the activity and acts towards it with volition.
- *Intrinsic motivation* is the most autonomous, where activities are performed purely for the pleasure and satisfaction that result from the challenge they present to oneself.

We present (a)motivations identified in our study on the self-determination continuum, as shown in Figure 5.3. The figure colours represent their favourability for

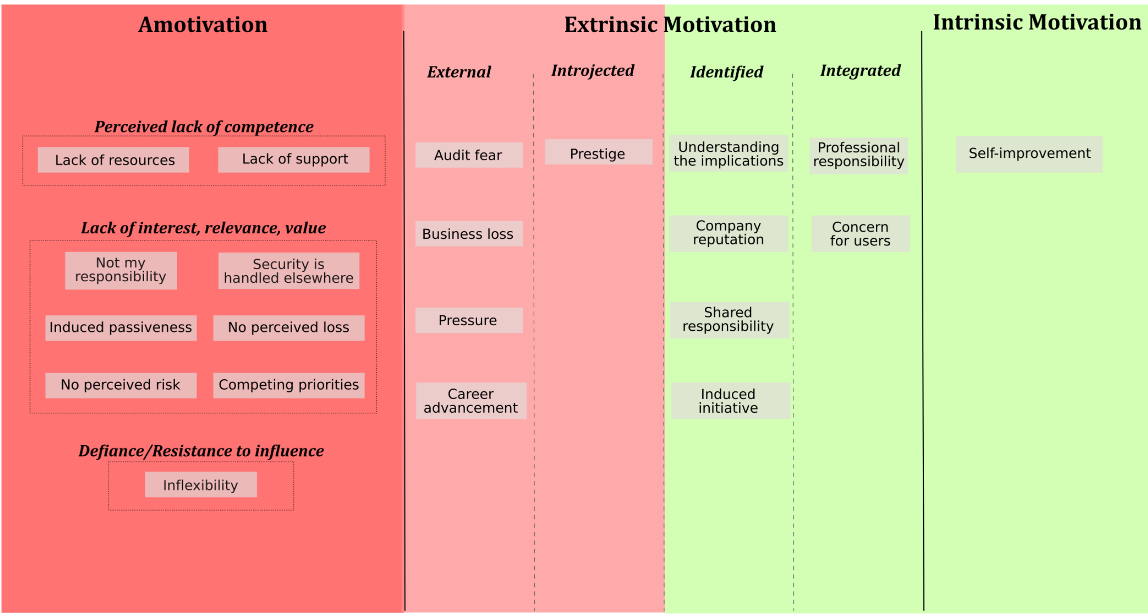


Figure 5.3: The self-determination continuum of software security

software security. At the far left of the continuum, we present amotivations that led participants (or their teams) to not act towards software security. These are coloured in bright red to indicate they are problematic. To their right, we present software security motivations. As we move towards the right, activity regulation increases in autonomy. Motivations under "external regulation" and "introjected regulation" are coloured pale red, because these motivations are not truly internalized and are contingent on their perceived outcomes (*e.g.*, they are performed to comply with regulations or to maintain self-esteem). Motivations under "identified regulation", "integrated regulation", and "intrinsic motivation" are all internally driven (albeit to a varying degree), thus they are coloured in green in Figure 5.3 as they present the most favourable types of software security motivations.

5.3.1 Amotivation

We explored amotivations for software security: why security is deferred or dismissed.

Amotivation - Perceived Lack of Competence

Our analysis revealed that the *lack of resources* and the *lack of support* are two factors that led to a perceived lack of competence to address software security. Some participants indicated that they do not have the necessary budget, time, people-power, or expertise, to properly address security in their SDLC. We also found that this lack of trust in their ability to address security occurs when teams do not have a security plan in place, when security tools are nonexistent or lacking, and when developers are unaware of the availability of such tools. For example, P-T4 said, "*I wish I knew of tools, but unfortunately I don't really know of any tool. So, I would probably be happy to say I would like to use some tools, but I don't know of any. I kinda wish I did, but I don't.*"

Amotivation - Lack of Interest, Relevance, or Value

The other type of amotivation comes from the lack of interest, relevance, or value of performing security tasks. The lack of relevance happens when security is not considered one of the developer's everyday duties (*not my responsibility*), or when security is viewed as another entity's responsibility (*security is handled elsewhere*), such as another team or team-member. Our analysis shows that when this is the general attitude in a team, it could have detrimental effects such as *induced passiveness*. It could lead developers (even those who believe in the importance of addressing security) to become amotivated towards security and rather focus on their 'more valuable' existing duties. For example, P-T10/T11 said, "*I don't really trust them [my team members] to run any kind of like source code scanners or anything like that. I know I'm certainly not going to.*"

Additionally, our analysis shows different reasons why security efforts lack value for some teams in our dataset. First, we found that some of our teams suffer from the optimistic bias [139], thinking that attackers would not be interested in their applications, or that they are not a big enough company to be a target for attacks. Thus, as they see *no perceived risk*, security efforts lack value. P-T7/T8 said, "*For a small company, nobody will usually attack or compromise the vulnerabilities in your system. If something really bad happens, usually, you don't really get enough [bad] reputation*

as well." We also found that when there are no perceived negative consequences to the individuals or to the business from the lack of security (*no perceived loss*), then security efforts lack value. For example, when developers are not held responsible for security issues found in their code, they would rather spend their time on aspects for which they will be held responsible. P-T7/T8 explained, "*[If] I made a bad security decision, nobody would blame me as much as if I made a decision that lead to a [non-security] bug in the system. So the priority of security is definitely lower than introducing bugs in the system.*" Moreover, as different tasks compete for resources (the developer's time in the previous quote), when security has no perceived value, those deemed more valuable are prioritized.

Amotivation - Defiance/Resistance to Influence

The final amotivation we identified is *inflexibility*. In our dataset, we found that some developers would work around security, not because it is difficult to comply, but rather because it conflicts with their perception of the proper way of coding, or it conflict with how they are used to writing code. P-T10/T11 explained how one of his team-members is resisting to use a framework in the proper way, despite having "*gotten into so many arguments*"(P-T10/T11) with his manager, "*I can tell he is very self-absorbed with his own thoughts, and he thinks that what he says is somehow the truth, even if it doesn't necessarily pan out that way*".

5.3.2 Intrinsic and Extrinsic Motivations

Internally-Driven Motivations

We start by the most favourable form of motivation [141], where the driving force is either intrinsic or internal to a certain degree.

We classify *self-improvement* as (the only) intrinsic motivation to security because it is driven by the developer's own interest to challenge herself and improve her capability of producing issue-free code. For extrinsic motivation, *professional responsibility* and *concern for users* are two motivations, where the action is not performed for its inherent enjoyment, rather to fulfill what the developer views as their responsibility to their profession and to safeguard users' privacy and security. For example, P-T3

said, "*I would not feel comfortable with basically having something used by end users that I didn't feel was secure, or I didn't feel respective of privacy, umm so I would try very hard to not compromise on that.*" In addition, we identified motivations, where participants view the goal of addressing security as personally important (*i.e.*, identified regulation). For example, our analysis shows that *understanding the implications* of ignoring or dismissing security increased security awareness and motivated developers and their teams to integrate security in their SDLCs. P-T4 explained, "*I know for me personally when I realized just how catastrophic something could be, just by making a simple mistake, or not even a simple mistake, just overlooking something simple. uhh it changes your focus.*" This was especially true when the understanding comes through practical examples of how the developer's code could lead to a security issue or through experiencing a real security issue at work. Caring about the *company reputation* and recognizing how it could be negatively affected in case of a security breach is another example of identified regulation motivation. Moreover, when the whole project team is responsible for security, as opposed to singling out a specific entity, our participants recognized that as part of the team they should participate in this *shared responsibility*. This could in turn have a snowball effect and lead to motivating more team-members to recognize the importance of considering security since their colleagues do (*induced initiative*). For example, P-T7/T8 said, "*When you see your colleagues actually spending time on something, you might think that 'well, it's something that's worth spending time on', but if you worked in a company that nobody just touches security then you might not be motivated that much.*"

Externally-Driven Motivations

Introjected regulation motivation is when the actions are driven by ego and internal rewards and punishments. Our analysis shows that addressing security can be driven by the desire to be recognized as the security expert or receive acknowledgement (*prestige*), which also helps in maintaining self-esteem and self-worth. P-T1 explained, "*Whenever somebody wants to find about you, then they go and check you in the employee website. Then, when they click your name and check, it shows a badge that you're security certified, which gives you a good feeling.*" We found three external

motivations that are driven by the desire to avoid negative consequences of the lack of security: an overseeing entity finding non-compliance with regulations (*audit fear*), losing marketshare or market value due to a security breach (*business loss*), and being monitored and pressured by superiors (*pressure*). For example, P-T2 explained, "*We have a safety audit, [organizations' names] all these guys they actually send auditors to us every, I don't know, how ever many months [..], and they look at process. They, you know, scan every single check-in, every single review, [...] and they say 'oh, no! You haven't done that, you lose your certification.' We lose our certification, we have no company, we have no customers.*" In addition, we found that receiving rewards in the form of *career advancement* (*e.g.*, "*promotions or move throughout the scales and employment bands*"(P-T5)) is another external motivation for security.

5.4 Internalizing Software Security

As the data analysis progressed, we recognized that our themes could be connected by a central theme about internalizing security—participants' perception of the driving force behind their tasks (*e.g.*, their own will or external factors). This is the last stage of coding in Grounded Theory, selective coding. In our data analysis, we saw varied motivation towards security and varying degrees of internalization. For example, some participants spoke of the importance of security tasks and how they personally value these tasks, while others were indifferent and indicated that security tasks are only performed to satisfy an external driving force.

We developed a human-oriented model that describes the process of internalizing software security based on our analysis (see Figure 5.4). Within the model, the end goal is the internalization of software security. As shown in Figure 5.4, the two levers that influence this internalization are "competence" and "relatedness". The model shows activities that improve competence and relatedness to facilitate internalization. To build this model, we incorporated successful strategies that are currently employed by development teams as described by our participants, while avoiding and addressing conditions that led to a lack of security motivation in some teams.

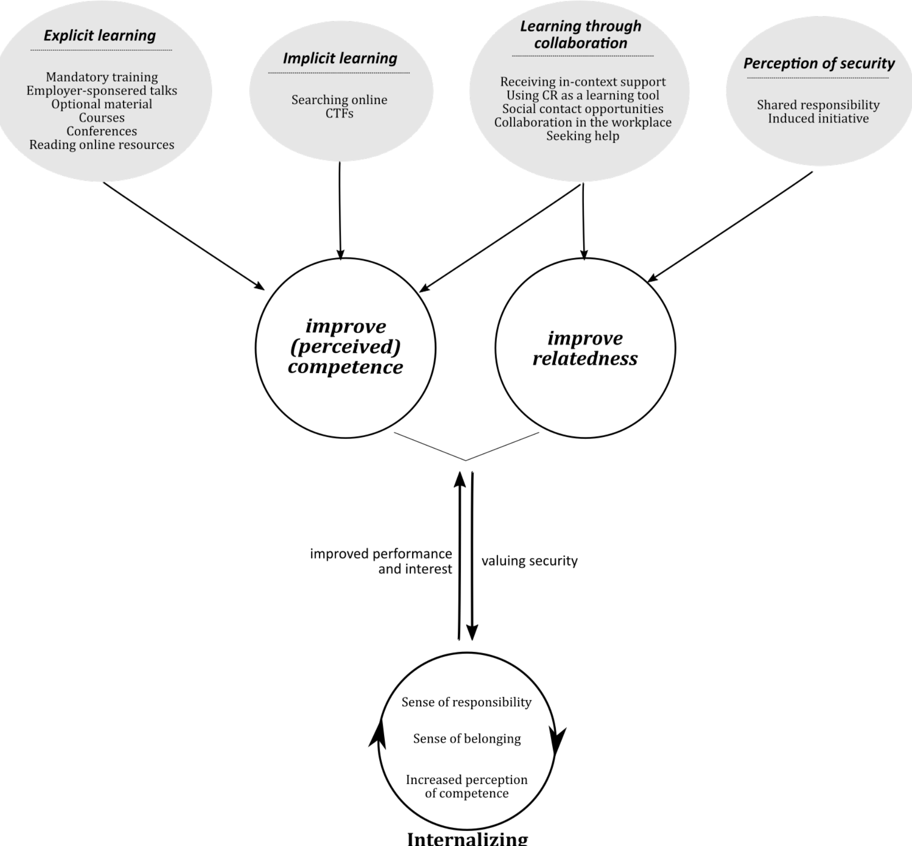


Figure 5.4: Internalizing software security model

Improving perceived competence. Acquiring software security knowledge and expertise improves developers' abilities to address security in their code, as well as their perception thereof. Security knowledge can be acquired through the different security learning opportunities discussed in Section 5.2. In the model (Figure 5.4), we grouped learning opportunities into: explicit learning, implicit learning, and learning through collaboration.

In general, learning opportunities in our model do not focus exclusively on learning about technical details (*e.g.*, types of vulnerabilities and how to fix them), but also highlight the potential adverse consequences of the lack of security. This allows the developer to *understand the implications* of her code on her team and *company reputation*, and users' privacy and security (*concern for users*). Consequently, the developer learns to value security in development and identifies with its goals.

Improving relatedness. Specifically for software security, support for relatedness was achieved by portraying security as a *shared responsibility*; the whole team works together towards producing a secure application. Thus to be an effective part of this team, each team-member needs to do their part. P-T12 said, "*I think it's just a general cultural thing too, you know, everyone knows that people are watching out for you, it's seen as a good thing to kind of, be aware of [security].*"

Improving both perceived competence and relatedness. As shown in Figure 5.4, some strategies improve both competence and relatedness. As team-members and different teams collaborate, this increases their coherence and understanding of each others' work (relatedness), and it also allows for transfer of knowledge between different teams and team-members (competence). These opportunities often include supporting security tasks in-context (*e.g.*, *receiving in-context support* and *using CR*[2] *as a learning tool*), one of the recommended methods of learning about security [119,165]. In addition, they help avoid hindering internalization which may occur when activities are beyond the developer's capabilities or understanding [142]. For example, P-T13 explained their rationale for pairing senior and junior developers, "*It's for more of the mentoring, to make sure that the junior doesn't feel, you know, like they are left alone on the issue or they are frustrated or stuck, and have somebody to kinda guide them through the work that they are doing step by step.*" Although learning through collaboration is implicit learning, we chose to show them separately in the model, as they fall in the intersection between competence and relatedness.

An ongoing process of internalization. As developers' perceived competence and relatedness increase, they gain and deepen their sense of belonging to their team, company, and society, as well as, their sense of responsibility. Thus, developers go into a continuous process of internalizing the extrinsically-motivated security activities, a process of active learning and self-growth [142]. Internalization also promotes growth and coherence within the team and across different teams [142]. As shown in Figure 5.4, this could be seen as an ongoing process; as the developer feels competent and as part of a team that cares about security, she values security and identifies with

[2] code review

its goal. This leads to internalizing security, which in turn improves her performance and interest in security, and so on.

Influential factors. Logically speaking, the duration it takes to (fully) internalize software security would be influenced by the developer's existing security knowledge and awareness. Although we do not have longitudinal data to support this, we expect that, under the same conditions, developers who have prior background in security or have some awareness of its implications would internalize and accept security more readily than those who do not. Of course there are other intervening conditions that influence this process. Some of these condition were discussed in Section 4.4.2. We reiterate them here along with other conditions, in the context of motivation to software security.

As our analysis revealed, the attitude towards software security by the developer's superiors and those up in the company hierarchy has a substantial effect on the developer's motivation to learn about and address security. For example, we found that participants who discussed security with their superiors valued it more than others. For example, P-T14 reported "*As I was working with [my CTO], um, he was telling me, you know, all these different kinds of possible attack vectors that may happen, such as, what happens if the attacker gets access to the actual heap of the program, the memory [...] So stuff like that, I've never had to experience before [...] So, it was really, really interesting.*" Additionally, we found that teams where security was integrated in their culture usually had a security plan to follow. This could be both a cause and effect of motivation towards security; they developed a security plan to motivate security, and their motivation to security improves their security plan (recall it is an ongoing process).

The availability of resources also affects motivation both on the micro level (individual developers) and on the macro level (the whole team). For example, with limited time to work on their tasks, we found that our participants preferred to prioritize their primary tasks; those who were motivated to address security would ask for a deadline extension. If there was leeway, the deadline extension was granted. Otherwise, either security would be deferred or the team would have to assign (or hire) extra personnel. With a limited budget, that may not always be possible.

5.5 Summary

In this chapter, we explored different security knowledge acquisition opportunities. Our data shows that implicit learning, especially when it is part of the SDLC, can be more effective that other types of learning, and can have a positive impact on software security. For example, our findings show that developers engage in learning about security when it is in-situ with their existing tasks. This finding supports previous research recommending teaching developers about security in-context [165]. In addition, we discussed the importance of encouraging collaboration within and between teams. When different teams collaborate, this promotes knowledge sharing between the different team members, which can improve developers' security knowledge, reduce conflicts, and harmonize the team.

In addition, we discussed developers' motivations towards software security. Finding the best way to motivate developers is not a trivial task. Even though external rewards and punishment may help induce external motivation, previous research in other domains [32, 64, 141, 146] suggests that these can have a detrimental consequences, such as negatively influencing conceptual learning and problem solving. Thus, relying solely on external motivations may be a contributing factor to the poor performance and inadequate security practices that we uncovered. From participants in our dataset who had external motivations for security (*e.g.*, audits), those who also had internal motivations had better security processes than those who did not.

Thus, guided by our analysis, we built a model to explain how software security can be transformed to be internally motivated, rather than an external chore. Such transformation occurs by recognizing the value of software security and believing in one's ability to have an impact on the security of the software being built. To improve chances of success, mandating security tasks should be accompanied by improving the morale when it comes to security. Based on our data, this can be through adopting a security culture, supporting developers in these tasks, providing positive encouragement, and allowing teams to see value and identify with such tasks.

Chapter 6

Survey

In this chapter, we present a survey study that tests and expands our findings from Chapters 4 and 5. Specifically, we test whether the security knowledge acquisition opportunities and the motivations identified in our interviews holds with a larger sample. We conducted an online survey to reach participants with a broad range of experiences. In this survey, we focus on how developers and their teams direct their efforts towards software security, as well as the strategies developers employ to deal with security. We also explore developers' work motivation styles, their motivation towards software security, as well as factors that may deter developers from addressing security.

In particular, this chapter addresses the following three research questions.

RQ1 How does security fit in the development lifecycle in real life?

RQ2 What are the current motivators and deterrents to developers paying attention to security?

RQ3 Does the development methodology, company size, or adopting Test-Driven Development (TDD) influence software security?

6.1 Survey Methodology

We conducted an REB-approved anonymous online survey hosted by Qualtrics. The recruitment notice explained that the purpose of the survey is to explore developers' motivation and experience at work, as well as their experience with software security and how it fits in the development lifecycle.

6.1.1 Survey Design

The survey included 10 multiple/single choice questions, 91 Likert-scale type questions, 9 short answer questions, and one open-ended question. The survey also included logic for contingency questions to avoid presenting participants with questions that do not apply to them. Questions that discuss the same topic were grouped to minimize the cognitive load on participants and allow them to consider the topic more deeply [93]. In addition to demographic questions, we asked participants questions to investigate how much effort they spend on security in their development lifecycle, strategies they use to handle security, as well as their experiences with security vulnerabilities. The survey included 5-point Likert-scale type questions to explore developers' motivations specifically towards software security, as well as software security deterrents. To help determine participants' general motivation at work, we included the established 18-item Work Extrinsic and Intrinsic Motivation Scale (WEIMS) using a 5-point Likert scale [167]. Through the WEIMS, we generated the Work Self-Determination Index (W-SDI) which ranges from -24 to $+24$ for a 5-point Likert scale. A positive W-SDI indicates a self-determined motivation profile, whereas a negative score indicates non-self determination [167]. The full survey is included in Appendix C.

We wanted to capture participants' original understanding of software security, thus in the open-ended question, we asked participants to describe what it means to them "to include security into the development process". However, to ensure that participants have a baseline understanding of software security and to avoid confusion, we then provided a brief explanation of software security and how it differs from security functions. This is particularly important for closed-ended questions, where we cannot determine participants' interpretation of software security. For example, if a participant mistakes software security for security functions (*e.g.*, using an authentication scheme), this could negatively affect the integrity of our data. As shown in Figure 6.1, the survey also includes a question to verify that the participant has actually read the definition. Participants were prevented from continuing the survey until they gave the correct response.

For the rest of the survey, when we mention "security", we refer to software security as decribed below. Please note that we are **not** asking about other aspects of security, such as infrastructure and IT security (*e.g.*, ensuring all users in the organization always have software patches installed, and use secure passwords on their accounts)

Software security

- Software security is the idea of building an application that is resistant to: malicious attacks, being used by unauthorized people, or causing harm by inappropriate possibly-accidental use.
- Software security aims to minimize vulnerabilities that could be exploited by attackers (e.g., eliminating buffer overflow vulnerabilities)
- The use of static analysis tools to find potential vulnerabilities in the software being built is an example of software security.

Security functions

- Security functions are the application's security features to protect resources, e.g., authentication to protect user data.
- They can be implemented as functionality within an application (e.g., user authentication)
- Verifying usernames and passwords is an example of security functions.

Which of the following aims to reduce malicious attacks that exploit vulnerabilities?
Please select the most accurate choice based on the description above.

- ○ User authentication
- ○ Software security
- ○ Security functions
- ○ All of the above

Figure 6.1: Explanation differentiating between software security and security functions in the survey.

6.1.2 Testing the Survey Tool

After developing the survey, we went through a pre-testing process to ensure that the survey is clear and well-understood, to test whether questions are measuring what

we intended, and to eliminate any ambiguities. Following the three-stage process recommended by Dillman [51], the survey was reviewed by colleagues and experts in the field to uncover any errors, potential misunderstandings, or any unexpected outcomes. Next, we showed the survey to developers to discuss its clarity and motivation. Finally, we performed pilot-testing with 11 developers to identify any flaws in the survey and to determine whether the survey is of appropriate length. We used the feedback from each stage to update the survey before moving to the next stage.

6.1.3 Participant Recruitment

Participants were recruited through two methods.

We recruited participants through Qualtrics paid service. Participants were compensated by Qualtrics in various ways, such as, SkyMiles, gift card, or points to the equivalent of $6.40 USD. We paid Qualtrics $32 USD per participant for recruitment and data collection.

We also recruited participants through announcing the survey to our professional contacts and contacts in some of the major development companies. Participants received a $10 Amazon gift card as compensation.

6.1.4 Data Quality

To ensure the quality of our data, we took multiple precautions. We provided participants with a description of software security to avoid differences in interpretation and to make sure all our participants were on the same page. Participants were prevented from progressing with the survey until they showed understanding of our description software security (*cf.* Section 6.1.1).

In addition, Qualtrics automatically discarded responses from participants who took less than 7 minutes to complete the survey, as well as those who provided invalid responses, such as entering gibberish in open-ended questions or providing conflicting responses ($n = 17$).

Table 6.1: Summary of participant demographics

Country and Gender		
Canada		63 (51.2%)
USA		60 (48.8%)
Male		93 (75.6%)
Female		28 (22.8%)
Other or not specified		2 (1.6%)
Professional Experience		
Time spent in company		$\mu = 8$ years $Md = 5$ years
Time spent in team		$\mu = 4.6$ years $Md = 2.5$ years
Time spent in dev. in general		$\mu = 16.4$ years $Md = 15$ years
Learned to develop from	Self-taught	22 (17.9%)
	High-school courses	1 (0.8%)
	College courses	22 (17.9%)
	University courses	60 (48.8%)
	Online courses	1 (0.8%)
	Industry or on-the-job training	14 (11.4%)
	Other	3 (2.4%)
Organization Information		
Age		$\mu = 41.3$ years $Md = 20$ years
Size	1-9	5 (4.1%)
	10-249	29 (23.6%)
	250-499	15 (12.2%)
	500-999	14 (11.4%)
	1,000 or more	60 (48.8%)
Team Information		
Team size		$\mu = 13.3$ members $Md = 8$ members
TDD	Yes	32 (26%)
	No	82 (66.7%)
	Don't know	9 (7.3%)
Dev Method*	Waterfall development	27 (22%)
	Iterative (but not truly agile)	26 (21.1%)
	Rational Unified Process	1 (0.8%)
	Agile development	58 (47.2%)
	Other	10(8.1%)

*One participant did not indicate a development methodology.

Table 6.2: Summary of statistical tests

Design	Continuous (normality not assumed) or Ordinal data	
Between-subjects	two groups	Mann-Whitney test
	more than 2 groups	Kruskal-Wallis test
Within-subjects	two groups	Wilcoxon test
	more than 2 groups	Friedman test

Significant values were adjusted by Bonferroni-correction for multiple tests as needed.

6.1.5 Participant Demographics

Through the different recruitment channels, we recruited a total of 140 participants, and we discarded 17 for quality issues. The data reported herein is from the remaining 123 valid responses. Participants are currently working in development in Canada ($n = 63, 51\%$) or the US ($n = 60, 49\%$). Our data includes participants working on a wide range of applications types (see Appendix D.) The survey took an average of 24 minutes to complete ($Md = 17$ minutes). A summary of participants demographic is available in Table 6.1.

6.2 Survey Analysis

All the results presented in this chapter represent participants' self-reported behaviours and attitudes. We analyzed our data using SPSS Statistics v.25. All statistical tests assumed $p < .05$ as a significant level, unless Bonferroni-correction was applied. Table 6.2 summarizes the tests conducted.

Other than eligibility-qualifying questions, none of the survey questions were mandatory. Thus, missing values may exist; these are ignored from the analysis when applicable. In such cases, we indicate the actual number of data points (participants) when reporting the results.

6.2.1 Addressing the Research Questions

RQ1: ***How does security fit in the development lifecycle in real life?*** To address this research question, we looked at how much of development teams' efforts relates specifically to software security and how it is distributed across the different

SDLC stages. In addition, we investigated attitudes and behaviours towards security, as well as experiences with security vulnerabilities and how these experiences influenced awareness and concern for security, if applicable. We also explored participants' strategies to handle software security, and further analyzed these strategies using factor analysis (more details on factor analysis in Section 6.2.2 below).

We performed within-subjects statistical analysis to determine whether security efforts vary across SDLC stages and to explore whether participants' rely on certain strategies more than others. Within subject analysis of strategies was done using the factors extracted from factor analysis.

RQ2: ***What are the current motivators and deterrents to developers paying attention to security?*** The second research question focused on motives to engage in or ignore security. Through Likert-scale questions, we explored different motivators and deterrents to software security. Factor analysis (described in Section 6.2.2) was used to identify patterns in motivations and deterrents.

Within-subjects statistical analysis was used to determine whether the different types of motivators identified through factor analysis are equally effective in motivating security. The same was done for security deterrents.

RQ3: ***Does the development methodology, company size, or adopting TDD influence software security?*** The third research question explores whether specific characteristics influence software security. We focused on three characteristics: the development methodology employed by the participant's team, the company size in terms of the number of employees, and whether the development team employs TDD. To answer this question, we used between-subjects tests to explore whether each of these characteristics influenced efforts towards security, behaviours and attitudes, strategies to handle software security, security motivators, and deterrents to software security.

6.2.2 Factor Analysis

We used factor analysis to analyze participants' security strategies, motivators, and deterrents. Through principal axis factor analysis, we were able to identify patterns

and group closely related information, thus, reducing the set of variables into a smaller set (*factors*), while retaining the majority of the original information [60]. Within and between subjects statistical tests (described above) on strategies, motivations, and deterrents all used the resultant factors.

As recommended, we retained variables with factor loadings with absolute value greater than 0.4 [60,156]. For all our factor analyses, the Kaiser Meyer-Olkin (KMO) measure [83,84] verified the sampling adequacy. Factor analysis results for strategies, motivators, and deterrents are presented in their respective sections.

6.2.3 Developers' Work Motivation

As explained in Section 6.1.1, we used the WEIMS [167] to explore our participants' motivation at work. We found that the vast majority (88.6%) of our participants exhibited self-determined motivation profiles (W-SDI > 0). This is a promising result, as it shows that our participants do not lack motivation when it comes to performing their jobs. In Section 6.4.1, we explore participants' motivators specifically for software security.

6.2.4 Developers' Mental Models of Software Security

We analyzed participants' descriptions of software security and found that the majority of participants (65%) had a reasonable understanding. The majority of participants discussed that software security aims to minimize vulnerabilities, minimize the negative consequences of malicious attacks, and prevent unauthorized access or use of their software or the data it handles. Participants also explained that security should be included from the earliest stages and throughout the development process. For example, one participant described software security as, "*To think about security from the earliest planning phases as possible (at least starting during requirement gathering) and continue to focus on security implications throughout the remainder of the development process.*" In addition, some participants indicated that security defences should be proactive and that developers should adopt an attacker-mindset. For example, a participant said, "*Whenever you start developing a thing, rather than*

asking how will we achieve 'this', you ask how will someone exploit 'this'. Everything will eventually be exploited in some way, and when your processes are done in a proper, security conscious way, as much of the potential harm as possible should be mitigated." Participants also discussed various methods to ensure software security, such as, internal and external audits, security testing, automated checks, code analysis and reviews, thinking about security when writing code, and incorporating security in design. Some participants also discussed the importance of following best practices, using tools and programming languages that have been approved by their company, and receiving support from security experts in the company. Even though the majority of participants' description implied that they value software security, one participant interestingly described software security as "*Processes that slow me down. A necessary evil to protect our clients and company's data.*"

Among the alternative interpretations of software security provided by some participants were maintaining job security, the ability to be creative in their tasks, protecting the codebase, and properly implementing security functions (*e.g.*, encrypting confidential information and passwords).

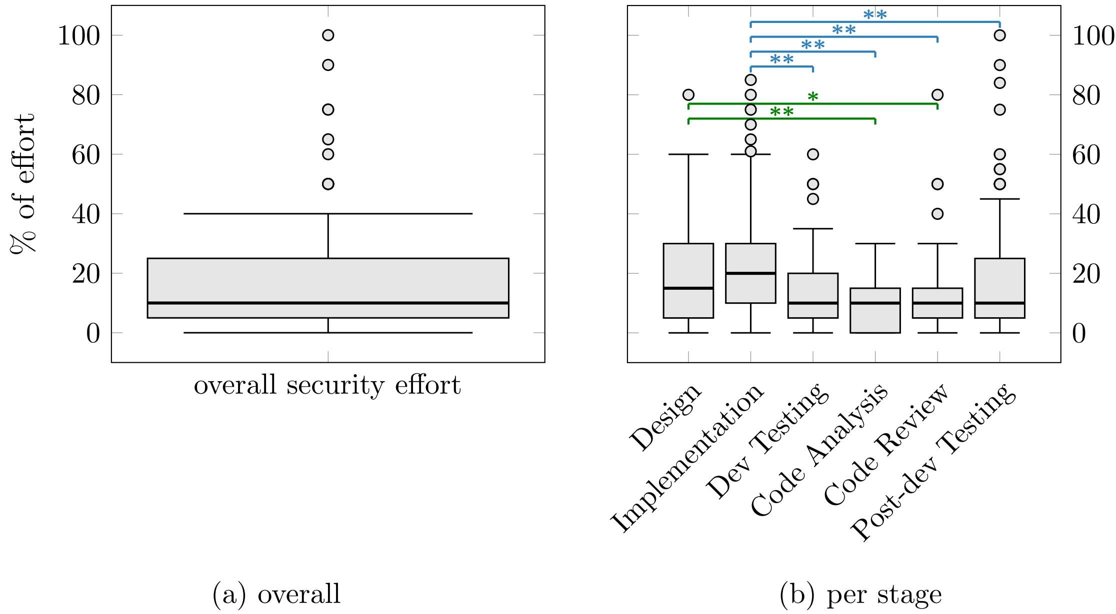
We will now discuss our results arranged by research question. In discussing responses to Likert-scale type questions, we group "strongly agree" and "agree" responses within the text, and likewise group "strongly disagree" and "disagree" responses.

6.3 Security in the SDLC

The survey had seven questions focusing on how software security fits in the development lifecycle (RQ1); exploring development teams' attitudes, efforts, and strategies towards software security. These questions are marked (RQ1) in Appendix C.

6.3.1 Efforts Towards Security

Participants reported the percentage of effort directed towards security out of the overall development lifecycle effort. They also reported the percentage of effort out of all security efforts as a percentage for each stage. The total for all stages must equal 100%.



(a) overall

(b) per stage

Figure 6.2: Software security efforts in the SDLC. (The figure shows stages that significantly differ in efforts towards security. $* : p < .05$, $** : p < .01$)

As shown in Figure 6.2a, participants indicated that on average 18.7% ($Md =$ 10%) of their teams' overall effort in the development lifecycle relates specifically to security tasks. Six participants (4.9%) indicated that their teams do not spend any effort on security. Figure 6.2b shows that overall, the implementation stage had the highest percentage of effort spent on security throughout the SDLC ($\mu = 24.3\%, Md = 20\%$), followed by the design stage ($\mu = 18.4\%, Md = 15\%$). The code analysis stage had the lowest average percentage ($\mu = 9.4\%, Md = 10\%$).

We used Friedman's ANOVA to determine whether the distribution of security efforts significantly differs across the different SDLC stages. Statistical test results are presented in Table 6.3. We found that security effort in the implementation stage was significantly higher than in the code analysis, developer testing, code review, and post-development testing stages. Security effort in the design stage was also significantly higher than in the code analysis and code review stages.

6.3.2 Behaviours and Attitudes

We asked participants to indicate on a 4-point Likert scale their agreement with statements about their teams; how they view software security, whether they have

Table 6.3: Within subject statistical analysis comparing security efforts in SDLC stages. The table only shows significant differences from the pairwise comparisons.

Variable	Test results	
Security efforts	$^{**}\chi^2_F(5) = 78.9, n = 123$	
	design - code analysis	$^{**}r = 0.3$
	design - code review	$^{*}r = 0.2$
	implementation - code analysis	$^{**}r = 0.4$
	implementation - code review	$^{**}r = 0.4$
	implementation - dev. testing	$^{**}r = 0.3$
	implementation - post-dev. testing	$^{**}r = 0.3$

$^{*}p < 0.05,^{**} p < 0.01$, bold item has higher mean

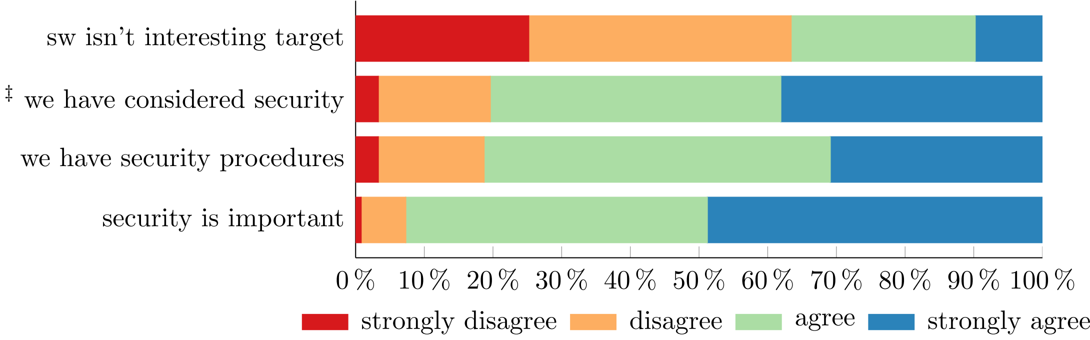
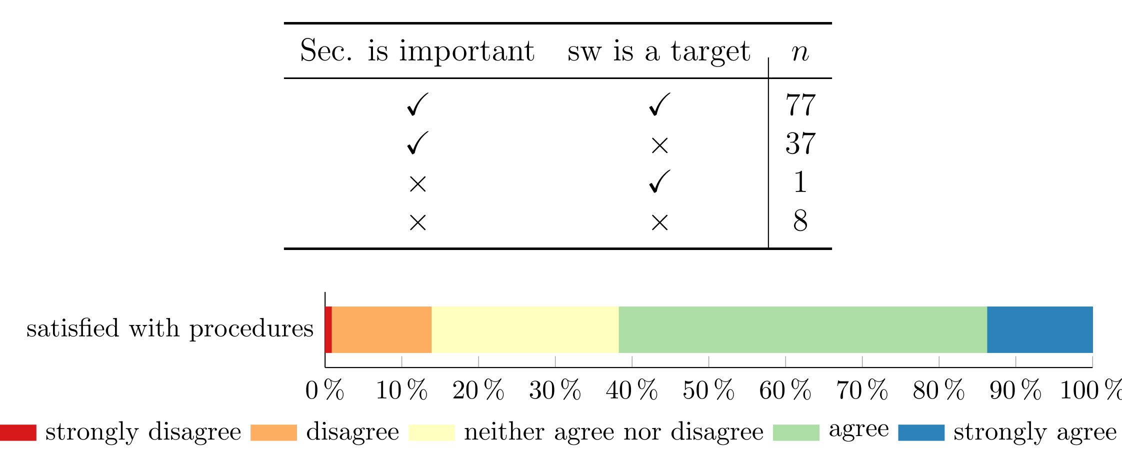


Figure 6.3: Participants' opinion of their teams. (‡: This question was reverse scored and reworded for clarity)

security procedures, whether they consider their software an interesting target for attackers, and whether they have considered the security of their software.

As shown in Figure 6.3, only 20% of our participants indicated that their teams have not considered the security of their software. Although 37% of our participants do no think their applications are interesting targets for attackers, we found promising attitudes towards software security. The vast majority of our participants indicated that their teams believe that software security is important (93%) and that they have specific procedures in place to address software security (81%). All participants, except one, who reported security is not important for their teams also indicated that their software is not an interesting target for attackers (see Table 6.4).

Table 6.4: Number of participants (n) indicating their agreement that security is important for their team and that their software is an interesting target for attackers



Sec. is important	sw is a target	n
✓	✓	77
✓	×	37
×	✓	1
×	×	8

Figure 6.4: Satisfaction with teams' procedures

6.3.3 Experiencing Security Issues

We asked participants to indicate (on 5-point Likert scales) their satisfaction with their teams' security processes and the likelihood that their software contains vulnerabilities.

In general, as Figure 6.4 shows, most of our participants are satisfied with their teams' handling of software security (62%), whereas only 14% indicated dissatisfaction. Despite their satisfaction, only 18% of participants believed their software is free of security issues; more than half of our participants (51%) indicated that software developed by their team likely contains security issues (Figure 6.5).

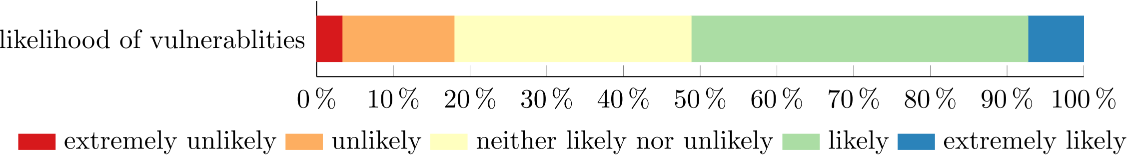


Figure 6.5: Likelihood of the existence of vulnerabilities in team's code

Participants were asked to report whether their software has experienced a security issue at some point.

More than a third of our participants reported that their software experienced

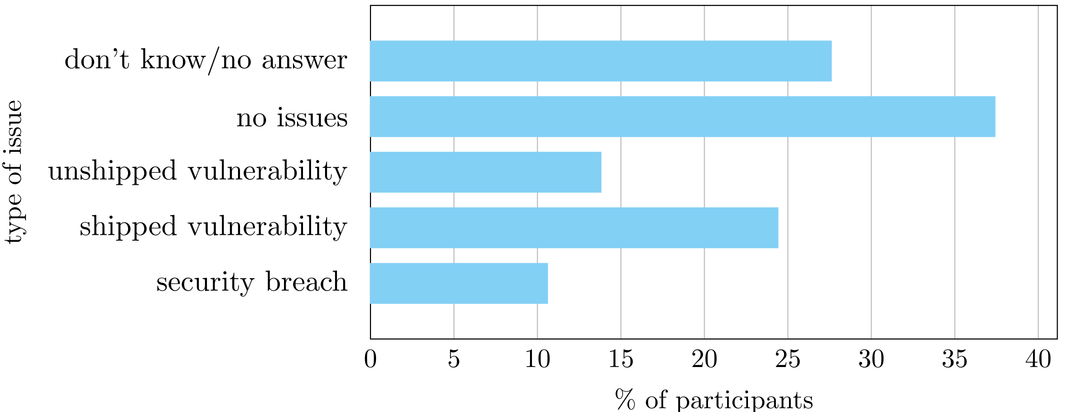


Figure 6.6: Types of security issues experienced by participants' companies.

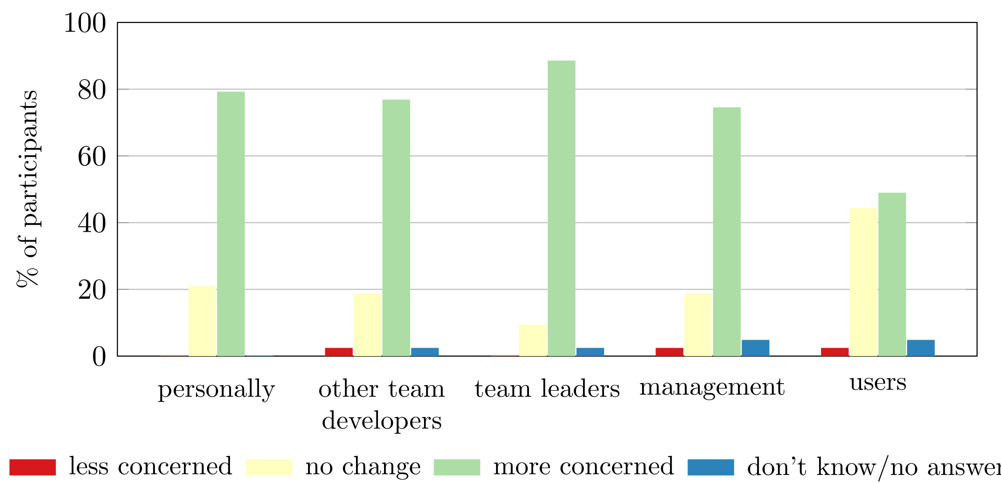


Figure 6.7: Long term effect of experiencing security issues on awareness and concern for security ($n = 43$).

at least one security issue. As shown in Figure 6.6, out of the three potential security issues presented to participants in the survey, vulnerable shipped code was most frequently reported (24%). Fourteen percent of the participants reported their software contained vulnerabilities that were discovered before it was shipped, and 11% reported their software experienced a security breach. We note that these numbers are not mutually exclusive; some participants (11%) indicated that their software suffered multiple security issues.

For participants who reported security issues ($n = 43$), we explored the long-term reaction to experiencing such issues by the different stakeholders. Although it may

be expected that awareness and attitude towards security improves right after experiencing an issue, our data suggests that this improvement in security awareness and attitude is longstanding. Figure 6.7 shows that the majority of our participants (79%) indicated that experiencing a security issue increased their awareness and concern for security over the long-term. Our participants also reported the same effect on other developers in their teams (77%), team leaders (88%), higher management (74%), and users (49%). This supports our findings in Chapter 4 that experiencing a real-threat helps avoid the optimistic bias and can lead to improved attitudes and behaviours towards security.

Forty-four percent of participants indicated that their company experiencing security issue(s) and this did not change their users' awareness and concern for security. "Users" had the highest percentage of "no change" across the different stakeholders, as shown in Figure 6.7. This is reasonable given that users are not typically aware of such software security issues unless, *e.g.*, a security breach is publicized.

6.3.4 Strategies to Address Software Security

We presented participants with a list of 16 potential strategies for handling software security (see Q26 in Appendix C). This list is based on strategies discussed by participants in the interview study. We asked participants to rate their agreement with relying on these strategies on a 5-point Likert scale ranging from 1: (strongly disagree) to 5: (strongly agree).

Figure 6.8 shows participants' agreement/disagreement with relying on different strategies for handling software security. We note that the figure does not show whether these strategies are favourable for security (*i.e.*, blue and green are not necessarily advantageous for security).

As shown in Figure 6.8, most participants indicated that when working on a security issue, they rely on support by their colleagues who faced similar issues. In addition, the majority of participants reported relying on those with more experience in their workplace for security advice. More than half of our participants also indicated relying on their own mental checklists of security issues they need to consider, or on company-wide support, such as security documentations and checklists, and tools

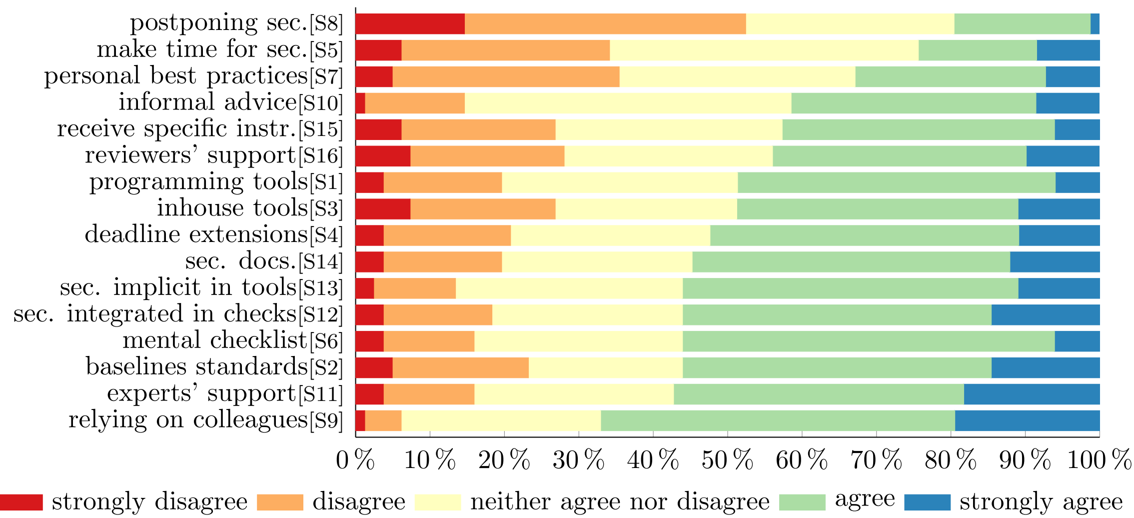


Figure 6.8: Strategies for handling software security ($n = 82$). ([Si] represents the statement's label in Q26 in Appendix C)

and automated checks where security best practices have already been integrated.

We performed factor analysis to integrate these 16 strategies to a smaller set. Table 6.5 presents the factor analysis results. Through our analysis, we found that 12 of the strategies could be grouped into two factors; four strategies did not conform to any factor. We named the first resultant factor: *company-wide engagement*, as it describes how developers rely on strategies and support by their companies, *e.g.*, through relying the more experienced members of the team, or through using custom tools that handle software security. This factor encompassed nine strategies. The second factor incorporated three strategies and is named: *personal strategies*, where developers have devised their own strategies to deal with software security, *e.g.*, through creating their own mental check-list of security issues to consider.

To use these two factors for further analyses, we created a variable for each factor by averaging participants' response to all strategies belonging to the factor. Figure 6.9, shows that to handle software security, our participants ($n = 87$) rely more on *company-wide engagement* than on their own *personal strategies*. A Wilcoxon signed rank test also confirmed this observation ($T = 814, p < .01, r = -0.3$).

Table 6.5: Factor analysis for software security strategies

	Variables (Strategies as presented in the survey)	factor loading
	Company-wide Engagement ($\alpha = 0.9$)	
S13	Software security best practices are incorporated in tools we use	0.9
S12	Software security best practices are incorporated in automated checks we run	0.8
S2	Our company/team has baseline security standards with which 3rd party code should comply	0.8
S11	I can rely on the more experienced members of my company/team for help and security advice	0.8
S9	When working on a software security issue, I can get help from others who worked on similar issues	0.5
S3	We built our own in-house frameworks to help guarantee software security	0.5
S15	I receive specific instructions on how to solve security issues found in my code	0.5
S14	We have a document/checklist of items that we need to consider for our application to be secure	0.5
S16	In code reviews, reviewers explain security issues and fixes to me rather than referring me to resources/books	0.4
	Personal Strategies ($\alpha = 0.6$)	
S6	I have my own mental checklist of software security issues that I need to consider in my code	0.9
S7	I have come up with my own software security best practices	0.7
S5	When a deadline approaches, I try to reduce my workload to focus on securing my software	0.5
	Strategies not belonging to any factor	
S1	We rely on libraries and frameworks (including APIs) to help guarantee software security	
S4	I can get deadline extensions to handle software security	
S8	If I didn't have time to address software security, I'd ship the product after adding a work around that allows me to remotely disable the software feature suffering a security breach	
S10	I prefer to ask for software security advice informally (e.g., by casually asking a colleague, or through discussions over lunch)	

$KMO = 0.8$

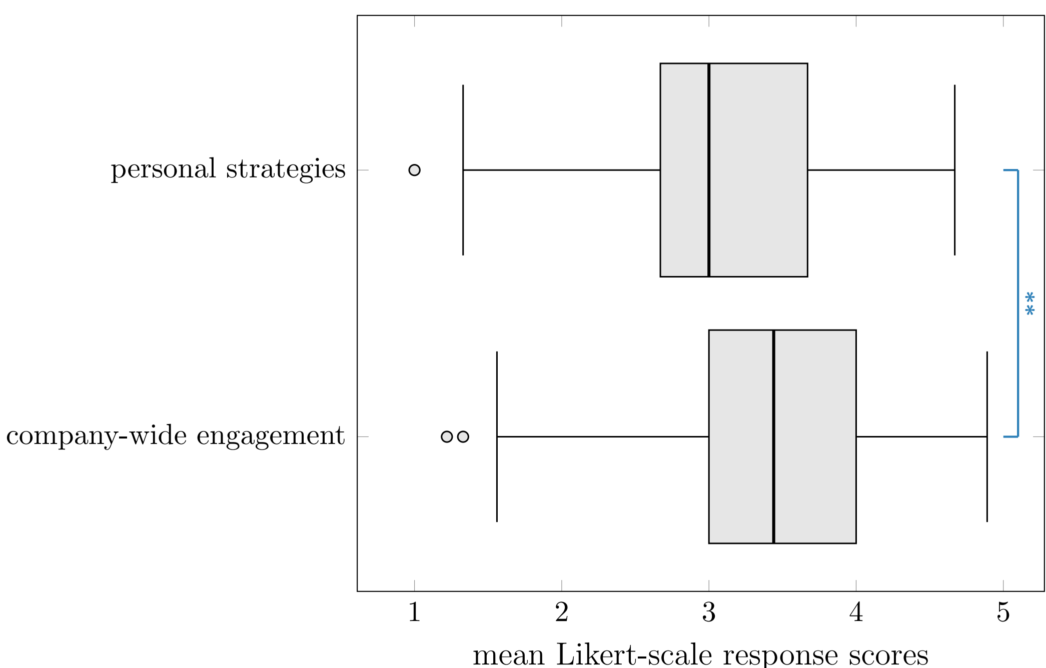


Figure 6.9: Strategies for handling software security after factor analysis ($n = 87$). (1:(strongly disagree) – 5:(strongly agree). The figure shows significant difference between strategies. $* : p < .05$, $** : p < .01$)

6.4 Motivators and Deterrents to Security

To explore what motivates developers to address software security, we presented our participants with a list of 21 potential security motivators (see Q24 in Appendix C). In addition, we presented them with a list of 29 statements that could explain reasons for deferring or ignoring security (see Q25 in Appendix C). Participants were asked to rank their agreement with each statement on a 5-point Likert-type scale. These lists are based on the motivators and amotivators of security discussed in Chapter 5.

6.4.1 Software Security Motivators

We asked participants *"I care about security because..."* and presented them with potential motivations for software security. Participants rated their agreement with each motivation on a 5-point Likert scale, ranging from 1: (strongly disagree) to 5: (strongly agree). Participants also had a "not applicable" option to choose if a motivation did not apply to their current workplace.

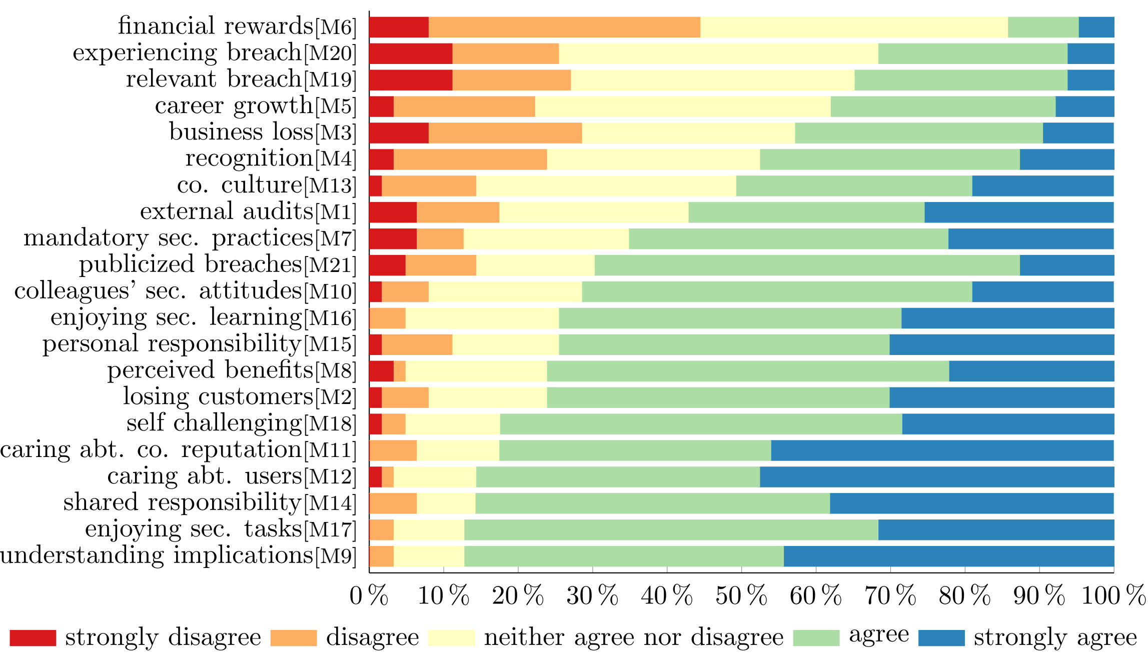


Figure 6.10: Software security motivators ($n = 63$). ([Mi] represents the statement's label in Q24 in Appendix C)

As shown in Figure 6.10, the top six motivators for software security are all autonomous motivations that are internally driven. The vast majority of participants indicated that they care about security because they understand their code can have security implications, they enjoy addressing security issues in their code, they consider security as a shared responsibility by all those involved in the SDLC, or because they care about their users and their company reputation, or because they like to challenge themselves to writing secure code. On the other hand, participants indicated that receiving financial rewards (an external motivation) was least motivating.

We used factor analysis to combine the 21 potential motivators into a smaller set. Table 6.6 presents the results. Our factor analysis grouped 15 of the motivators into four factor; six motivators did not conform to any particular factor. We named the factors: *workplace environment*, *identifying with security importance*, *rewards*, and *perceived negative consequences*. To use these factors for further analyses, we created a variable for each factor by averaging participants' responses to all motivators belonging to the factor. Figure 6.11 shows participants' ($n = 76$)[1] motivations to

[1]We could only include data from participants who answered all questions in each factors.

Table 6.6: Factor analysis for motivation

	Variables (Motivators as presented in the survey)	factor loading
	Workplace Environment ($\alpha = 0.9$)	
M13	Software security is in my company's culture	0.7
M7	My company mandates security practices & I have to follow them	0.7
M10	My colleagues care about software security	0.6
M8	I see the benefit in security practices mandated by my company	0.6
	Identifying with Security Importance ($\alpha = 0.8$)	
M14	Software security is a shared responsibility by all those involved in the development lifecycle	0.8
M12	I care about my users' security and privacy	0.7
M9	I understand that my code can have security implications	0.6
M16	I feel good when I learn about software security	0.6
M15	I see software security as my responsibility	0.6
M11	I care about my company's reputation	0.4
	Rewards ($\alpha = 0.8$)	
M6	My efforts towards software security are financially rewarding	0.8
M4	My efforts towards software security are recognized	0.8
M5	My efforts towards software security help me grow in the company	0.7
	Perceived Negative Consequences ($\alpha = 0.6$)	
M21	I realized securing my code is important after reading about security breaches in the news	0.7
M1	My company is audited for software security by an external entity	0.6
	Motivations not belonging to any factor	
M2	My company would lose customers in case of a software security breach	
M3	My company could fail (cease to operate) in case of a software security breach	
M17	I feel good when I address potential security issues in my code	
M18	I like to challenge myself to write secure code	
M19	Similar software to that on which I work suffered a security breach and management now cares about securing our applications	
M20	Similar software to that on which I work suffered a security breach and it was an eye-opener for me	

$KMO = 0.9$

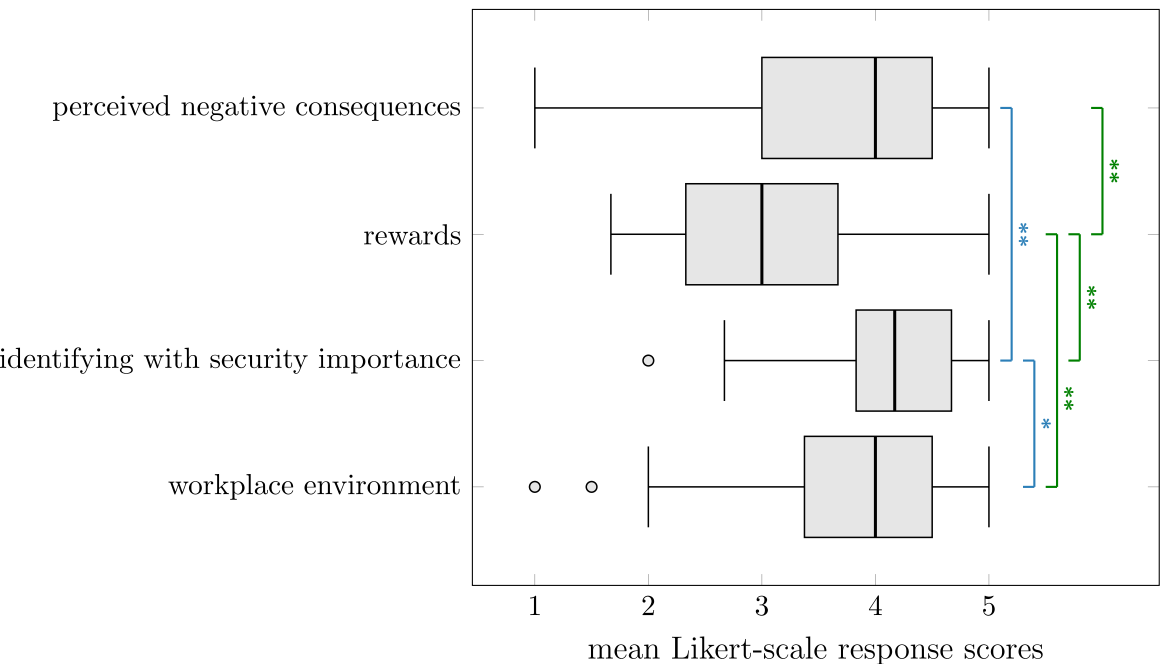


Figure 6.11: Motivations for software security after factor analysis ($n = 76$). (1:(strongly disagree) – 5:(strongly agree). The figure shows significant difference between motivations. $* : p < .05$, $** : p < .01$)

include software security in their work. Whereas external rewards had the lowest median for participants' agreement, identifying with the importance of software security (*e.g.*, viewing security as a shared responsibility and caring about users' security) was the motivator with the highest median. The motivating factor with the second highest median agreement was *workplace environment* (*e.g.*, security is in participants' company culture and their colleagues also care about security).

In fact, we found statistically significant difference between the four software security motivators ($\chi^2_F(3) = 85.75, p < .01$). Pairwise comparisons using Wilcoxon tests with Bonferroni correction were used to follow up this finding. We found that *rewards* was the least significant motivator compared to *workplace environment* ($T = 1.13, p < .01, r = 0.44$), i*dentifying with security importance* ($T = 1.78, p < .01, r = 0.69$), and *perceived negative consequences* ($T = -0.95, p < .01, r = -0.37$). In addition, it appears that identifying with the importance of security is the most motivating factor; it can motivate developers more than *perceived negative consequences* ($T = 0.83, p < .01, r = 0.32$) and *workplace environment* ($T = -0.65, p < .05, r = -0.25$).

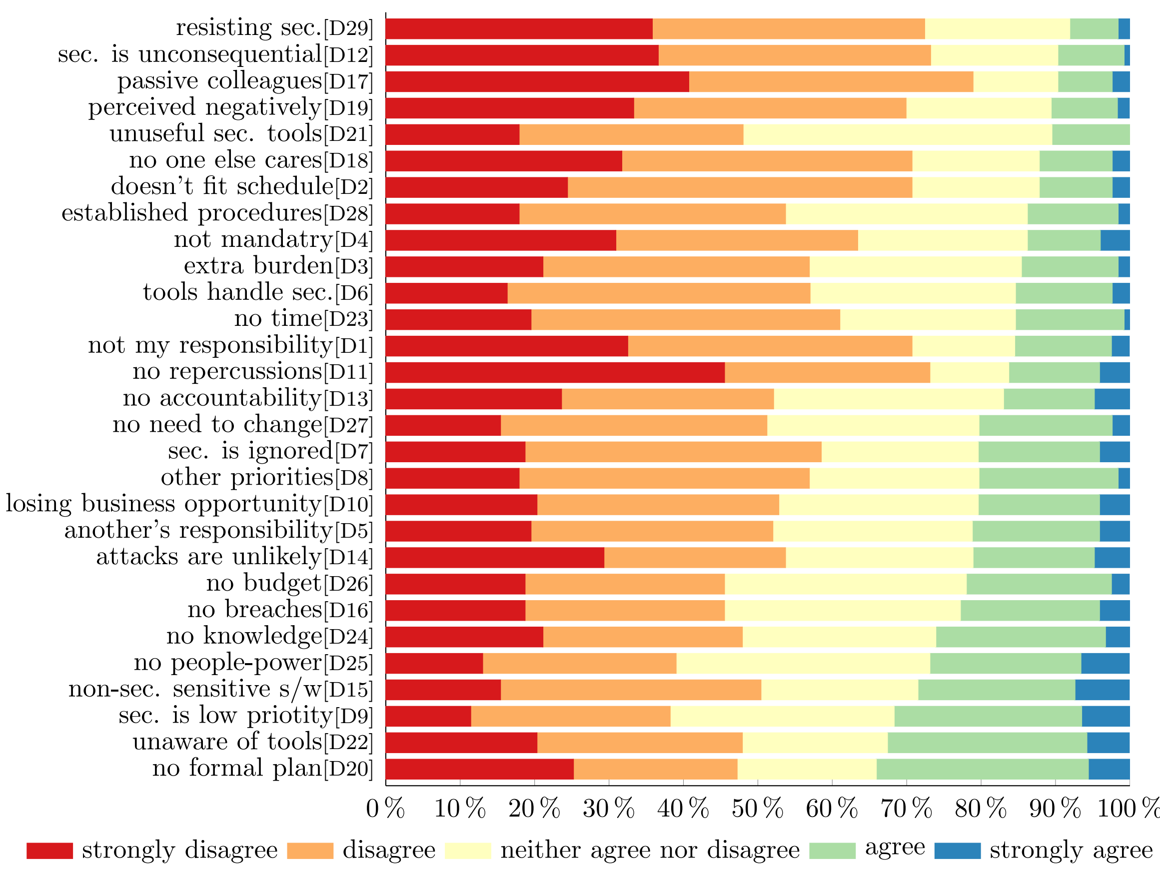


Figure 6.12: Deterrents to software security. ([D*i*] represents the statements label in Q25 in Appendix C)

6.4.2 Deterrents to Software Security

Participants rated their agreement with potential deterrents to software security on a 5-point Likert scale, ranging from 1: (strongly disagree) to 5: (strongly agree). Interestingly, our participants generally oppose statements that imply deferring or ignoring security, as suggested by the overwhelming red and orange chart in Figure 6.12. However, the biggest deterrent to software security was not having a formal plan or process, followed by participants' unawareness of the existence of tools that would analyze their code security.

Our factor analysis combined 18 of the 29 software deterrents into four factors; 11 deterrents did not correspond to any particular factor. Table 6.7 presents the results of the analysis. Our first two factors are *security is irrelevant* and *competing priorities & no plan*. These factors describe how a lack of security can stem from systemic causes

within the company or team such as whether there are consequences for the lack of security, how much of a priority is security, and if there are specific security plans in place. The other two factors, *unequiped for security* and *disillusioned*, describe security deterrents on a more personal level, *e.g.*, a lack of support, knowledge, and awareness can deter developers from addressing software security, as well as being in a workplace environment that thwarts security efforts, rather than nurtures them.

Table 6.7: Factor analysis for security deterrents

	Variables (Deterrents as presented in the survey)	factor loading
	Security is Irrelevant ($\alpha = 0.8$)	
D1	Software security is not my responsibility because it's not in my job description	0.6
D15	The software I develop is not prone to security attacks	0.6
D16	Things are fine as they are, we haven't experienced any security breaches	0.6
D11	There are no repercussions to ignoring software security	0.6
D12	We do not have competition, so we won't lose customers in case of a software security issue	0.5
D5	Software security is handled by someone else in the product lifecycle	0.5
	Competing Priorities & no Plan ($\alpha = 0.9$)	
D20	We do not have a formal process for software security	-0.7
D4	Software security is not mandated by my employer	-0.7
D8	We defer software security due to competing priorities	-0.7
D7	My team doesn't spend any specific efforts towards software security	-0.6
D9	In my team, it is more important to deliver features on time than to address software security	-0.6
	Unequipped for Security ($\alpha = 0.8$)	

. . . continued

	Variables (Deterrents as presented in the survey)	factor loading
D22	I am not aware of tools that would allow security analysis of my code	0.8
D24	I do not have necessary knowledge to address software security	0.6
D21	Available security code analysis tools are not useful	0.5
D28	We have been following the same procedures for years and I don't want to change them	0.5
	Disillusioned ($\alpha = 0.9$)	
D18	I understand the importance of addressing security, but I won't waste my time on it since no one else does	-0.7
D19	I used to push for software security, but I was perceived negatively by my colleagues	-0.7
D17	No one else cares about software security, I won't either	-0.6
	Deterrents not belonging to any factor	
D2	Software security does not fit in my schedule	
D3	Software security is a burden on top of my main responsibilities	
D6	We don't have to worry much about security because frameworks [...] we use handle software security for us	
D10	If we focus more on software security, we might lose our business opportunities	
D13	I won't be blamed if a security issue is found in my code	
D14	It's unlikely that attackers will attack us	
D23	I do not have time to address software security	
D25	There aren't enough people in my team to address software security	
D26	My team does not have the budget to address software security	

... continued

	Variables (Deterrents as presented in the survey)	factor loading
D27	We're doing fine, I don't think we should change in terms of software security	
D29	I tend to resist when I get assigned a security task	

$KMO = 0.9$

Considering the four factors, as shown in Figure 6.13, the deterrent with the least median agreement was *disillusioned* (*e.g.*, due to being perceived negatively by their colleagues when they push for security). The two factors with the highest median agreement for being deterrents to software security were (1) being *unequipped for security* because of a perceived lack of security knowledge or because the necessary tools were unavailable, and (2) *competing priorities & no plan*, where security has a lower priority than other aspects of the software and the team does not have specific security plans or procedures.

We found statistically significant difference between participants' responses for the four factors ($\chi^2_F(3) = 51.1, p < .01$). Pairwise comparisons using Wilcoxon tests with Bonferroni correction, showed that being *disillusioned* was less likely than thinking *security is irrelevant* ($T = 0.7, p < .01, r = 0.3$), having *competing priorities & no plan* ($T = 0.9, p < .01, r = 0.3$), and being *unequipped for security* ($T = 1, p < .01, r = 0.4$). No other pairs showed significant differences.

6.5 Effect of Different Characteristics on Software Security

In this section, we focus on three main characteristics that may influence software security overall. These characteristics are: the type of development methodology used by the participants' teams, the size of the company to which participants' teams belong, and whether they perform TDD. Table 6.8 summarizes our between-subject test results exploring whether each of the mentioned characteristics influences how security fits in the development lifecycle, and motivations and deterrents to software security.

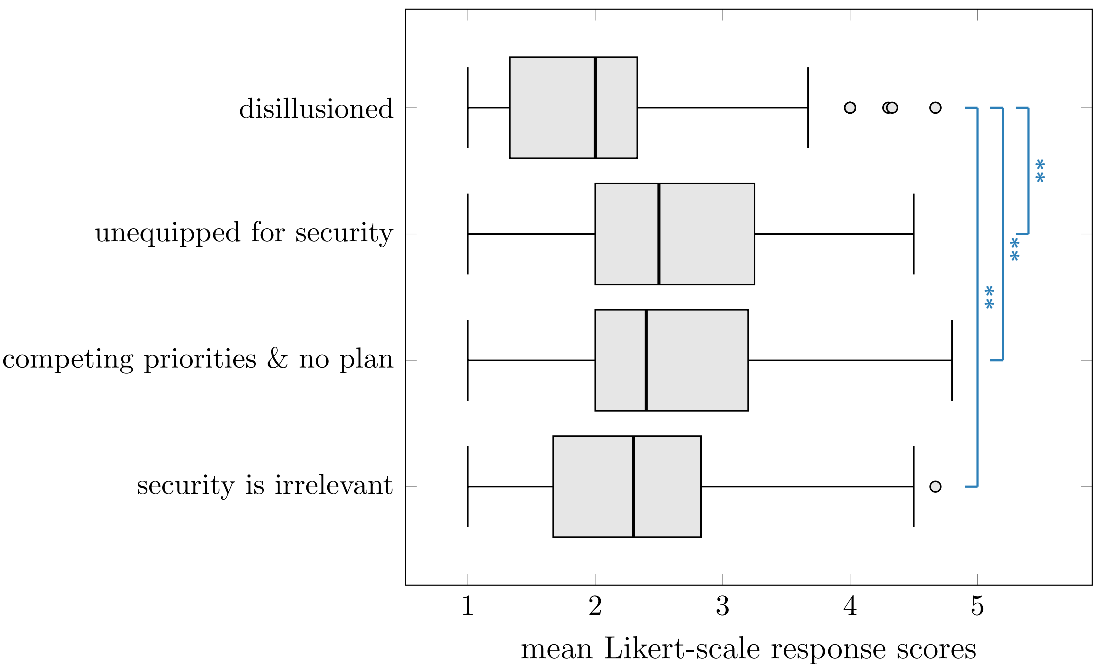


Figure 6.13: Software security deterrents after factor analysis. (1:(strongly disagree) – 5:(strongly agree). The figure shows significant difference between motivations. $* : p < .05$, $** : p < .01$)

More specifically, we focus on whether the development method, company size, or adopting TDD influence security efforts, software security strategies, behaviours and attitudes, security motivators, or deterrents to software security.

6.5.1 Development Methodology

We explore the effect of the development methodology used on software security. Particularly, we focus on the three development methodologies with the highest percentages of participants in our data: Waterfall (22%), Iterative(21%), and Agile development(47%).

Our analysis using Kruskal-Wallis tests with Bonferroni-correction shows that the development methodology did not have a significant effect on any of the variables tested (see Table 6.8). We did not find evidence that the development method influenced the percentage of overall effort that teams direct to software security, nor did it influence how they distribute their effort per development stage. In addition, the

Table 6.8: Between subject statistical analysis of the effect of development methodology, company size, and adopting TDD on software security

Variable	method	co. size	TDD - non-TDD
Security efforts			
overall	*ns*	*ns*	**TDD** - non: $(n = 114)$ $^{**}U = 905, r = -0.2$
design	*ns*	*ns*	*ns*
implementation	*ns*	*ns*	*ns*
dev. testing	*ns*	*ns*	*ns*
code analysis	*ns*	*ns*	**TDD** - non: $(n = 114)$ $^{**}U = 554.5, r = -0.3$
code review	*ns*	*ns*	*ns*
post-dev testing	*ns*	*ns*	*ns*
Behaviours and attitudes			
security is important	*ns*	*ns*	*ns*
we have security procedures	*ns*	*ns*	**TDD** - non: $(n = 114)$ $^{*}U = 995.5, r = -0.1$
sw isnt interesting target	*ns*	*ns*	*ns*
‡we have considered security	*ns*	*ns*	*ns*
Strategies to handle software security			
company-wide engagement	*ns*	*ns*	**TDD** - non: $(n = 80)$ $^{**}U = 397, r = -0.3$
personal strategies	*ns*	*ns*	**TDD** - non: $(n = 80)$ $^{*}U = 506, r = -0.2$
Software security motivators			
workplace environment	*ns*	SME-**LE**: $(n = 76)$ $^{*}U = 861, r = -0.2$	*ns*
identifying with security importance	$ns^{\dagger}$	*ns*	*ns*
rewards	$ns^{\dagger}$	*ns*	**TDD** - non: $(n = 68)$ $^{**}U = 453, r = -0.1$
perceived negative consequences	$ns^{\dagger}$	*ns*	*ns*
Deterrents to software security			
security is irrelevant	*ns*	*ns*	*ns*
competing priorities & no plan	*ns*	**SME** - LE: $^{*}U = 1345.5, r = -0.2$	*ns*
unequiped for security	*ns*	**SME** - LE: $^{*}U = 1413, r = -0.1$	*ns*
disillusioned	*ns*	*ns*	*ns*

$^{*}p < .05, ^{**}p < .01$, bold item has higher mean. *ns*: not significant. ‡: a reverse scored question. †: indicates that test was significant, however, pairwise comparison indicated non-significance

development methodology had no influence on behaviours and attitudes towards security (*cf.* Section 6.3.2), software security strategies (*cf.* Section 6.3.4), or deterrents to software security (*cf.* Section 6.4). Our results indicated that the development methodology may influence some security motivations, *identifying with security importance* ($H(2) = 7, p < .05$), *rewards* ($H(2) = 6.4, p < .05$), and *perceived negative consequences* ($H(2) = 6.4, p < .05$). However, follow-up pairwise comparisons with Bonferroni-correction were insignificant.

6.5.2 Company Size

To explore whether the size of the company influences software security, we classified our participants' companies into either SMEs or Large Enterprises (LEs). Following the classification used in North America [34, 70], a company with fewer than 500 employees was classified as SME, and LE otherwise. Our dataset contained 49 participants (40%) in SMEs and 74 (60%) in LEs.

Using Mann-Whitney tests, we did not find evidence that the company size influenced the percentage of effort on software security, in the SDLC overall or per development stage. In addition, it did not influence participants' security attitudes and behaviours, nor did it influence their strategies towards software security.

However, our results show a significant difference in security motivations between SME and LE participants. Being in a *workplace environment* that nurtures security was more motivating for participants in LEs, compared to those in SMEs.

Our results also show that deterrents to software security vary significantly with company size. Specifically, having *competing priorities & no plan* is a significant deterrent to security for participants in SMEs compared to those in LEs. Likewise, being *unequipped for security*, *e.g.*, not having the necessary security awareness and knowledge, and the lack support, is a significant deterrent to security for SME participants compared to their counterpart.

6.5.3 Test-Driven Development (TDD)

Out of the three characteristics explored, TDD most influences software security.

Our results show that efforts directed towards software security are influenced by

whether the team performs TDD. Participants who perform TDD spend significantly more overall effort on security than those who do not perform TDD. Focusing on each SDLC stage, TDD participants spend significantly higher efforts towards security during code analysis than their counterpart.

Moreover, TDD influences some behaviours and attitudes towards security. Specifically, significantly more TDD participants indicated that they have specific procedures to address software security ($U = 995.5, p < 0.05, r = -0.1$).

We also found that adopting TDD influences software security strategies. TDD participants rely significantly more on *company-wide engagement* for support in handling software security than non-TDD participants. Similarly, they rely on their own *personal strategies* to handle software security significantly more than participants who do not perform TDD.

Finally, our results show that TDD participants are not significantly different than those who do not perform TDD when it comes to security deterrents and the majority of security motivators. However, we found that rewards is a more significant security motivator to TDD participants compared to their counterpart.

6.6 Discussion

In general, our results are promising for software security. Whereas previous research [175, 179, 182] found that developers generally exhibit a "security is not my responsibility" attitude, the vast majority of our participants did not dismiss security. They acknowledge the importance of software security and have specific procedures in place to address it. The few participants who indicated security is not important for their teams indicated that their software is not an interesting target to attackers. We do not imply that completely ignoring security is acceptable, but rather consider the possibility that these teams may be making an *educated* economic decision, having assessed the risk and found that it was negligible.

In addition, our participants exhibited consciousness of the adequacy of their security efforts. It is interesting that some participants ($n = 33$) indicated that security is important for their teams and that they are satisfied with how they are handling software security, but also indicating that their software is likely vulnerable.

We did not expect this combination. One explanation could be that participants were being pragmatic; there will always be security issues and you can never prove security [76]. Another explanation could be that participants are satisfied that they are doing their best to ensure security, even if it may not be ideal or enough. If that is the case, further exploration is needed into whether, and why, these teams' security practices are insufficient. In Chapter 4, we discussed that a lack of resources may discourage teams from addressing security or following security best practices. In fact, we found evidence in our survey data that the lack of resources may be why participants believe that their applications are vulnerable, despite being satisfied with their practices. For example, 45% (15 of 33) of participants displaying this interesting combination indicated they lack at least one of: knowledge, awareness, budget, tools, time, and people-power to handle software security.

We will now discuss and answer the three research questions introduced at the beginning of this chapter.

6.6.1 RQ1: How Does Security Fit in the Development Lifecycle in Real Life?

Effort directed specifically towards software security in the SDLC varied between participants. It is infeasible to determine whether the reported percentages of effort are adequate. In general, we found that participants focus their security efforts during implementing rather than in later stages. Reasons for this are unclear. It could be because they try to get it right from the beginning, thus reducing the effort needed during later stages, or it could be because later stages are mainly functionality-oriented. Although participants reported personally-devised strategies to handle software security (*e.g.*, mental checklists), they also rely significantly on their companies' and teams' support and guidance. Participants rely on their colleagues' support (conforming with previous research [29, 91]) and support from members with higher security expertise. In addition, they rely on their company/team to ensure a secure foundation for their software, *e.g.*, through mandating a baseline standard for third-party code or developing in-house tools to handle security. This re-affirms the importance of companies' role in promoting and ensuring software security, through promoting

security learning opportunities and collaboration between employees, especially for security.

Although promising, the state of software security is not optimal. More than third of participants indicated that their companies faced security issues, ranging from vulnerabilities discovered in unshipped code to actual security breaches. It could be because functionality and on-time shipping was prioritized and security was postponed. In fact, seven out of the thirty participants who reporting vulnerabilities in shipped code indicated that when deadlines approach, they ship their code with a backdoor that allows them to address the security issues later. This behaviour is clearly troubling.

The silver lining to experiencing a security issue is that it improved software security awareness for the different stakeholders (developers, team leaders, higher management, and users). This confirms that experiencing such adverse events helps refute the optimistic bias and promote security awareness. In fact, we found that reading about security breaches, experiencing a security breach first-hand, and knowing about a relevant software that suffered a security breach are all motivators to security. However, we could not determine which had greater impact on our participants.

6.6.2 RQ2: What are The Current Motivators and Deterrents to Developers Paying Attention to Security?

Participants' software security motivation pattern appears to match their general work motivation pattern. The vast majority of our participants are self-determined and autonomously motivated towards their work in general, as well as towards software security. Their top software security motivators are all intrinsic and internal motivations. In fact, identifying with the importance of software security was the highest motivator, whereas receiving rewards, as an external motivator, was least popular.

On the other hand, confirming our previous findings (*cf.* Section 4.4.2), being ill-equipped to handle security (*e.g.*, due to lacking resources or support) was the highest deterrent to software security. Next was not having a specific security plan and having to deal with competing priorities. These are reasonable reasons that

may prevent teams or developers from focusing on software security. By identifying these deterrents, we can focus our efforts on overcoming them, *e.g.*, as discussed in Chapter 4, by devising lightweight best practices that do not require extensive resources, or through better support for teams to devise security plans that fit their resources and work styles.

6.6.3 RQ3: Does the Development Methodology, Company Size, or Adopting TDD Influence Software Security?

Contrary to previous literature [26, 31, 144], our results suggest that the development methodology does not significantly influence teams' handling of software security.

On the other hand, company size had some effect, impacting software security motivations and deterrents. Being in a workplace environment that, *e.g.*, promotes security culture and practices, was more motivating for LE developers, compared to those working in SMEs. For SMEs, being unequipped to handle security and having to deal with competing priorities while lacking concrete security procedures and plans were significant deterrents to software security. It appears that LEs , being more established compared to SMEs, can focus on building a security-oriented workplace and can afford to prioritize and plan for security.

TDD was the most influential characteristic that we explored. Adopting TDD appears to be associated with higher security efforts overall, and particularly during the code analysis stage. In addition, TDD teams appear to take a more structured approach towards security by having specific security procedures in place, compared to non-TDD teams. Moreover, TDD developers were more likely to rely on company-wide strategies and support, as well as their own personally-devised strategies, to handle software security. In terms of security motivations, receiving rewards is a significant motivator for TDD developers compared to their counterpart. Adopting TDD did not influence security deterrents.

6.7 Limitations

Our survey was conducted online, which may have influenced data quality. However, we took different measures to filter out poor quality responses. All our results are

based on participants' self-reported responses, which may be subject to bias and may not exactly represent real-life. For example, in the security motivations questions, participants were given a "not applicable" option in case a motivator did not apply to their workplace. However, we cannot be sure if all participants chose this option when a motivator did not apply to them. In addition, during our factor analyses, Likert-scale data was treated as continuous data, rather than ordinal data. However, some researchers [68,97] argue that Likert-scale data may be used in parametric tests, especially if they employ at least a 5-point scale [85].

6.8 Conclusion

We presented a survey study with 123 participants to explore how they address software security, as well as security motivators and deterrents. Participants consider security as part of their development process to varying degrees. Most interestingly, we believe that our results affirm that *developers are not the weakest link*. Our analysis shows that our participants are self-driven in their work in general, as well as in their motivation towards software security. Thus, developers in our study are not explicitly ignoring security, dismissing it, or considering it not part of their responsibility. In fact, identifying with the importance of software security was the highest motivator. In addition, developers in our study are not oblivious to the state of security; they believe that it is important and some of them are not always satisfied with their teams' processes. Although typically not their primary task, the majority of participants objected to statements in our survey that implied deferring or ignoring security.

Chapter 7

Discussion, Future work, and Conclusions

In this chapter, we discuss our research contributions, provide insights on conducting similar studies with developers, answer the research question posed in Chapter 1, and provide concluding remarks.

7.1 Thesis Contributions

In Section 1.3, we discussed the research question and objectives for this thesis. In general, this thesis adds to the growing body of research focusing on the human factors of software security, specifically organizational processes, strategies employed by developers to handle software security, and developers' security knowledge, motivation, and attitude. We now outline the contributions of this thesis.

Supporting collaboration and exploration during security code analysis. We evaluated the usability of FindBugs [5], a popular SAT, to explore issues faced by developers while analyzing their code. Our study revealed serious usability issues. We then took a user-centered approach to design a visual environment to support source code analysis, while overcoming the usability issues we uncovered in FindBugs. We prototyped and evaluated the usability of our approach. Indeed our evaluation suggests that our approach can support collaboration amongst developers and encourage discussion and exploration of potential issues. We also provided general recommendations to guide future designs of code review tools and enhance their usability.

Investigating software security status-quo. Through interviews and an online survey study with software developers, we investigated software security status quo. Our studies revealed varying approaches to software security. To facilitate comparison of existing practices to best practices, we amalgamated software security best practices

extracted from the literature into a concise list (Section 4.3). Interview data analysis showed that real-life security practices differ markedly from best practices identified in the literature. Best practices are often ignored, as compliance would increase the burden on the development team. However, the survey data implied that developers are not intentionally ignoring security, and in some cases, they are even dissatisfied with their teams' security processes. We discussed factors that influence software security processes (Section 4.4.2), such as security knowledge and external pressure.

Proposing a security knowledge acquisition taxonomy. We developed a taxonomy (Table 5.1) of software security learning opportunities through further analysis of interview data. Some of these opportunities were not explicitly regarded as learning methods, though our analysis revealed their potential for knowledge acquisition. We explained how the learning opportunities varied in their formality and the developers' motivation to initiate them. Opportunities where learning was a by-product of developers' tasks were most common. We discuss how opportunities that require collaboration within the project team could lead to more coherent teams and help bridge the gap between developers and security experts.

Exploring motivations and amotivations for software security. We explored factors that could influence developers' motivation towards software security (Figure 5.3). We identified factors that may lead developers to become amotivated towards software security, despite their security background. In addition, we identified motivations to software security and explained how they vary in their degree of internalization (whether developers perceive their actions as self- or externally-driven). We discuss the importance of transforming software security activities from being externally-driven to being driven by developers' own volition. We also discuss how this can have a positive impact on performance and learning.

Proposing a model for internalizing software security. To help promote the internalization of software security, we proposed a human-oriented model (Figure 5.4) to describe how software security external motivations can be transformed into internal motivations. The model highlights the role of knowledge acquisition opportunities

and collaboration within the project team in promoting and facilitating the internalization process. We based the model on successful motivation strategies identified in our studies, while overcoming factors that could lead to amotivation.

7.2 Insights on Conducting Studies with Developers

In the research presented herein, we recruited developers currently employed in industry for our interview study and online survey study. In our experience, recruiting developers was a more complicated process than recruiting typical end-users.

Interview studies. Participants were generally busy and time-constrained; interviews were usually held during lunch breaks, in the evenings, or on the weekend. Some participants had to reschedule multiple times due to approaching deadlines or unexpected conflicting team meetings.

In addition, developers may be restricted by their companies on what they can reveal about their processes and their typical workday, or they could just be self-conscious about what they say "on-record". For example, while informally talking to developers about our research during development-oriented meet-ups, they were engaged in the discussion, offering their insights and their own experiences. However, the majority were reluctant to participate in the interview study. One developer explicitly explained that he was worried that participating would be against his company policy.

Moreover, software security can be a sensitive topic. For example, when developers talk about specific practices or vulnerabilities in their code, this could later be exploited by malicious actors if the company's identity was somehow revealed. In our interviews, multiple participants revealed their company's name and names of partner companies, despite our instructions not to do so. In addition, some developers may be worried that revealing some information may negatively influence their companies' reputations. For example, one of our participants repeatedly confirmed that his company would not be linked to his interview data before discussing flaws in their security practices.

Suggestions for future research. Rather than addressing all the research topics

in one interview, perhaps researchers could divide the topics across multiple interviews. This would shorten the duration per interview session, which could encourage more developers to participate. Also, in addition to typical recruitment methods, researchers could consider reaching out to managers or team-leaders to announce studies to their developers, if participation was allowed per company policy. This could help alleviate worry of developers who are concerned that participation is frowned-upon by their company. Moreover, clearly explaining to developers the interview's topics of discussion, as well as the precautions taken to protect their and their companies' identities may ease developers' skepticism and encourage them to participate.

Online surveys. We recruited participants using a paid-service, as well as through announcements to developers and snowballing. In general, participation rate in the survey study was better than the interview study, which could be because the survey was anonymous and required less time-commitment.

Recruitment through the paid service was relatively fast, yet expensive. Thus, for large projects, using the paid-service would require a considerable budget. In addition, some participants recruited through the paid service were not paying proper attention, or were just clicking through the questions. For example, some participants would (illogically) indicate they spent longer in their current teams than in their companies. Participants recruited through the paid-service were also more likely to skip qualitative questions, or include minimalistic (unhelpful) responses. On the other hand, participants recruited through announcements and snowballing provided better data quality and more insightful responses to qualitative questions. It seems that recruiting through announcement and snowballing creates an implicit social contract that is lacking when recruitment occurs through a paid-service.

Suggestions for future research. Employing measures to ensure data quality is important for survey studies. Our experience showed that it is especially important when recruiting through a paid-service, where some participants may be gaming the system. Some services offer to replace poor quality responses at no additional cost, thus researchers have the opportunity to weed out such poor quality responses by employing multiple strategies in their surveys. We looked for inconsistencies, such as

comparing duration participants have been working in development in general versus in their current company, or comparing responses to a factual question that was asked multiple times. In addition, in our experience, participants gave relatively brief responses to qualitative questions, thus this type of data may be better collected through interviews. Surveys can be used with closed-ended questions (*e.g.*, multiple/single choice questions and Likert-scale type questions), and include open-ended questions for clarifications.

7.3 Answering the Thesis Research Question: Recommendations for Supporting Developers

Software developers are responsible for making decisions that can impact the security of their whole user-base [13]. However, developers do not always have the security expertise to make these decisions [13, 71]. In this thesis, the main research question is "*How can we support the human dimension of software security throughout the SDLC by better understanding factors that motivate, or impede, security efforts?*" Based on our research, we make the following recommendations to support efforts towards software security without adding a substantial burden on the developers.

> **Encourage sharing of responsibility and collaboration.**
> Companies should portray security as a shared responsibility of all those involved in the development process while promoting collaboration within and between teams.

As shown in our analysis in Chapter 4, some developers are dissociated from the responsibility of software security—developers implement while security testers handle security. In addition, our analysis in Chapter 4 (page 75) showed an example where relying on a single operations-level engineer resulted in adhoc security processes and led developers to perceive the engineer as an outsider. In contrast, we found that considering security a shared responsibility leads to better engagement in security processes and less push-back. We do not imply that the security novice and the experts have equal responsibilities, rather that no one should be completely

exempt from the responsibility. "Recognize that defense is a shared responsibility" is one of the best practices identified in the literature (*cf.* Section 4.3), however, we extend it to highlight that this has to be infused with the encouragement to collaborate within and between the different teams involved in the SDLC. Chapter 5 explored how developers collaborate with each other as well as with other teams, *e.g.*, through seeking the advice of the more experienced. We found that these developers spoke about their teams as unified systems working towards a common goal, and that through collaboration, each team worked towards acquiring deeper knowledge of the software from the other teams' perspective. Thus, as suggested by our analysis, collaboration leads to more coherent teams, reduces conflicts, and helps bridge the gap [165] between developers and security experts. Our prototype presented in Chapter 3 showed how a visual analysis environment could support collaboration and encourage discussions and exploration during security code analysis.

> **Support developers in-context.**
> Companies should support developers in-context of their tasks to reduce frustration with security-related tasks, and further promote collaboration between developers and security experts.

Security in software development is cognitively-demanding [119] and developers need support especially when dealing with security tasks [71]. Previous research recommended in-context security training for developers [119] (*e.g.*, through the inclusion of tools in their IDEs [115, 165, 180]). We extend this by also recommending facilitating human support for developers in context of their tasks. Beyond tool support, we highlight the importance of "people support". We found, conforming with previous research (*e.g.*, [91]), that developers often turn to their colleagues for help, and that interacting with those who have security expertise can positively influence software security. Our analysis revealed approaches towards incorporating people support in-context of developers' tasks, to help reduce frustration with security-related tasks, and further promote collaboration between developers and security experts. For example, this was done through establishing a security advisory group that developers

can turn to for support and advice about securing their applications. In-context people support was also done by ensuring that code review and testing feedback is informative, *e.g.*, that it opens a dialogue with the developers, and that it includes steps to follow to reproduce the security issues detected and specific steps that the developer can follow to fix these issues.

> **Support implicit learning opportunities.**
> Companies should support implicit learning opportunities to promote security knowledge without burdening the developers.

In Chapter 5, we provided a taxonomy of different security learning opportunities identified in our data. Although not explicitly regarded by our participants as learning methods, we identified implicit learning opportunities in-context of developers' tasks. For example, participants discussed that code review, testing feedback, and advice provided by security experts includes an explanation of the types of issues detected and sometimes even explains their implications. In Section 5.2.4, we discussed how this personalized and practical type of learning can be effective in acquiring security knowledge compared to the one-size-fits-all mandatory training. Task-related security knowledge is especially important since developers are typically not in a security mindset while coding and have difficulty mapping abstract security knowledge to their tasks [119]. By virtue of being part of their tasks, developers may engage in such hands-on learning, rather than perceiving it as an extra unnecessary burden.

> **Have an extra set of eyes.**
> Companies should bring in a new perspective to software security analysis by including an entity that has not been directly involved in the development process.

Including an entity that has not been directly involved in the development process (*e.g.*, testers, reviewers, auditors) in analyzing software security can bring in a new perspective who may see beyond best-case scenarios. Previous work [165] found

that some teams rely on auditors to identify vulnerabilities, whereas developers are responsible for addressing them. Similarly, security practices discussed in Chapter 4 showed that some teams have security checkpoints independent of the development team. This recommendation does not conflict with our previous recommendation: "Encourage sharing of responsibility and collaboration". In fact, we highlight the importance of positioning this step as part of the SDLC that is intended to help the developers, rather than laying blame on the developers or relieving them of their responsibility towards software security. Our work showed that collaboration between developers and such (external) entities can help avoid negatively perceiving them as outsiders solely responsible for security and with which developers just have to comply. In addition, our analysis suggests that collaboration is crucial to avoid conflicts, to make sure this entity has proper understanding of the software, and to reduce false positives and false negatives.

> **Focus on internalizing security.**
> Companies should focus on "humanizing software security" to encourage developers to act towards it with volition.

Actions that are prompted by internal motivations are associated with better performance, engagement, and cognitive abilities [141]. Since software security is typically a hard and demanding task [71, 119], acting towards software security with internal motivations is likely to produce promising results. In fact, we recognized through our analysis in Chapter 5 that developers who acted towards security for reasons that extend beyond mandates had better security processes than others. In Section 5.4, we discussed how companies and teams can facilitate the internalization of security by providing adequate support, promoting collaboration between developers and security experts, and supporting security learning opportunities. In addition, our analysis suggests that *humanizing software security* can promote internal security motivations, *e.g.*, by familiarizing developers with the implications of the lack of security on users and organizations while providing clear and concise examples from real-life security breaches. Moreover, although this has not been directly addressed

by our research, we believe that the process of internalizing security can start with education; software development courses can make developers aware of the consequences on humans in addition to the technical aspects of security. To the best of our knowledge, our recommendation for internalizing software security has not been identified in previous literature.

We drew on our research to provide these recommendations to support developers throughout the SDLC. We acknowledge that implementing these recommendations is not a trivial task, and that it requires commitment (and potentially resources). However, our findings suggest that striving to accomplish these recommendations is worthwhile for improving the state of software security. Our recommendations aim to reduce the cognitive load on developers, to facilitate security learning, and to improve the coherence and the security culture in teams. In addition, they aim to promote security through internal motivations, which are associated with improved engagement and performance.

7.4 Future Research Directions

This thesis focused on a research area in usable security that is still in its early stages [13, 65]. In this section, we discuss further research directions.

Observing development teams in their working environment. In this thesis, we took a holistic perspective to software security, focusing on security practices, security motivations, behaviours, and attitudes. Our results are self-reported through interview and survey studies, and the reasonable next step is to conduct ethnographic studies to observe development teams in their workplace. This will provide ecological validity to the results and reduce biases. Through ethnographic studies, researchers could discover pain points in development and security processes that may not be evident to developers. In addition, as we found that management can greatly influence software security processes and motivations, it would be interesting to further investigate developers' interaction with their superiors and how this affects developers' security practices and motivations. Moreover, in Chapter 4, we found that some participants' teams were successful at building a security-oriented culture.

Through ethnographic studies, researchers can explore these teams' motivations and approaches, which could help other teams in adopting successful security approaches.

Reducing the cognitive load. Performing security tasks and processing security information introduce a substantial cognitive load on developers [119]. As shown by our work and previous research (*e.g.*, [13, 119, 165]), developers do not always have the necessary expertise or time to address security. We highlight the need for better support for developers through tools and methodologies that reduce this burden. For example, OWASP currently has a "Secure coding practices checklist" [126] composed of 214 generic items, some of which may not apply to all applications. Thus, developing concise application-specific checklists may be an interesting next step. In addition, existing code analysis tools are lacking in terms of visual representations. Incorporating visualizations may improve vulnerability analysis by encouraging collaboration, exploration, and the discovery of hidden patterns in the code. Moreover, further research is needed in evaluating and improving the usability and security of APIs and frameworks that can help take the developer out of the loop.

Exploring the effectiveness different types of motivations. Our studies revealed different motivations to software security that vary in their degree of internalization. Some companies use external motivations (*e.g.*, rewards) to promote security. However, previous research [141, 142] found that they have a detrimental effect on intrinsic motivation as it shifts the perceived locus of causality from internal to external. As a future research direction, the effectiveness of different types of motivations in encouraging and improving security could be explored. In addition, we highlight the need for future research focusing on the long-term effect of rewards (and punishment) on intrinsic motivation and the internalization of software security.

Develop empirically-evaluated security best practices. Through our analysis in Chapter 4, we found that available best practices fail to discuss the baseline for ensuring security, or how to choose which best practices to follow based on limited resources and expertise. It was also interesting to find that most security best practices are from industry sources and are not necessarily empirically verified. For

future research, we suggest devising a lightweight version of security best practices and evaluating its benefit for teams that do not have enough resources to implement security throughout the SDLC, or when implementing traditional security practices would be too disruptive to their workflow.

7.5 Conclusion

This thesis challenges the position that software developers are the reason for security issues. The findings herein assert that *developers are not the weakest link*. Through this thesis, we have identified various factors that influence security processes and motivation to address security. These factors include company culture towards security, resources available to address security, external pressures (*e.g.*, security audits), experiencing a security incident, the development team's division of labour style, and the availability and usability of tools that identify software security issues. The developer's security knowledge and access to security resources in the company are also influential.

Focusing on the human dimension of software security, our main goal in this thesis is to support developers through better understanding their motivations, attitudes, and processes to handle security. We presented a visual analysis environment to support security code analysis while encouraging collaboration between developers and reviewers. We investigated how security currently fits in the development lifecycle, and identified discrepancies between existing practices and best practices in the literature. We then presented a taxonomy of opportunities through which developers acquire security knowledge, and explored their motivations to software security. In addition, we presented a human-oriented model that promotes the internalization of security. This model highlights the crucial role that security knowledge opportunities and collaboration between teams play in facilitating the internalization of security.

Bibliography

[1] AngularJS Developer Guide. https://docs.angularjs.org/guide/security.

[2] BSIMM. https://www.bsimm.com. [Accessed Feb-2017].

[3] CVE - Common Vulnerability Exposures. https://cve.mitre.org. [Accessed Jan-2018].

[4] DEF CON Hacking Conference. https://www.defcon.org/index.html. [Accessed Jan-2018].

[5] FindBugs - Find Bugs in Java Programs. http://findbugs.sourceforge.net/findbugs2.html. [Accessed June-2016].

[6] Google's Approach to IT Security. https://static.googleusercontent.com/media/1.9.22.221/en//enterprise/pdf/whygoogle/google-common-security-whitepaper.pdf. [Accessed July-2016].

[7] Stack Overflow - Where Developers Learn, Share, & Build Careers. https://stackoverflow.com. [Accessed Jan-2018].

[8] Cyber security boost for UK firms. https://www.gov.uk/government/news/cyber-security-boost-for-uk-firms, 2015. [Accessed May-2017].

[9] IT Health Check (ITHC): supporting guidance. https://www.gov.uk/government/publications/it-health-check-ithc-supporting-guidance/it-health-check-ithc-supporting-guidance, 2015. [Accessed May-2017].

[10] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. Comparing the Usability of Cryptographic APIs. In *Proceedings of the 38th IEEE Symposium on Security and Privacy*, 2017.

[11] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 289–305, May 2016.

[12] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. How Internet Resources Might Be Helping You Develop Faster but Less Securely. *IEEE Security Privacy*, 15(2):50–60, March 2017.

[13] Y. Acar, S. Fahl, and M. L. Mazurek. You are Not Your Developer, Either: A Research Agenda for Usable Security and Privacy Research Beyond End Users. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 3–8, November 2016.

[14] Y. Acar, C. Stransky, D. Wermke, C. Weir, M. L. Mazurek, and S. Fahl. Developers Need Support, Too: A Survey of Security Advice for Software Developers. In *Cybersecurity Development (SecDev), 2017 IEEE*, pages 22–26. IEEE, 2017.

[15] K. Alfert and A. Fronk. Manipulation of 3-dimensional visualization of Java class relations. In *Proceedings of the 6th World Conference on Integrated Design Process Technology. Society for Design and Process Science*, 2002.

[16] K. Allendoerfer, S. Aluker, G. Panjwani, J. Proctor, D. Sturtz, M. Vukovic, and C. Chen. Adapting the cognitive walkthrough method to assess the usability of a knowledge domain visualization. In *IEEE Symposium on Information Visualization*, INFOVIS '05, pages 195–202, Oct.

[17] P. Anderson. Measuring the Value of Static-Analysis Tool Deployments. *Security Privacy, IEEE*, 10(3):40–47, May 2012.

[18] C. Anslow, S. Marshall, J. Noble, and R. Biddle. SourceVis: Collaborative software visualization for co-located environments. In *IEEE Working Conference on Software Visualization*, VISSOFT '13, pages 1–10, Sept 2013.

[19] H. Assal and S. Chiasson. Motivations and Amotivations for Software Security. In *SOUPS Workshop on Security Information Workers (WSIW)*. USENIX Association, 2018.

[20] H. Assal and S. Chiasson. Security in the Software Development Lifecycle. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, Baltimore, MD, 2018. USENIX Association.

[21] H. Assal, S. Chiasson, and R. Biddle. Cesar: Visual representation of source code vulnerabilities. In *2016 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pages 1–8, Oct 2016.

[22] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, Sept 2008.

[23] N. Ayewah and W. Pugh. The Google FindBugs Fixit. In *International Symposium on Software Testing and Analysis*, ISSTA '10, pages 241–252, New York, NY, USA, 2010.

[24] B. K. Marshall. Passwords Found in the Wild for January 2013. `http://blog.passwordresearch.com/2013/02/`. [Accessed April-2017].

[25] B. Schneier. Security Risks of Embedded Systems. https://www.schneier.com/blog/archives/2014/01/security_risks_9.html, January 2014. [Accessed May-2017].

[26] D. Baca, M. Boldt, B. Carlsson, and A. Jacobsson. A Novel Security-Enhanced Agile Software Development Process Applied in an Industrial Setting. In *2015 10th International Conference on Availability, Reliability and Security*, pages 11–19, Aug 2015.

[27] D. Baca, K. Petersen, B. Carlsson, and L. Lundberg. Static Code Analysis to Detect Software Security Vulnerabilities - Does Experience Matter? In *2009 International Conference on Availability, Reliability and Security*, pages 804–810, March 2009.

[28] A. Bacchelli and C. Bird. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 712–721, Piscataway, NJ, USA, 2013. IEEE Press.

[29] R. Balebako and L. Cranor. Improving App Privacy: Nudging App Developers to Protect User Privacy. *IEEE Security Privacy*, 12(4):55–58, July 2014.

[30] R. Balebako, A. Marsh, J. Lin, J. Hong, and L. F. Cranor. The Privacy and Security Behaviors of Smartphone App Developers. In *Workshop on Usable Security (USEC)*, 10.14722/usec.2014.23 006, 2014. Internet Society.

[31] S. Bartsch. Practitioners' Perspectives on Security in Agile Development. In *2011 Sixth International Conference on Availability, Reliability and Security*, pages 479–484, Aug 2011.

[32] G. G. Bear, J. C. Slaughter, L. S. Mantz, and E. Farley-Ripple. Rewards, praise, and punitive consequences: Relations with intrinsic and extrinsic motivation. *Teaching and Teacher Education*, 65(Complete):10–20, 2017.

[33] R. A. Becker and W. S. Cleveland. Brushing Scatterplots. *Technometrics*, 29(2):127–142, 1987.

[34] G. Berisha and J. Shiroka Pula. Defining Small and Medium Enterprises: A Critical Review. *Academic Journal of Business, Administration, Law and Social Sciences*, 1, 2015.

[35] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM*, 53(2):66–75, February 2010.

[36] A. Blackwell and T. Green. A Cognitive Dimensions questionnaire optimised for users. In *Annual Meeting of the Psychology of Programming Interest Group*, pages 137–152, 2000.

[37] A. Blackwell and T. Green. Notational systems–the cognitive dimensions of notations framework. In *HCI Models, Theories, and Frameworks: Toward an Interdisciplinary Science*. Morgan Kaufmann, 2003.

[38] CERT and CMU. Cybersecurity Engineering. `https://www.cert.org/cybersecurity-engineering/`. [Accessed Feb-2017].

[39] B. Chess and G. McGraw. Static Analysis for Security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[40] S. Chiasson, P. C. van Oorschot, and R. Biddle. Even experts deserve usable security: Design guidelines for security management systems. In *SOUPS Workshop on Usable IT Security Management (USM)*, pages 1–4, 2007.

[41] Codenomicon. The Heartbleed Bug. `http://heartbleed.com`. [Accessed June-2016].

[42] J. S. Collofello and S. N. Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *Journal of Systems and Software*, 9(3):191–195, 1989.

[43] J. M. Corbin and A. L. Strauss. *Basics of qualitative research: techniques and procedures for developing grounded theory*. Sage Publications, Inc, Los Angeles, Calif, 3rd edition, 2008.

[44] L. F. Cranor and S. Garfinkel. *Security and usability: designing secure systems that people can use*. O'Reilly Media, Inc., 2005.

[45] CTFtime. CTF? WTF? `https://ctftime.org/ctf-wtf/`. [Accessed Jan-2018].

[46] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. *Visualizing the Execution of Java Programs*, pages 151–162. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[47] E. Deci and R. M. Ryan. *Intrinsic Motivation and Self-Determination in Human Behavior*. Springer US, 1 edition, 1985.

[48] C. Z. Dib. Formal, nonformal and informal education: concepts/applicability. *AIP Conference Proceedings*, 173(1):300–315, 1988.

[49] S. Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, New York;Berlin;, 2007.

[50] J. Dietrich, V. Yakovlev, C. McCartin, G. Jenson, and M. Duchrow. Cluster Analysis of Java Dependency Graphs. In *Proceedings of the 4th ACM Symposium on Software Visualization*, SoftVis '08, pages 91–94, New York, NY, USA, 2008. ACM.

[51] D. A. Dillman. *Mail and Internet Surveys: The tailored design method*. John Wiley & Sons, Inc., 2000.

[52] S. Elo and H. Kyngäs. The Qualitative Content Analysis Process. *Journal of Advanced Nursing*, 62(1):107–115, 2008.

[53] Y. Engeström. Learning by expanding. *Center for Activity Theory and Developmental Work Research*, 1987.

[54] Y. Engeström. Expansive Learning at Work: Toward an activity theoretical reconceptualization. *Journal of Education and Work*, 14(1):133–156, 2001.

[55] Y. Engeström, R. Miettinen, and R.-L. Punamäki. *Perspectives on Activity Theory*. Cambridge University Press, 1999.

[56] M. Eraut. Non-formal learning and tacit knowledge in professional work. *British Journal of Educational Psychology*, 70(1):113–136, 2000.

[57] H. Eshach. Bridging In-school and Out-of-school Learning: Formal, Non-Formal, and Informal Education. *Journal of Science Education and Technology*, 16(2):171–190, Apr 2007.

[58] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[59] W. Fang, B. P. Miller, and J. A. Kupsch. Automated Tracing and Visualization of Software Security Structure and Properties. In *Proceedings of the Ninth International Symposium on Visualization for Cyber Security*, VizSec '12, pages 9–16, New York, NY, USA, 2012. ACM.

[60] A. Field. *Discovering statistics using IBM SPSS statistics*. SAGE Publications Ltd, 2013.

[61] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack Overflow Considered Harmful? The Impact of Copy Paste on Android Application Security. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 121–136, May 2017.

[62] J. Fonseca, M. Vieira, K. Buragga, and N. Zaman. A Survey on Secure Software Development Lifecycles. *Software Development Techniques for Constructive Information Systems Design*, pages 57–73, 2013.

[63] A. Forward and T. C. Lethbridge. A Taxonomy of Software Types to Facilitate Search and Evidence-based Software Engineering. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, pages 179–191, New York, NY, USA, 2008. ACM.

[64] M. Gagné and E. L. Deci. Self-determination theory and work motivation. *Journal of Organizational Behavior*, 26(4):331–362.

[65] S. Garfinkel and H. R. Lipford. Usable Security: History, Themes, and Challenges. *Synthesis Lectures on Information Security, Privacy, and Trust*, 5(2):1–124, 2014.

[66] D. Geer. Are Companies Actually Using Secure Development Life Cycles? *Computer*, 43(6):12–16, June 2010.

[67] B. G. Glaser and A. L. Strauss. *The discovery of grounded theory: strategies for qualitative research*. Aldine, 1967.

[68] Gene V Glass, Percy D Peckham, and James R Sanders. Consequences of failure to meet assumptions underlying the fixed effects analyses of variance and covariance. *Review of educational research*, 42(3):237–288, 1972.

[69] J. Goodall, H. Radwan, and L. Halseth. Visual Analysis of Code Security. In *IEEE Symposium on Visualization for Cyber Security*, VizSec '10, pages 46–51, New York, NY, USA, 2010.

[70] Government of Canada. SME Research and Statistics. `http://www.ic.gc.ca/eic/site/061.nsf/eng/Home`, 2018. [Accessed June-2018].

[71] M. Green and M. Smith. Developers are Not the Enemy!: The Need for Usable Security APIs. *IEEE Security Privacy*, 14(5):40–46, Sept 2016.

[72] A. Greenberg. Hackers Remotely Kill a Jeep on the Highway—With Me in It. `https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/`, 2015. [Accessed May-2017].

[73] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier. Toward Large-Scale Vulnerability Discovery Using Machine Learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 85–96, New York, NY, USA, 2016. ACM.

[74] D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel. Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses. In *2008 IEEE Symposium on Security and Privacy (SP 2008)*, pages 129–142, May 2008.

[75] M. Harbach, M. Hettig, S. Weber, and M. Smith. Using Personal Examples to Improve Risk Communication for Security & Privacy Decisions. In *Proceedings of the 32nd Annual ACM Conference on Human Factors in Computing Systems*, CHI '14, pages 2647–2656, New York, NY, USA, 2014. ACM.

[76] C. Herley and P. C. v. Oorschot. SoK: Science, Security and the Elusive Goal of Security as a Scientific Pursuit. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 99–120, May 2017.

[77] M. Howard and S. Lipner. *The security development lifecycle: SDL, a process for developing demonstrably more secure software*. Microsoft Press, Redmond, Wash, 2006.

[78] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.

[79] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *35th International Conference on Software Engineering (ICSE)*, pages 672–681, May 2013.

[80] E. E. Jones. Content analysis for the social sciences and humanities. *Psyccritiques*, 14(11):615–616, 1969.

[81] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S P'06)*, May 2006.

[82] K. R. Laughery Jr. and K. R. Laughery Sr. Human factors in software engineering: A review of the literature. *Journal of Systems and Software*, 5(1):3–14, 1985.

[83] H. F. Kaiser. A Second Generation Little Jiffy. *Psychometrika*, 35(4):401–415, Dec 1970.

[84] H. F. Kaiser and J. Rice. Little Jiffy, Mark IV. *Educational and Psychological Measurement*, 34(1):111–117, 1974.

[85] Karen Grace-Martin. Can Likert Scale Data ever be Continuous? `https://www.theanalysisfactor.com/can-likert-scale-data-ever-be-continuous/`. [Accessed Aug-2018].

[86] R. Kissel, K. M. Stine, M. A. Scholl, H. Rossman, J. Fahlsing, and J. Gulick. *Security considerations in the system development life cycle*. NIST, 2008.

[87] R. L. Kissel. NIST Interagency/Internal Report (NISTIR) - 7298rev2. `https://www.nist.gov/publications/glossary-key-information-security-terms-1`. [Accessed May-2018].

[88] K. Krippendorff. Estimating the reliability, systematic error and random error of interval data. *Educational and Psychological Measurement*, 30(1):61–70, 1970.

[89] K. Krippendorff. Testing the reliability of content analysis data. *The content analysis reader*, pages 350–357, 2009.

[90] K. Kuutti. Activity theory as a potential framework for human-computer interaction research. *Context and consciousness: Activity theory and human-computer interaction*, 1744, 1996.

[91] T. D. LaToza and B. A. Myers. On the Importance of Understanding the Strategies That Developers Use. In *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '10, pages 72–75, New York, NY, USA, 2010. ACM.

[92] L. Layman, L. Williams, and R. St. Amant. Toward Reducing Fault Fix Time: Understanding Developer Behavior for the Design of Automated Fault Detection Tools. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 176–185, Sept 2007.

[93] J. Lazar, J. H. Feng, and H. Hochheiser. *Research methods in human-computer interaction*. John Wiley, Hoboken, NJ, 2010.

[94] T. C. Lethbridge and R. Laganière. *Object-oriented software engineering: practical software development using UML and Java*. McGraw-Hill Education, 2nd edition, 2005.

[95] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead. Does bug prediction support human developers? Findings from a Google case study. In *35th International Conference on Software Engineering (ICSE)*, ICSE '13, pages 372–381, May 2013.

[96] H. Lipford, T. Thomas, B. Chu, and E. Murphy-Hill. Interactive Code Annotation for Security Vulnerability Detection. In *Proceedings of the 2014 ACM Workshop on Security Information Workers*, SIW '14, pages 17–22, New York, NY, USA, 2014. ACM.

[97] Gitta H Lubke and Bengt O Muthén. Applying multigroup confirmatory factor models for continuous outcomes to likert scale data complicates meaningful group comparisons. *Structural equation modeling*, 11(4):514–534, 2004.

[98] M. van Zadelhoff. Cybersecurity Has a Serious Talent Shortage. Here's How to Fix It. `https://hbr.org/2017/05/cybersecurity-has-a-serious-talent-shortage-heres-how-to-fix-it`. [Accessed Jan-2018].

[99] R. C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[100] G. McGraw. *Software security: building security in*. Addison-Wesley, Upper Saddle River, NJ, 2006.

[101] G. McGraw. Seven Myths of Software Security Best Practices. `http://searchsecurity.techtarget.com/opinion/McGraw-Seven-myths-of-software-security-best-practices`, 2015. [Accessed Feb-2017].

[102] S. Mckenna, D. Staheli, and M. Meyer. Unlocking user-centered design methods for building cyber security visualizations. In *IEEE Symposium on Visualization for Cyber Security*, VizSec '15, pages 1–8, Oct 2015.

[103] A. N. Meyer, L. E. Barton, G. C. Murphy, T. Zimmermann, and T. Fritz. The Work Life of Developers: Activities, Switches and Perceived Productivity. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2017.

[104] Microsoft Corp. Microsoft Security Development Lifecycle. `https://www.microsoft.com/en-us/sdl`. [Accessed June-2016].

[105] Microsoft Corp. Necessary, Explained, Actionable, and Tested (NEAT) Cards. `https://cloudblogs.microsoft.com/microsoftsecure/2012/10/09/necessary-explained-actionable-and-tested-neat-cards/`. [Accessed Aug-2018].

[106] J. Mifsud. 12 Effective Guidelines For Breadcrumb Usability and SEO. `http://usabilitygeek.com/12-effective-guidelines-for-breadcrumb-usability-and-seo/`. [Accessed June-2016].

[107] R. Millman. Nearly 1500 vulnerabilities found in automated medical equipment. `https://www.scmagazineuk.com/nearly-1500-vulnerabilities-found-in-automated-medical-equipment/article/531672/`, 2016. [Accessed Feb-2017].

[108] P. Morrison. A Security Practices Evaluation Framework. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE '15, pages 935–938, Piscataway, NJ, USA, 2015. IEEE Press.

[109] S. Müller, M. Würsch, T. Fritz, and H. C. Gall. An Approach for Collaborative Code Reviews Using Multi-touch Technology. In *International Workshop on Co-operative and Human Aspects of Software Engineering*, CHASE '12, pages 93–99, Piscataway, NJ, USA, 2012.

[110] T. Nafees, N. Coull, I. Ferguson, and A. Sampson. Vulnerability Anti-Pattern: A Timeless Way to Capture Poor Software Practices (Vulnerabilities). In *Pattern Languages of Programs Conference*, 2017.

[111] T. Nafees, N. Coull, R. I. Ferguson, and A. Sampson. Idea-Caution Before Exploitation: The Use of Cybersecurity Domain Knowledge to Educate Software Engineers Against Software Vulnerabilities. In E. Bodden, M. Payer, and E. Athanasopoulos, editors, *Engineering Secure Software and Systems*, pages 133–142, Cham, 2017. Springer International Publishing.

[112] B. A. Nardi. Studying context: A comparison of activity theory, situated action models, and distributed cognition. *Context and consciousness: Activity theory and human-computer interaction*, 69102, 1996.

[113] NASA. Software Assurance Guidebook, NASA-GB-A201. https://www.hq.nasa.gov/office/codeq/doctree/nasa_gb_a201.pdf, 2002. [Accessed Feb-2017].

[114] National Vulnerabiliy Database. NVD statistics results. https://web.nvd.nist.gov/view/vuln/statistics. [Accessed March-2017].

[115] D. C. Nguyen, D. Wermke, Y. Acar, M. Backes, C. Weir, and S. Fahl. A Stitch in Time: Supporting Android Developers in WritingSecure Code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1065–1077, New York, NY, USA, 2017. ACM.

[116] NIST. National Vulnerability Database. https://nvd.nist.gov. [Accessed March-2017].

[117] K. O'Connor. *Activity Theory*. American Cancer Society, 2015.

[118] V. Okun, A. Delaitre, and P. E. Black. Report on the Static Analysis Tool Exposition (SATE) IV. In *NIST Special Publication 500-297*. 2013.

[119] D. Oliveira, M. Rosenthal, N. Morin, K.-C. Yeh, J. Cappos, and Y. Zhuang. It's the Psychology Stupid: How Heuristics Explain Software Vulnerabilities and How Priming Can Illuminate Developer's Blind Spots. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 296–305, New York, NY, USA, 2014. ACM.

[120] Organisation for Economic Co-operation and Development (OECD). Recognition of Non-formal and Informal Learning - Home. http://www.oecd.org/edu/skills-beyond-school/recognitionofnon-formalandinformallearning-home.htm. [Accessed Jan-2018].

[121] OWASP. CLASP. https://www.owasp.org/index.php/CLASP. [Accessed Feb-2017].

[122] OWASP. OWASP CTF Project. https://www.owasp.org/index.php/Category:OWASP_CTF_Project. [Accessed Jan-2018].

[123] OWASP. OWASP Guide Project. https://www.owasp.org/index.php/Category:OWASP_Guide_Project. [Accessed Feb-2017].

[124] OWASP. OWASP SAMM Project. https://www.owasp.org/index.php/OWASP_SAMM_Project. [Accessed Feb-2017].

[125] OWASP. OWASP Testing Project. https://www.owasp.org/index.php/OWASP_Testing_Project. [Accessed Feb-2017].

[126] OWASP. OWASP Secure Coding Practices Checklist. https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_Checklist, January 2016. [Accessed June-2018].

[127] OWASP. Static Code Analysis. https://www.owasp.org/index.php/Static_Code_Analysis, 2017. [Accessed May-2017].

[128] T. Panas, Rüdiger Lincke, and Welf Löwe. Online-configuration of Software Visualizations with Vizz3D. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, pages 173–182, New York, NY, USA, 2005. ACM.

[129] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol. Would static analysis tools help developers with code reviews? In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 161–170, March 2015.

[130] B. D. Payne and W. K. Edwards. A Brief Introduction to Usable Security. *IEEE Internet Computing*, 12(3):13–21, May 2008.

[131] H. Perl, S. Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 426–437, New York, NY, USA, 2015. ACM.

[132] O. Pieczul, S. Foley, and M. E. Zurko. Developer-centered Security and the Symmetry of Ignorance. In *Proceedings of the 2017 New Security Paradigms Workshop*, NSPW 2017, pages 46–56, New York, NY, USA, 2017. ACM.

[133] A. Poller, L. Kocksch, S. Türpe, F. A. Epp, and K. Kinder-Kurlanda. Can Security Become a Routine?: A Study of Organizational Change in an Agile Software Development Group. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*, CSCW '17, pages 2489–2503, New York, NY, USA, 2017. ACM.

[134] P. G. Polson, C. Lewis, J. Rieman, and C. Wharton. Cognitive walkthroughs: a method for theory-based evaluation of user interfaces. *International Journal of Man-Machine Studies*, 36(5):741–773, 1992.

[135] J. Radcliffe. Hacking Medical Devices for Fun and Insulin: Breaking the Human SCADA System. `https://media.blackhat.com/bh-us-11/Radcliffe/BH_US_11_Radcliffe_Hacking_Medical_Devices_WP.pdf`, 2011. [Accessed Feb-2017].

[136] Rapid 7 Community. #IoTsec Disclosure: 10 New Vulnerabilities for Several Video Baby Monitors. `https://community.rapid7.com/community/infosec/blog/2015/09/02/iotsec-disclosure-10-new-vulns-for-several-video-baby-monitors`, 2015. [Accessed Feb-2017].

[137] J. Ratner. *Human factors and Web development*. CRC Press, 2003.

[138] S. P. Reiss. Dyvise: Performance analysis of production systems. In *In Proceedings of the International Conference on Software Engineering (ICSE)(2009), IEEE Computer*, 2009.

[139] H.-S. Rhee, Y. U. Ryu, and C.-T. Kim. Unrealistic optimism on information security management. *Computers & Security*, 31(2):221–232, 2012.

[140] N. B. Ruparelia. Software development lifecycle models. *SIGSOFT Software Engineering Notes*, 35(3):8–13, May 2010.

[141] R. M. Ryan and E. L. Deci. Self-determination theory and the facilitation of intrinsic motivation, social development, and well-being. *American Psychologist*, 55(1):68, 2000.

[142] R. M. Ryan and E. L. Deci. *Self-determination theory: Basic psychological needs in motivation, development, and wellness*. Guilford Publications, 2017.

[143] S. Streichsbier. Improve Web Application Security with Frameworks: A case study. `http://www.vantagepoint.sg/blog/18-improve-web-application-security-with-frameworks-a-case-study`. [Accessed Feb-2017].

[144] R. Sass. How to Balance Between Security and Agile Development the Right Way. `https://resources.whitesourcesoftware.com/blog-whitesource/how-to-balance-between-security-and-agile-development-the-right-way`, 2016. [Accessed May-2018].

[145] R. Seacord. Top 10 secure coding practices. `https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices`, 2011. [Accessed Feb-2017].

[146] M. Selart, T. Nordström, B. Kuvaas, and K. Takemura. Effects of Reward on Selfregulation, Intrinsic Motivation and Creativity. *Scandinavian Journal of Educational Research*, 52(5):439–458, 2008.

[147] G. Sevitsky, W. De Pauw, and R. Konuru. An information exploration tool for performance analysis of Java programs. In *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 38*, pages 85–101, 2001.

[148] B. Shneiderman. Tree Visualization with Tree-maps: 2-D Space-filling Approach. *ACM Transactions on Graphics (TOG)*, 11(1):92–99, January 1992.

[149] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford. Questions Developers Ask While Diagnosing Potential Security Vulnerabilities with Static Analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 248–259, New York, NY, USA, 2015. ACM.

[150] J. Smith, B. Johnson, E. Murphy-Hill, B. T. Chu, and H. Richter. How Developers Diagnose Potential Security Vulnerabilities with a Static Analysis Tool. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[151] J. Snow. *On the mode of communication of cholera.* John Churchill, 1855.

[152] I. Sommerville. *Software engineering.* Pearson, Boston, 9th edition, 2011.

[153] J. Sophy. 43 Percent of Cyber Attacks Target Small Business. `https://smallbiztrends.com/2016/04/cyber-attacks-target-small-business.html`, 2016. [Accessed Feb-2017].

[154] G. Stahl. Conceptualizing the Intersubjective Group. *International Journal of Computer-Supported Collaborative Learning*, 10(3):209–217, Sep 2015.

[155] M. Stanislav. R7-2015-27 and R7-2015-24: Fisher-Price Smart Toy & hereO GPS Platform Vulnerabilities (FIXED). `https://community.rapid7.com/community/infosec/blog/2016/02/02/security-vulnerabilities-within-fisher-price-smart-toy-hereo-gps-platform`, 2016. [Accessed Feb-2017].

[156] J. P. Stevens. *Applied multivariate statistics for the social sciences.* New Jersey: Lawrance Erlbaum Association, 2002.

[157] G. Stoneburner, A. Goguen, and A. Feringa. Risk Management Guide for Information Technology Systems. `https://csrc.nist.gov/publications/detail/sp/800-30/archive/2002-07-01`. [Accessed May-2018].

[158] C. Stransky, Y. Acar, D. C. Nguyen, D. Wermke, D. Kim, E. M. Redmiles, M. Backes, S. Garfinkel, M. L. Mazurek, and S. Fahl. Lessons Learned from Using an Online Platform to Conduct Large-Scale, Online Controlled Security Experiments with Software Developers. In *10th USENIX Workshop on Cyber Security Experimentation and Test (CSET 17)*, Vancouver, BC, 2017. USENIX Association.

[159] A. L. Strauss and J. M. Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory.* Sage Publications, Inc., 2 edition, 1998.

[160] Symantec Security Response. ShellShock: All you need to know about the Bash Bug vulnerability. `http://www.symantec.com/connect/blogs/shellshock-all-you-need-know-about-bash-bug-vulnerability`, 2014. [Accessed June-2016].

[161] T. D. Harmon. Cyber Security Capture The Flag (CTF): What Is It? `https://blogs.cisco.com/perspectives/cyber-security-capture-the-flag-ctf-what-is-it`. [Accessed Jan-2018].

[162] A. Telea, H. Hoogendorp, O. Ersoy, and D. Reniers. Extraction and visualization of call dependencies for large C/C++ code bases: A comparative study. In *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 81–88, Sept 2009.

[163] T. Thomas, B. Chu, H. Lipford, J. Smith, and E. Murphy-Hill. A study of interactive code annotation for access control vulnerabilities. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 73–77, Oct 2015.

[164] T. W. Thomas, H. Lipford, B. Chu, J. Smith, and E. Murphy-Hill. What questions remain? an examination of how developers understand an interactive static analysis tool. In *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*, Denver, CO, 2016. USENIX Association.

[165] T. W. Thomas, M. Tabassum, B. Chu, and H. Lipford. Security During Application Development: An Application Security Expert Perspective. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 262:1–262:12, New York, NY, USA, 2018. ACM.

[166] I. A. Tondel, M. G. Jaatun, and P. H. Meland. Security Requirements for the Rest of Us: A Survey. *IEEE Software*, 25(1):20–27, Jan 2008.

[167] M. A. Tremblay, C. M. Blanchard, S. Taylor, L. G. Pelletier, and M. Villeneuve. Work Extrinsic and Intrinsic Motivation Scale: Its value for organizational psychology research. *Canadian Journal of Behavioural Science/Revue canadienne des sciences du comportement*, 41(4):213, 2009.

[168] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin. ALETHEIA: Improving the Usability of Static Security Analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 762–774, New York, NY, USA, 2014. ACM.

[169] S. Türpe. Idea: Usable Platforms for Secure Programming – Mining Unix for Insight and Guidelines. In J. Caballero, E. Bodden, and E. Athanasopoulos, editors, *Engineering Secure Software and Systems*, pages 207–215, Cham, 2016. Springer International Publishing.

[170] R. J. Vallerand and R. Blssonnette. Intrinsic, Extrinsic, and Amotivational Styles as Predictors of Behavior: A Prospective Study. *Journal of Personality*, 60(3):599–620.

[171] N. D. Weinstein and W. M. Klein. Unrealistic Optimism: Present and Future. *Journal of Social and Clinical Psychology*, 15(1):1–8, 2017/08/12 1996.

[172] C. Weir, A. Rashid, and J. Noble. I'd Like to Have an Argument, Please: Using Dialectic for Effective App Security. *The 2nd European Workshop on Usable Security (EuroUSEC 2017)*, 2017.

[173] C. Wharton, J. Rieman, C. Lewis, and P. Polson. Usability Inspection Methods. chapter The Cognitive Walkthrough Method: A Practitioner's Guide, pages 105–140. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[174] A. Whitten and J. D. Tygar. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. In *USENIX Security Symposium*, volume 348, 1999.

[175] J. Witschey, S. Xiao, and E. Murphy-Hill. Technical and Personal Factors Influencing Developers' Adoption of Security Tools. In *Proceedings of the 2014 ACM Workshop on Security Information Workers*, SIW '14, pages 23–26. ACM, 2014.

[176] C. Woody. Strengthening Ties Between Process and Security. `https://www.us-cert.gov/bsi/articles/knowledge/sdlc-process/strengthening-ties-between-process-and-security#touch`, 2013. [Accessed Feb-2017].

[177] I. M.Y. Woon and A. Kankanhalli. Investigation of IS professionals' intention to practise secure development of applications. *International Journal of Human-Computer Studies*, 65(1):29–41, 2007.

[178] G. Wurster and P. C. van Oorschot. The Developer is the Enemy. In *Proceedings of the 2008 New Security Paradigms Workshop*, NSPW '08, pages 89–97, New York, NY, USA, 2008. ACM.

[179] S. Xiao, J. Witschey, and E. Murphy-Hill. Social Influences on Secure Development Tool Adoption: Why Security Tools Spread. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '14, pages 1095–1106, New York, NY, USA, 2014. ACM.

[180] J. Xie, B. Chu, H. R. Lipford, and J. T. Melton. ASIDE: IDE Support for Web Application Security. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 267–276, New York, NY, USA, 2011. ACM.

[181] J. Xie, H. Lipford, and B.-T. Chu. Evaluating Interactive Support for Secure Programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 2707–2716, New York, NY, USA, 2012. ACM.

[182] J. Xie, H. R. Lipford, and B. Chu. Why do programmers make security errors? In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 161–164, Sept 2011.

[183] F. Yamaguchi, F. Lindner, and K. Rieck. Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.

[184] Y. Ye and K. Kishida. Toward an Understanding of the Motivation Open Source Software Developers. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 419–429, Washington, DC, USA, 2003. IEEE Computer Society.

[185] H. Zhong and Z. Su. An Empirical Study on Real Bug Fixes. In *International Conference on Software Engineering*, ICSE '15, pages 913–923, Piscataway, NJ, USA, 2015.

Appendix A

Interview Script

The following questions represent the main themes discussed during the interviews. We may have probed for more details depending on participants' responses.

- What type of development do you do?
- What are your main priorities when doing development? (In order of priority)
- Do your priorities change when a deadline approaches?
- What about security? Is it something you worry about?
- Which are the best methods in your opinion for ensuring the security of software applications?
- How does security fit in your priorities?
- Which resources do you use to gain security knowledge?
- Do you get training (formal, or self-learning) to gain better knowledge of software security? How often?
- Which software security best practices are you familiar with?
- Are there any obligations by your supervisor/employer for performing security testing?
- What methods do you use to try to ensure the security of applications?
- Do you perform testing on your (or someone else's) applications/code?
- Do you perform code reviews?

- How would you describe the relation between the development and the testing team?
- Can you think of a story of security issue that was frustrating and how you dealt with it?

Appendix B

Motivations and Amotivations for Software Security

Table B.1: Motivations and Amotivations of software security

Code	Description	Example Quote
		Amotivation - Felt lack of competence
Lack of resources	The shortage in resources, *e.g.*, budget and human power, needed to perform security tasks	"*We don't have that much manpower to explicitly test security vulnerabilities, [..] we don't have those kind of resources. But ideally if we did have [a big] company size, I would have a team dedicated to find exploits, um, that sorta thing. But unfortunately we don't.*"
Lack of support	The inadequate security tools and processes, or the lack thereof	"*We don't have any formal process of like a code review, sitting down and talking about security risks*"
		Amotivation - Lack of interest, relevance, value
Not my responsibility	Security is not part of my duties	"*Developers are similar to me, they don't care that much about security or it's not part of their day to day job, therefore they don't pay much attention to the security aspect of the code.*"
Security is handled elsewhere	Security is another entity's responsibility	"*I usually don't as a developer go to the extreme of testing vulnerability in my feature, that's someone else's to do.*"

continued ...

... continued

Code	Description	Example Quote
Induced passiveness	The surrounding environment causes passiveness towards security	"*I don't really trust them [my team members] to run any kind of like source code scanners or anything like that. I know I'm certainly not going to.*"
No perceived loss	The lack of competition, expected repercussions, and loss	"*I can introduce a big security issue and I definitely won't be blamed that much for it*"
No perceived risk	The company or application type is perceived as not a valuable target for attacks	"*For a small company, nobody will usually attack or compromise the vulnerabilities in your system. If something really bad happens, usually, you don't really get enough [bad] reputation as well.*"
Competing priorities	Other tasks compete for resources and are prioritized over security	"*I have security issues that are frustrating, but I haven't been able to deal with them yet. [...] It's not something that we've been able to deal with yet, just cause of priorities with everything else.*"

continued ...

... continued

Code	Description	Example Quote
	Amotivation - Defiance/Resistance to influence	
Inflexibility	The resistance to new technology and being set in one's way	"*[My team is] using a framework and these guys, they used the framework incorrectly, they didn't like how certain part of this coding framework works and has been designed, so they decided to do things completely different than it [...] And I am sure it's gonna result in a security risk down the line.*"
	Extrinsic Motivation - External	
Audit fear	The presence of an overseeing and supervising entity	"*One of the main reasons that they did [address security] was audits. I think they had to comply with certain security regulation standard, basically every quarter or so they're being checked for compliance, therefore they had the make sure the auditors can't find any issue during the penetration test.*"
Business loss	Losses that a business can incur, *e.g.*, losing customers, due to security issues	"*We ended up ignoring security until we got a decent customer base where we were actually concerned that if our product was compromised, we will lose these customers.*"
Pressure	Continuous pressure by superiors	"*If they find a security issue, then you will be in trouble. Everybody will be at your back, and you have to fix it as soon as possible.*"

continued ...

... continued

Code	Description	Example Quote
Career advancement	Software security efforts and knowledge move employees up in the hierarchy	"*When it comes time to do promotions or move throughout the scales and employment bands, the people with the higher knowledge on everything move up and the people who don't necessarily, like, didn't take those security training seriously, [...] they sort of stay in the same range.*"
Extrinsic Motivation - Introjected		
Prestige	Acknowledgement and preserving self-image	"*Whenever somebody wants to find about you, then they go and check you in the employee website. Then, when they click your name and check, it shows a badge that you're security certified, which gives you a good feeling.*"
Extrinsic Motivation - Identified		
Understanding the implications	Recognizing and understanding the potential implications of ignoring security	"*Just understanding the implications, I guess, of what could happen [would motivate developers be more security-oriented]. I know for me personally when I realized just how catastrophic something could be, just by making a simple mistake, or not even a simple mistake, just overlooking something simple. uhh it changes your focus.*"

continued ...

… continued

Code	Description	Example Quote
Company reputation	The company and its employees care about their reputation and how customers perceive the company	"*We need to know safe secure coding techniques, we need to know what paths the attackers might take, and have you fixed everything on your code and your code doesn't have any vulnerabilities. […] because finally, it is going to go under your logo.*"
Shared responsibility	The responsibility of software security is shared among different teams within the project team	"*[If we find a vulnerability,] we try not to say, 'you personally are responsible for causing this vulnerability'. I mean, it's a team effort, people looked at that code and they passed on it too, then it's shared, really.*"
Induced initiative	Opportunities may exist that lead developers to take the software security initiative	"*When you see your colleagues actually spending time on something, you might think that 'well, it's something that's worth spending time on', but if you worked in a company that nobody just touches security then you might not be motivated that much.*"
Extrinsic Motivation - Integrated		
Professional responsibility	Feeling responsible as a professional	"*I would hesitate to release anything that's not functional and I also hesitate to release anything that had security concerns.*"

continued …

... continued

Code	Description	Example Quote
Concern for users	Caring about users' privacy and security	"*I would not feel comfortable with basically having something used by end users that I didn't feel was secure, or I didn't feel respective of privacy, umm so I would try very hard to not compromise on that.*"
	Intrinsic Motivation	
Self-improvement	The interest in, and self-satisfaction from, improving one's implementation	"*And sometimes I will have the challenge, that 'okay, this time I'm going to submit for a review where nobody will give me a comment', though that never happened, but still...*"

Appendix C

Developers' Survey

Q1 Where are you currently employed?

- ○ Africa
- ○ Asia
- ○ Canada
- ○ Europe
- ○ Latin America and the Carribean
- ○ Oceania (Australia, New Zealand, Melanesia, Micronesia, Polynesia)
- ○ USA

Q2 Please select the statement that best describes your current work.

- ○ I am currently on leave
- ○ I am self-employed
- ○ I am currently employed in design (e.g., UI designer, interaction designer)
- ○ I am currently employed in developing software (e.g., programmer, developer, web developer, software engineer, etc.)
- ○ I am currently employed in testing software (e.g., tester, quality analyst, automation engineer, etc.)
- ○ None of the above

Q3 Please select your gender.

- ○ Male
- ○ Female
- ○ Other or not specified

Q4 How long have you been working: [*textbox and a single-choice menu containing "days, weeks, months, years"*]

- in your current company?
- in your current team
- in general as a professional developer?

Q5 What is your job title?

Q6 Where did you *mainly* learn to program and develop software?

- ○ Self-taught
- ○ High school courses
- ○ College courses
- ○ University courses
- ○ Online courses
- ○ Industry or on-the-job training
- ○ Other. Please specify.

Q7 Please select the primary development process used by your team.

- ○ Waterfall development (aka Traditional)
- ○ Iterative (but not truly agile), such as Spiral
- ○ Rational Unified Process (RUP)
- ○ Agile development (including: Scrum, Dynamic Systems Development Model (DSDM), Crystal Methods, Extreme programming (XP), Rapid Application Development (RAD), Feature Driven Development (FDD))
- ○ Other. Please specify

Q8 Does your project team perform Test-Driven Development (TDD)?

- ○ Yes
- ○ No
- ○ I don't know

Q9 Please describe the types of software you develop and work on.

Q10 Select the most appropriate category that best describes the software you work on.

- ○ Games

- ○ Information display and transaction entry, including websites
- ○ Tools for artistic creativity
- ○ Other consumer-oriented software, including productivity software
- ○ Transaction processing systems for business
- ○ Other Business-oriented software, including management information
- ○ Scientific software, including analysis and visualization
- ○ Computational intensive software such as audio and video processing, machine learning
- ○ Design and engineering software, including CAD-CAM
- ○ Networking and communications software, including telecom and wireless
- ○ Operating systems and their support utilities
- ○ Software for vehicles, aerospace and robots
- ○ Other real-time control and embedded or systems software for devices
- ○ Middleware, system components, libraries and frameworks
- ○ Other tools for software developers, such as IDEs and compilers
- ○ Other. Please specify.

Q11 How old is your organization? [*textbox and a single-choice menu containing "days, weeks, months, years"*]

Q12 What is the total number of employees in your organization?

- ○ 1 to 9
- ○ 10 to 249
- ○ 250 to 499
- ○ 500 to 999
- ○ 1,000 or more

Q13 How many members are there in your team? Please enter numbers only.

Q14 On a scale from 5 (corresponds exactly) to 1 (does not correspond at all), please indicate to what extent each of the following items corresponds to the reasons why you are presently involved in your work. **Why do you do what you do?**

- Because this is the type of work I chose to do to attain a certain lifestyle.

- For the income it provides me.
- I ask myself this question, I don't seem to be able to manage the important tasks related to this work.
- Because I derive much pleasure from learning new things.
- Because it has become a fundamental part of who I am.
- Because I want to succeed at this job, if not I would be very ashamed of myself.
- Because I chose this type of work to attain my career goals.
- For the satisfaction I experience from taking on interesting challenges.
- Because it allows me to earn money.
- Because it is part of the way in which I have chosen to live my life.
- Because I want to be very good at this work, otherwise I would be very disappointed.
- I don't know why, we are provided with unrealistic working conditions.
- Because I want to be a "winner" in life.
- Because it is the type of work I have chosen to attain certain important objectives.
- For the satisfaction I experience when I am successful at doing difficult tasks.
- Because this type of work provides me with security.
- I don't know, too much is expected of us.
- Because this job is a part of my life.

Q15 What does it mean to include security into the development process?

Q16 For the rest of the survey, when we mention "security", we refer to software security as described below. Please note that we are **not** asking about other aspects of security, such as infrastructure and IT security (*e.g.*, ensuring all users in the organization always have software patches installed, and use secure passwords on their accounts)

Software security

- Software security is the idea of building an application that is resistant to:

malicious attacks, being used by unauthorized people, or causing harm by inappropriate possibly-accidental use.

- Software security aims to minimize vulnerabilities that could be exploited by attackers (e.g., eliminating buffer overflow vulnerabilities)
- The use of static analysis tools to find potential vulnerabilities in the software being built is an example of software security.

Security functions

- Security functions are the application's security features to protect resources, *e.g.*, authentication to protect user data.
- They can be implemented as functionality within an application (*e.g.*, user authentication).
- Verifying usernames and passwords is an example of security functions.

Which of the following aims to reduce malicious attacks that exploit vulnerabilities? Please select the most accurate choice based on the description above.

- ○ User authentication
- ○ Software security
- ○ Security functions
- ○ All of the above

Q17 ***(RQ1)*** Please select the statement that best describes your team. [*4-point Likert scale: strongly agree-strongly disagree.*]

- My team believes that software security is important
- We have specific procedures in place to address software security
- We do not think our applications/features are interesting targets for attackers
- We haven't really considered the security of our software/applications/features

Q18 ***(RQ1)*** As a percentage, how much of your team's overall effort in the development lifecycle relates specifically to security tasks?

Q19 ***(RQ1)*** How are your project team's total software security efforts divided among the following stages? [*Text boxes, total must equal 100*]

- The design stage
- While implementing the code
- During testing by developers
- During code analysis (e.g., using static analysis tools)
- During code review
- During testing that is done by someone other than the code owner

Q20 ***(RQ1)*** Rate your agreement with the following. [*5-point Likert scale: strongly agree-strongly disagree.*]

- In my team, when we're choosing a framework/API, we consider whether it gives us security advantages
- I am satisfied with how my team is handling software security

Q21 ***(RQ1)*** How likely do you think it is that features developed by your team contain security issues? [*5-point Likert scale: extremely likely-extremely unlikely.*]

Q22 ***(RQ1)*** Has your company ever experienced a security issue with software it has developed (e.g., discovering a security vulnerability, or experiencing a security breach)?

☐ Yes, we experienced a security breach
☐ Yes, a vulnerability in shipped code was discovered
☐ Yes, a vulnerability in un-shipped code was discovered
☐ No
☐ I don't know or prefer not to answer

Q23 ***(RQ1)*** How did experiencing a security issue change the attitude towards security over the long term for each of the following? [*single choice: It led to more awareness and concern for security, It didn't lead to any change, It lead to less care and awareness for security, I don't know/I prefer not to answer*]

- You
- Other developers
- Team leaders
- Higher management

- Users/Customers

Q24 ***(RQ2)*** Rate your agreement with the following statements. I care about security because... [*5-point Likert scale: strongly agree-strongly disagree, and 'not applicable' choice*]

[M1] My company is audited for software security by an external entity
[M2] My company would lose customers in case of a software security breach
[M3] My company could fail (cease to operate) in case of a software security breach
[M4] My efforts towards software security are recognized
[M5] My efforts towards software security help me grow in the company
[M6] My efforts towards software security are financially rewarding (*e.g.*, through bonuses or a raise)
[M7] My company mandates security practices and I have to follow them
[M8] I see the benefit in security practices mandated by my company
[M9] I understand that my code can have security implications
[M10] My colleagues care about software security
[M11] I care about my company's reputation
[M12] I care about my users' security and privacy
[M13] Software security is in my company's culture
[M14] Software security is a shared responsibility by all those involved in the development lifecycle
[M15] I see software security as my responsibility
[M16] I feel good when I learn about software security
[M17] I feel good when I address potential security issues in my code
[M18] I like to challenge myself to write secure code
[M19] Similar software to that on which I work suffered a security breach and management now cares about securing our applications
[M20] Similar software to that on which I work suffered a security breach and it was an eye-opener for me
[M21] I realized securing my code is important after reading about security breaches in the news

Q25 ***(RQ2)*** Rate your agreement with each of the following statements. [*5-point Likert scale: strongly agree-strongly disagree*]

- [D1] Software security is not my responsibility because it's not in my job description
- [D2] Software security does not fit in my schedule
- [D3] Software security is a burden on top of my main responsibilities
- [D4] Software security is not mandated by my employer
- [D5] Software security is handled by someone else in the product lifecycle
- [D6] We don't have to worry much about security because frameworks (including APIs)/programming language in-house tools we use handle software security for us
- [D7] My team doesn't spend any specific efforts towards software security
- [D8] We defer software security due to competing priorities
- [D9] In my team, it is more important to deliver features on time than to address software security
- [D10] If we focus more on software security, we might lose our business opportunities
- [D11] There are no repercussions to ignoring software security
- [D12] We do not have competition, so we won't lose customers in case of a software security issue
- [D13] I won't be blamed if a security issue is found in my code
- [D14] It's unlikely that attackers will attack us
- [D15] The software I develop is not prone to security attacks
- [D16] Things are fine as they are, we haven't experienced any security breaches
- [D17] No one else cares about software security, I won't either
- [D18] I understand the importance of addressing security, but I won't waste my time on it since no one else does
- [D19] I used to push for software security, but I was perceived negatively by my colleagues
- [D20] We do not have a formal process for software security
- [D21] Available security code analysis tools are not useful

- [D22] I am not aware of tools that would allow security analysis of my code
- [D23] I do not have time to address software security
- [D24] I do not have necessary knowledge to address software security
- [D25] There aren't enough people in my team to address software security
- [D26] My team does not have the budget to address software security
- [D27] We're doing fine, I don't think we should change in terms of software security
- [D28] We have been following the same procedures for years and I don't want to change them
- [D29] I tend to resist when I get assigned a security task

Q26 ***(RQ1)*** Rate your agreement with each of the following statements. [*5-point Likert scale: strongly agree-strongly disagree, and 'not applicable' choice*]

- [S1] We rely on libraries and frameworks (including APIs) to help guarantee software security
- [S2] Our company/team has baseline security standards with which 3rd party code should comply
- [S3] We built our own in-house frameworks to help guarantee software security
- [S4] I can get deadline extensions to handle software security
- [S5] When a deadline approaches, I try to reduce my workload to focus on securing my software
- [S6] I have my own mental checklist of software security issues that I need to consider in my code
- [S7] I have come up with my own software security best practices
- [S8] If I didn't have time to address software security, I'd ship the product after adding a work around that allows me to remotely disable the software feature suffering a security breach
- [S9] When working on a software security issue, I can get help from others who worked on similar issues
- [S10] I prefer to ask for software security advice informally (*e.g.*, by casually asking a colleague, or through discussions over lunch)
- [S11] I can rely on the more experienced members of my company/team for help

and security advice

[S12] Software security best practices are incorporated in automated checks we run

[S13] Software security best practices are incorporated in tools we use

[S14] We have a document/checklist of items that we need to consider for our application to be secure

[S15] I receive specific instructions on how to solve security issues found in my code

[S16] In code reviews, reviewers explain security issues and fixes to me rather than referring me to resources/books

Appendix D

Types of Software Developed by Survey Participants

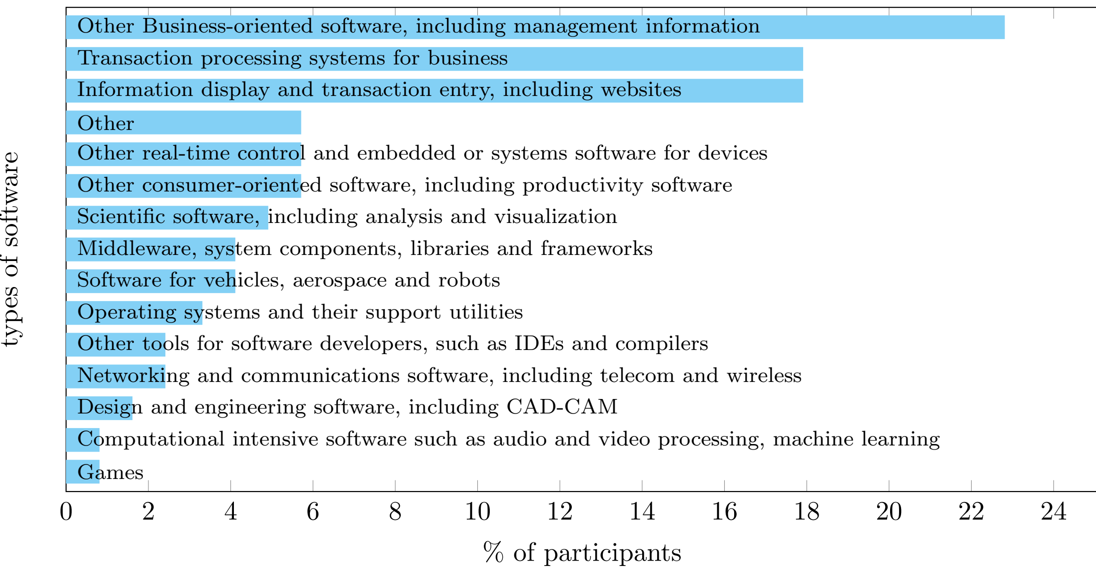


Figure D.1: Types of software developed by survey participants. The classification of software types was adapted from [63]