ABSTRACT

Title of Dissertation: Understanding and Improving Secure Development from a Human-Centered Perspective

Kelsey R. Fulton
Doctor of Philosophy, 2023

Dissertation Directed by: Dr. Michelle L. Mazurek
Department of Computer Science

Secure software development remains a difficult task, as is exemplified by the fact that vulnerabilities are discovered in production code on a regular basis. Researchers in the computer security field have worked for many years to mitigate this problem through building better security tooling, creating secure programming languages, improving secure development processes, and improving educational interventions. The success of these interventions depends on both the technical attributes of the intervention and the human and organizational factors that impact adoption, usability, and efficacy, suggesting the importance of understanding both the technical and human and organizational factors that influence the success of these interventions. While there has been much past work exploring the technical factors, there has been little work exploring the human and organizational factors.

To attempt to close this gap, I start by exploring why and how developers introduce, find, and fix vulnerabilities as they build secure code. By performing in-depth qualitative analysis on

data collected throughout an iteration of a secure programming competition, I uncover that teams with a detailed initial design tended to introduce fewer vulnerabilities than those without, and teams that worked on security steadily throughout building their codebase had fewer vulnerabilities. I also uncover that different types of vulnerabilities are discovered and fixed differently. Vulnerabilities that arose from simple programming mistakes tended to be found incidentally, although they were frequently found by testing for almost anything, related or not. They were often found by teams themselves, during their own build-phase testing (before attacks from other teams), and were typically easy to fix. In contrast, vulnerabilities related to misunderstanding security properties required deeper knowledge and more focused testing and were rarely found by build teams until they were exploited by other teams.

Next, I explore the adoption of current security development interventions by understanding the benefits and drawbacks of adopting a secure programming language by using Rust as a case study. Through the use of interviews with professional developers that had adopted or attempted to adopt Rust and a survey with the broader Rust community, I highlighted a range of positive features, including good tooling and documentation, benefits for the development lifecycle, and improvement of overall secure coding skills, as well as drawbacks including a steep learning curve, limited library support, and concerns about the ability to hire additional Rust developers in the future. These results have implications for promoting the adoption of Rust specifically and secure programming languages and tools more generally.

Lastly, given the importance of understanding the human and organizational factors of secure software development, I explore alternate approaches to conducting these studies to improve validity and reduce stress on participants. Through a lab study, I explore the efficacy of tasking participants with reading code and identifying vulnerabilities or reading code and fixing vulnerabilities

in code instead of tasking them with writing secure code from scratch. I find parallels in the functionality and security of results among the three conditions with key similarities in the types of vulnerabilities that developers introduce or fail to identify. Further, I demonstrate that participants in the read condition and the fix condition feel less negative effects such as frustration and time spent participating in the study. Our results suggest the possibility for using these methods as alternatives to writing code and avenues for future exploration.

UNDERSTANDING AND IMPROVING SECURE DEVELOPMENT FROM A HUMAN-CENTERED PERSPECTIVE

by

Kelsey R. Fulton

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2023

Advisory Committee:
Associate Professor Michelle L. Mazurek, Chair/Advisor
Associate Professor John Dickerson
Professor Michael Hicks
Professor Wayne Lutters
Assistant Professor Bradley Reaves

Acknowledgments

I owe my deepest gratitude to everyone who made this dissertation and PhD possible. Many people have contributed to my success, without whom I would not have completed this work.

First and foremost, I would like to thank my advisor, Dr. Michelle Mazurek. She has truly been exemplary advisor and constant source of career, academic, and emotional support. I truly count her as a friend and one of the best mentors I had and will ever have. She has been the model on which I base my own approach as a mentor and advisor.

I also owe a huge debt to my partner, Adam. His constant support and encouragement through good times and bad made this thesis possible. I want to thank my parents for convincing me that I could achieve anything in life that I wanted. I would not be where I am now without their encouragement and support. I also want to recognize my rats Bean, Roxy, Tater, Turnip, and Sprout. They were a source of smiles and comfort through the most stressful parts of my PhD.

I would like to thank Dr. Michael Hicks for his support and guidance throughout my graduate school career. I would also like to thank Dr. John Dickerson, Dr. Bradley Reaves, and Dr. Wayne Lutters for serving on my dissertation committee and providing me with valuable feedback on my dissertation.

My labmates in the Security, Privacy, and People lab (SP2) have truly made my grauduate experience one I will never forget. Specifically, I want to thank Dan Votipka, Joe Lewis, Nathan Malkin, Omer Akgul, and Noel Warford for all the support they have provided me and the time

they spent discussing ideas and conducting this research. The success of this research would not have been possible without them.

I would also like to thank the MC2, CS department, and IRB staff. Dana Purcell and Tom Hurst provided unending support for every administrative issue and were quick to answer any questions or address any concerns that I had. Joseph Smith and Andrea Dragan were quick to answer any IRB questions I had.

Finally, I would like to acknowledge funding support from Accenture, AT&T, Galois, Leidos, Patriot Technologies, NCC Group, Trail of Bits, Synposis, ASTech Consulting, Cigital, SuprTek, Cyberpoint, and Lockheed Martin; by NSF grants EDU-1319147, CNS-1801545, and CAREER-1943215; and by the U.S. Department of Commerce, National Institute for Standards and Technology, under Cooperative Agreement 70NANB15H330.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Secure software development is a difficult and important task. Vulnerabilities are still discovered in production code on a regular basis [1–3]. They can often result in dangerous and disastrous outcomes when exploited by malicious attackers [4–6]. These exploits can cost companies large sums of money and cause harm to the public [7–9].

There are several approaches to solving this problem. We could further invest in secure development tools such as fuzzers, static analyzers, or secure programming languages [10–13]. We could expand and enhance security education [14, 15]. We could focus on improving the security aspects of the development lifecycle [16,17]. However, none of these solutions provide a silver bullet. New tools and languages are often limited in capability or difficult to adopt [18,19]. Security education can reduce the occurrence of some vulnerabilities but not all [20]. Improving the security aspects of the development lifecycle can be difficult and costly [21, 22]. In practice, a combination of these solutions is essential, and an important question is which combination of these interventions yields the best results while minimizing the cost of investment and time to adopt. This difficult calculus suggests that it is essential to understand how and why developers introduce vulnerabilities; develop guidelines for the creation and adoption of secure development tools; and further explore developers' processes and mental models.

In this thesis, I focus on understanding the security thought processes and approaches of

software developers. Specifically, I investigate the following questions:

- How and why developers introduce, find, and fix different types of vulnerabilities (Chapter 3)
- The benefits and drawbacks of adopting secure programming languages (Chapter 4)
- How to best conduct secure software development studies to ensure experimental and external validity (Chapter 5)

The following statement summarizes the contributions of this thesis:

Humans are an essential aspect of secure software development, and we cannot achieve secure software development without first understanding the thought processes and needs of software developers and security professionals.

I validate this statement and answer these questions through a series of studies.

Understanding the vulnerabilities developers introduce, find, and fix In Chapter 3, I describe my, co-led, in-depth study of 14 teams' development processes during a three week undergraduate course centered on a *Build It, Break It, Fix it* (BIBIFI) secure-coding competition, which balances ecological validity with study control by having participants complete a multi-week, well-defined programming project with few process constraints [23]. Student teams built a software-based home IoT system with role-based access control policies. Teams then attempted to find vulnerabilities in other teams' code and fixed vulnerabilities in their own code found by other teams. The course scoring emphasized real-world constraints and priorities, i.e., security, performance, and functionality.

Implementing the BIBIFI competition as a short course allowed us to collect fine-grained data about participants' mindsets and approaches, both while developing software and when

finding vulnerabilities. Doing so allowed us to understand why participants introduced vulnerabilities, as well as how and why they found them and (sometimes) fixed them. Our prior exploration of BIBIFI submissions revealed, in depth, the type and details of introduced vulnerabilities [20]. This work confirms the prior results and adds insight into *why* developers introduce these vulnerabilities.

We find that both the extent of the formal design, and the overall development timeline, seem to relate to security outcomes. Teams with a detailed initial design tended to introduce fewer vulnerabilities than those without. However, teams with detailed initial designs tended to stick with them, so errors or misunderstandings in those designs became vulnerabilities in the finished code. In terms of timeline, teams with the fewest vulnerabilities did little or no security work on the very last days of the build phase, instead working steadily on security throughout. Conversely, teams that waited until the end to implement security often ran out of time and failed to implement key security functions, leading to many vulnerabilities.

Additionally, we find that different types of vulnerabilities are discovered differently. Teams often found vulnerabilities almost incidentally; vulnerabilities arising from the failure to implement some security requirement(s) were most often found when looking broadly for a related problem, rather than specifically for that vulnerability type. Vulnerabilities that arose from simple programming mistakes similarly tended to be found incidentally, although they were frequently found by testing for almost anything, related or not. In contrast, vulnerabilities related to misunderstanding security properties required deeper knowledge and more focused testing. Overall, student teams strongly preferred to search for simple vulnerabilities; more complex misunderstandings (and associated vulnerabilities) were rarely targeted, if at all.

Finally, we find that different types of vulnerabilities are fixed differently. Vulnerabilities relating to failing to implement some security requirement(s) were likely to remain unfixed at the

end of the study, possibly because fixing them would in many cases require rearchitecting large parts of the system design. Vulnerabilities associated with simple mistakes were often found by teams themselves, during their own build-phase testing (before attacks from other teams), and were typically easy to fix, but sometimes went unfixed due to time constraints. In contrast, vulnerabilities caused by misunderstanding security requirements were rarely found until they were exploited by other teams. Teams could only fix these vulnerabilities once they had corrected their misunderstandings, which could be challenging and time-consuming.

We observe that current automated tools tend to target vulnerability types, such as mistakes, that are easily found and exploited, but are unlikely to flag vulnerabilities that are difficult to both find and fix, such as those owing to a misunderstanding of security concepts. These vulnerabilities might be better addressed by consulting a security expert during the design process. Indeed, our results suggest the importance of including security in detailed designs, and revisiting and updating those designs while following secure development best practices.

Understanding the benefits and drawbacks of secure tools Given the importance of secure tools for identifying and preventing vulnerabilities, in Chapter 4, my colleagues and I aim to understand the barriers to and benefits of adopting secure tools, considering Rust in particular. We conducted semi-structured interviews with professional, primarily senior developers who have actively worked with Rust on their product teams, and/or attempted to get their companies to adopt Rust (n = 16). We also surveyed participants in Rust development community forums (n = 178). We asked participants about their general programming experience and experiences using and adopting Rust both personally and at a company. We also asked about the benefits and drawbacks of using Rust in both settings.

Our survey population likely represents those who view Rust at least somewhat positively, since we would not expect those who tried Rust and abandoned it to be members of Rust forums. That said, our survey population comprised people with a variety of general and Rust-specific development experience. 26% of respondents had used Rust for less than one year and they often held similar opinions to more experienced Rust users. Our results uncovered a wide variety of specific challenges and benefits.

Participants largely perceived Rust to succeed at its goals of security and performance. Other key strengths identified by participants include an active community, high-quality documentation, and clear error messages, all of which make it easy to find solutions to problems. Further, participants indicated that overall Rust benefits the development cycle in both speed and quality, and using Rust improved their mental models of secure programming in ways that extend to other languages.

However, participants also noted key drawbacks that can inhibit adoption, most seriously a steep learning curve to adjust to the paradigms that enforce security guarantees. Other concerns included dependency bloat, limited library support, slow compile times, high up-front costs, worries about future stability and maintenance, and apprehension about the ability to hire Rust programmers going forward. For our participants, these negatives, while important, were generally outweighed by the positive aspects of the language.

Lastly, participants offered advice for others wanting to advocate adoption of Rust or other secure languages: be patient, pick projects playing to the language's strengths, and offer support and mentorship during the transition.

Analyzing our findings, we offer recommendations aimed at supporting greater adoption of Rust in particular and secure languages generally. The popularity of Rust with our participants

highlights the importance of the ecosystem — tooling, documentation, community — when developing secure languages and tools that users will actually want to use. Our results also suggest that perhaps the most critical path toward increased adoption of Rust in particular is to flatten its learning curve, perhaps by finding ways to gradually train developers to use Rust's ownership and lifetimes. Further, we find that much of the cost of adoption occurs up front, while benefits tend to accrue later and with more uncertainty; security advocates should look for ways to rebalance this calculus by investing in a pipeline of trained developers and contributing to the longevity and stability of the Rust ecosystem.

Exploring alternative methods for secure development studies Finally, my prior work, as well as other prior work, highlights the importance of conducting human centered secure development studies. However, conducting secure development studies is often difficult due to the time-consuming nature of code-writing and challenges recruiting specialized populations [24–26]. Recruiting participants for professional developer studies is often challenging, as they participate in the study outside of their normal work hours [27]. This can lead to high drop-out rates in studies, resulting in smaller and less significant sample sizes [28]. In Chapter 5, my colleagues and I explore approaches to reduce the time spent by developers while maintaining the same quality and validity of the results of these experiments. We conducted a remote lab study with Python programmers, in which they completed two secure development tasks. Participants completed two symmetric encryption tasks (generating/storing a key and encrypting/decrypting data) using an assigned library and methodology. They were either tasked with writing secure code from scratch; reading existing code and finding and explaining the vulnerabilities and functionality issues in the code; or reading existing code and finding and fixing the vulnerabilities

and functionality issues in the code. Participants were then directed to share their experiences while participating in the study.

First, we find similarities between the write condition of our study and the results of the original study. We find similarities in the difference of the number of functional solutions produced for each task, the ability of participants to generate secure solutions using the two libraries, and the usability scores for the two libraries. Specifically, we find that overall participants using PyCrypto produced more functional but less secure code than those using Cryptography.io, mirroring the results of a prior study [28].

When comparing the results of the write, read, and fix conditions, we found some similarities in the abilities of participants to produce functional and secure solutions. We see similarities between the types of vulnerabilities introduced in the write condition and the types of vulnerabilities not identified in the read condition and the fix condition, but are unable to draw conclusions about the affect of the library on the security of the code.

Further, we see that participants in the read condition and the fix condition reported feeling significantly less frustrated than those in the write condition. They also reported feeling like participating in the study was fun significantly more often than those in the write condition. Finally, while participants in the read condition only spent slightly less time working on the study than those in the write condition, participants in the fix condition spent nearly half the time participating in the study.

Our result suggest that the read and fix conditions may be useful to understanding the upper and lower bounds of developers' security awareness, but that more work is needed to understand if they can detect the effects of different conditions on secure development. We conclude by making recommendations for important areas of further exploration and study design for the read

and fix conditions.

Chapter 2: Background and Related Work

We discuss related work across three key areas: measuring the types and causes of vulnerabilities developers introduce; security during software development; and methodology for conducting developer studies.

2.1 Types and causes of vulnerabilities

Several researchers have investigated the types of vulnerabilities developers introduce and the factors associated with their introduction through the use of real-world data and developer studies. We expand on this work to explore not just the types of vulnerabilities but also why and how they get introduced, found, and fixed.

2.1.1 Measuring vulnerabilities in the wild

Using metadata and code from online repositories, researchers have explored the types of vulnerabilities developers introduce and the factors that contribute to their introduction.

2.1.1.1 Measuring metadata in production code

Prior research has evaluated the relationship between different metadata properties in version-management systems and the introduction of vulnerabilities. While investigating the data associated

with commits from PHP and the Apache HTTP server, Meneely et al. found that codebases with new committing authors, a higher-than-average number of changes, and committers that edit others' code were associated with vulnerabilities [29, 30]. In follow-up work, they explored whether Linus' Law, "having more eyes on code improves its quality", applies to vulnerabilities as well [31]. Using the code in Chromium, the authors found that source code reviewed by more developers was more likely to contain a vulnerability, unless the reviewer had prior vulnerability patching experience. Finally, Śliwerski et al. explored approaches for identifying commits that addressed bugs using data from Eclipse CVS archives. They found that changes that induce fixes often span more files than other commits [32]. Perl et al. used metadata from Github and CVEs to train a classifier to identify commits that might contain vulnerabilities [33].

Other researchers have investigated trends in CVEs, the National Vulnerability Database (NVD), and Stack Overflow. Chang et al. explored CVEs and the NVD from 2007–2010, showing that the percentage of high-attacker control vulnerabilities decreased over time, but that more than 80% of all examined vulnerabilities were exploitable via network access without authentication [1]. Exploring C and C++ vulnerabilities on Stack Overflow, Zhang et al. found that 36% of CWE types were detected on Stack Overflow [34].

Measurements like these enable a high-level view of vulnerability characteristics; I expand on these findings by directly observing development processes, within a fixed project specification, to more concretely learn why and how vulnerabilities occur.

2.1.2 Measuring vulnerabilities through developer studies

Other researchers have sought to measure the introduction and discover of vulnerabilities through the use of human-centered studies. Our work complements these studies by exploring the *how* and *why* for a larger-scale project.

2.1.2.1 Exploring how developers introduce vulnerabilities

Instead of measuring production code, some researchers have investigated why and how developers introduce vulnerabilities by studying software developers. Nadi et al. explored the challenges faced by developers when using Java cryptography APIs by analyzing 100 StackOverflow posts, 100 GitHub repositories, and survey data from 48 developers [35]. They find that developers are confident in selecting the right cryptographic concepts but find it difficult to use certain algorithms correctly. Further studying the misuse and misunderstanding of Java cryptographic APIs, Oliveria et al. had developers solve puzzles that contained Java APIs and found that more experience does not correlate with the ability to identify security problems [36].

Using a secure coding competition, Ruef et al. found that teams that used C and C++ had the most efficient, but least secure solutions and teams with more programming language knowledge had more secure solutions [23]. In follow-on work, we analyzed 94 submissions to the competition to uncover the types of vulnerabilities introduced by developers when building software [20]. We found that misunderstandings of security concepts were the most prevalent in projects while mistakes were the least prevalent.

Exploring the misconfiguration of Content Security Policies (CSPs), Roth et al. uncover that complexity and inconsistence among browsers cause developers to introduce vulnerabilities

and information sources give developers insecure information [37]. Finifter et al. compared different teams' attempts to build a secure web application using different tools and frameworks [38]. They found no relationship between programming language and application security, but that automated security mechanisms were effective in preventing vulnerabilities. Exploring why developers struggle with password storage, Naiakshina et al. found that developers did not store passwords securely without prompting and had a number of misconceptions about what was required to perform this task securely [27, 39, 40].

2.1.2.2 Exploring how developers identify vulnerabilities

Other studies have experimentally investigated how effective developers are at looking for vulnerabilities. Exploring the efficacy of code review, Edmundson et al. uncovered that no participant found all three previously confirmed vulnerabilities, and more experience was not necessarily correlated with more accuracy in code review [41]. Other work suggested that users found more vulnerabilities faster with static analysis than with penetration testing [42]. Recently, Braz et al. explored why developers struggle to identify improper input validation and found that developers often failed to identify input validation vulnerabilities without prompting but were able to find the vulnerability once told to be aware of it [43]. In follow-on work, they find that asking developers to focus on security substantially increased their probability of finding vulnerabilities [44].

2.2 Security during software development

Prior work has explored how developers interact with and think about security while writing software through the use of interviews, surveys, ethnographic studies and lab studies. Our work complements these studies in a more controlled observational environment than an ethnographic study, without requiring as much self-reporting and recall as interviews or surveys.

2.2.1 Secure development processes

Some researchers have focused on understanding the personal and organizational secure development processes of developers.

2.2.1.1 Developers' processes and approaches

Some research has aimed to broadly understand security in the context of developers' processes. To explore security in each stage of the software development lifecycle, Assal et al. interviewed developers employed in industry finding a variety of approaches to software security that differ widely from best practices [45]. In follow-on work, they explore developers' motivations for and interactions with security practices uncovering that developers are motivated to write secure code but often face organizational obstacles that can result in vulnerabilities. [46]. More recently, Rauf et al. found that developers only intermittently considered security as they developed code [47]. Focusing on how open-source maintainers approach security, Wermke et al. found a large diversity in security measures and processes, with most projects not having a dedicated team for security on the project and, instead, relying on organizational resources or other team members [48]. Focusing on privacy in mobile development as it pertains to ad

networks, Mhaidli et al. found that developers prioritize profit and popularity of ad networks over privacy concerns, seeing the privacy of their app users as not their priority [49]. Some researchers have explored how to better support developers in their security processes, finding that security patterns and specifications have little to no improvement on security [50, 51].

2.2.1.2 Security processes in companies

Other researchers have studied secure development processes at companies. Exploring how company characteristics impact security, Balebeko et al. found that small companies have less positive security behaviors, such as relying on search engines for security advice and requiring security tools to be easy and cheap [52]. Investigating security processes in companies that make cryptographic products, Haney et al. found a strong security mindset guiding careful selection of resources and development practices [53].

Taking a different approach by embedding themselves in a company, Palombo et al. found that developers sometimes intentionally introduce or do not fix vulnerabilities because of varying business pressures on them [54]. Similarly, Tuladhar et al. conducted an ethnographic study to understand how a software development team adopts security during the development lifecycle [55]. By embedding themselves in the team, the authors were able to see a shift in the developers' secure development practices. They attribute this to the developers understanding how security can be applied to what they work on. They suggest the use of situated learning to help engage and create security-minded developers.

2.2.2 Secure tooling

Other researchers have explored secure development through the context of understanding the usability and adoption of secure tools. Our work complements this existing research by focusing specifically on secure programming languages, an area not explored before.

2.2.2.1 Usability of secure tools

Researchers have evaluated the usability of cryptography APIs. To understand the usability of Python cryptography APIs, Acar et al. conducted an experiment with 256 developers, finding that APIs designed for usability and simplicity do provide security benefits since they often remove the onus from the developer [28]. Patnaik et al. aim to identify specific usability issues in cryptographic libraries. They find usability issues such as missing information and not knowing what can be done with the API [56]. Focusing on deploying TLS, Krombholz et al. found that developers struggle with deploying TLS because of the complexity of the process [57]. Tiefenau et al. found that Let's Encrypt, a tool designed to simplify deploying TLS, made it easier and faster for developers to securely set up HTTPS [58].

Some researchers have focused on understanding the usability of fuzzing and static analysis tools, automated tools that allow developers to widely test their code. Other prior work has explored the feasibility of integrating secure tools into development to ease the burden of software developers. Motivated by prior work, Nguyen et al. investigated the use of an IDE plug-in to help Android developers write secure code [59]. They found that participants were able to seamlessly use their plug-in and even novice developers were able to write relatively secure code. Similarly, Gorski et al. evaluated the efficacy of API-integrated security advice to inform a

developer when they are misusing an API and offer a secure hint [60]. In an online experiment with 53 participants, they found that 73% of participants who received the advice fixed their insecure code. They conclude by recommending future work exploring the scalability of creating in-context security warnings and recommendations. Extending this evaluation to static analysis tool guidance, Tahaei et al. showed 132 participants four different vulnerabilities (SQL injection, logging sensitive data, encryption, and hard-coded credentials) paired with static analysis tool guidance and prompted participants to select the correct fix [61]. Participants held positive opinions of the guidance and liked the example solutions and vulnerable code. Despite a slight increase in correct answers with static analysis tool users, participants that used a static analysis tool still answered at least one code correction problem correctly.

Most similar to our work, other researchers have focused on the usability of programming languages and, specifically, Rust. Croft et al. studied the challenges faced by developers when using programming languages, finding that security challenges continue to be on the rise [62]. Exploring the challenges of Java, Meng et al. discovered that many challenges faced by developers were caused by APIs and libraries [63]. Focusing specifically on usability issues in Rust, researchers found that Rust's safety rules were difficult for programmers to apply in practice and that Rust cryptography APIs varied widely in usability [64, 65].

2.2.2.2 Improving secure tools

Other researchers have explored approaches to improving secure tools. Exploring what developers want from program analysis tools, Christakis et al. find that developers need to be able to dismiss warning messages and want to be able to guide the tool as needed [66]. To

better understand the needs of developers, Smith et al. observed participants interactions with a static analysis tool, finding participants ask questions about security concerns (attacks, fixes, and vulnerabilities) and questions about the software and other related resources [67]. Gorski et al. conducted a participatory design experiment with 25 professional software developers to understand which information is helpful to avoid security mistakes when issued an API misuse warning [68]. The authors detail five key considerations when designing a warning: message classification, title message, color, code location, and links to external elements. Participants emphasized the importance of context when deciding on detail and content of a warning. Danilova et al. used grounded theory and a survey to understand what developers want from security warnings [69]. They find that opinions vary widely among developers and no one warning type is preferred. Focusing on improving the learnability of Rust, Almeida et al. built RustViz to help developers build accurate mental models of the ownership model in the language [70].

2.2.2.3 Adoption of secure tools

Other researchers have investigated factors affecting secure tool adoption by developers. Xiao et al. explored the social factors influencing secure tool adoption, finding that company culture influences adoption and use of security tools through encouragement or discouragement to try new tools and managerial intervention in the security process [71]. In follow-on work, researchers surveyed developers about why they chose (not) to use security tools and found that the biggest predictor of adoption was peers demonstrating the use and benefits of the tool [72]. Haney et al. found that *security advocates* promoting tool adoption must first establish trust by being truthful about risks [73]. Other researchers have focused on the adoption of specific tools.

Sadowski et al. focused on static analysis tools by building Tricorder which integrates static analysis into developer workflow [74]. They found that developers were generally happy with the results from the static analysis tools and the number of mistakes in the codebase reduced. Johnson et al. explored the adoption of static analysis tools and found that despite developers finding static analyzers to be beneficial, their adoption is impacted by false positives and warning presentation [75]. Our work confirms these suggestions as they apply and extend to secure programming languages.

2.2.3 Information sources during development

Prior work has also explored the impact of information sources on secure development and ways to improve these resources. Our work complements this by studying other factors that impact secure development.

2.2.3.1 Structure of security information sources

Broadly surveying resources used by developers, Acar et al. found many resources contained outdated information and did not provide any concrete examples to help developers [76]. Exploring the content of and interactions with security posts on StackOverflow, researchers found that developers used StackOverflow to connect with and tend to secure problems despite the fact that insecure snippets had higher view counts, received a higher score, and has significantly more duplicates than secure snippets [77, 78].

2.2.3.2 Impact of information sources on security

Exploring the impact of information sources on code security, Acar et al. found that participants that used StackOverflow produced significantly less secure code [24]. Further exploring this phenomena, Fischer et al. quantified the presence of StackOverflow snippets in Android applications on Google Play, finding that 15.4% of the applications contained security-related snippets from StackOverflow. Of those, 97.9% had at least one insecure snippet [79]. To further understand why developers copy insecure code from StackOverflow, Bai et al. interviewed authors of GitHub repositories with security vulnerabilities from StackOverflow snippets, discovering that developers trusted their security knowledge to evaluate the snippets but understood that they needed more security knowledge to properly vet the snippets [80].

2.2.3.3 Improving security through information sources

Other researchers have focused on improving the security impact of the resources that developers use. Gorski et al. integrated security information into non-security API documentation and found that developers focus on sections with code examples and security code examples can help improve security [81]. Exploring how search rank effects security, Fischer et al. investigated the impact of the ranking of security snippets from StackOverflow in Google results. They found that reranking results to prioritize security improved the resulting code security [82].

2.3 Methodology for developer studies

More recently, work has explored methodology for developer studies along three main areas: participants for developer studies, effectively measuring developers' skills, and study

design. Our work expands upon this existing work by exploring the feasibility of reading code, identifying vulnerabilities, and fixing vulnerabilities as study methodologies, an area explored very little.

2.3.1 Participant recruitment

Some prior work has explored the validity of varying recruitment approaches. Yamashita et al. explored the use of freelance marketplaces to recruit participants, concluding that while freelance marketplaces offer flexibility, low cost, and access to a wide population, there is uncertainty about participants' background and skills [83]. Baltes et al. evaluated the use of various sampling methodologies for software development studies finding that sampling through public media to yielded the best results [84]. Acar et al. explored the use of Github as a recruitment tool for studies by recruiting 307 active GitHub users to complete security-related programming tasks, finding no statistical difference for functionality or security between participants that were students and professionals [26]. To provide insight into the ecological validity of using computer science students (CS) in developer studies, Naiakshina et al. recruited professional software developers to complete programming problems from a prior study [27] and compared the results [85]. They found that developers performed better than both students and freelancers, but the treatment effects, such as prompting for security, held the same for developers and students. Similarly, Salman et al. discovered that students and professionals do not produce substantially different results in software studies [86].

Most recently, researchers have compared and contrasted a variety of recruitment venues, finding that crowdsourcing platforms require screening to get high quality participants and that

CS students at a variety of educational stages are a viable alternative for developers [87, 88].

2.3.2 Measuring developers' skill

Prior work has explored the construction of scales and survey questions to evaluate the skills of participants in software developer studies. Feigenspan et al. wanted to build a scale to measure programming experience of study participants. To construct this scale, they used questions from published studies that evaluated programming experience. Comparing the participants' answers, they found self-efficacy to be an effective way to measure programming experience [89]. Similarly, Bergersen et al. built a scale to measure programming experience by having 65 professional developers complete 19 Java programming exercises [90]. They then validated and tested the scale for overfitting.

To aid in recruitment, Danilova et al. expanded this idea by building a screening questionnaire to help filter participants in programming studies, concluding with 6 recommended questions [91]. In follow-on work, they explored the use of time limits to increase the efficiency of the screening questionnaire, concluding that implementing a time limit on questions saves time and money while maintaining validity [92].

Focused on measuring security experience, Votipka et al. built a scale to measure the security self-efficacy of software developers [93]. The authors surveyed 22 security experts and various secure-development frameworks to develop 58 initial unique security tasks. Their resulting scale had 15 items with two sub-scales: one for measuring a belief in the ability to perform vulnerability identification and mitigation and one for measuring a belief in the ability to perform secure communication tasks.

2.3.3 Study design for secure development studies

To explore the use of task deception, often popular in end-user usable security studies, in secure software developer studies, Naiakshina et al. had 40 students complete a password storage task, finding that priming participants for security has a statistically significant effect [40]. In a replication, Danilova et al. found that deception may not be necessary for ecological validity [94].

As an alternate approach to traditional in-person lab studies, Huaman et al. built a virtual study environment to allow researchers to conduct lab studies remotely. They find that their virtual environment allows researchers to conduct studies remotely while maintaining ecological validity [95].

Most similar to our work, Danilova et al. evaluated the use of code review as a method for secure development studies, finding that code review could be a viable method for future developer studies, but they recommend that more research be done into this alternative [96].

Chapter 3: Exploring How and Why Developers Introduce, Find, and Fix Vulnerabilities

We must understand *why* developers introduce different vulnerabilities, as well as *how* and *why* testers (do not) find and fix them, in order to identify processes and tools that most effectively reduce real risks.

As discussed in Chapter 2, prior work has considered secure development through both controlled lab studies and open-source measurement (field-studies). Controlled lab studies (Section 2.1.2) allow for clear comparisons among different tools and strategies. While valuable, these studies are limited in ecological validity, as the program size and flexibility of approach are restricted by necessity. Conversely, open-source measurement studies (Section 2.1.1) provide results from a real-world setting. However, it is difficult to make clear comparisons between these codebases due to significant differences in goals and functional requirements. This research also typically cannot investigate developer motivations or thought processes, as only submitted code (with often-terse commit messages) is available.

In prior work, my co-authors and I sought to establish a middle point along this spectrum with the *Build It, Break It, Fix it* (BIBIFI) secure-coding competition, which balances ecological validity with study control by having participants complete a multi-week, well-defined programming project with few process constraints [23]. We then reviewed code submitted during four BIBIFI competitions to uncover an in-depth taxonomy of the vulnerabilities introduced by developers

while building secure software [20]. We manually analyzed submissions to discover characteristics of vulnerabilities developers introduced, such as general vulnerability type, severity, and ease of exploitation. However, as we only reviewed submitted code, we were not able to determine ***why and how*** developers introduce, find, and address vulnerabilities. Understanding this would enable better recommendations to improve secure software development, security education, and secure development tools.

To address this limitation, in this chapter I present an in-depth study of 14 teams' development processes during a three week undergraduate course centered on a BIBIFI competition. Student teams built a software-based home IoT system with role-based access control policies. Teams then attempted to find vulnerabilities in other teams' code and fixed vulnerabilities in their own code found by other teams. The course scoring emphasized real-world constraints and priorities, i.e., security, performance, and functionality.

Implementing the BIBIFI competition as a short course allowed us to collect fine-grained data about participants' mindsets and approaches, both while developing software and when finding vulnerabilities. Doing so allowed us to understand why participants introduced vulnerabilities, as well as how and why they found them and (sometimes) fixed them. Prior exploration of BIBIFI submissions revealed, in depth, the type and details of introduced vulnerabilities [20]. This work confirms the prior results and adds insight into *why* developers introduce these vulnerabilities. In particular, we consider why developers introduce different types of vulnerabilities, why different types of vulnerabilities are found during code review, and why and how developers fix different types of vulnerabilities.

The rest of this chapter is organized as follows: Section 3.1.5 reviews the data we collected in this study and how we analyzed it; Section 3.2 describes the vulnerabilities introduced by

our participants and the factors affecting the resulting security of their projects; Section 3.3 describes the vulnerabilities that were (not) exploited and how teams found them; Section 3.4 describes the vulnerabilities that were (not) fixed by teams and why they were (not) addressed; Section 3.5 discusses the implications of our results and distills recommendations for improving secure development.

3.1 Data and Analysis

This section describes the course structure and the data collected.

3.1.1 Course structure

The course followed a modified BIBIFI competition structure [23], organized into three phases: *build*, *break*, and *fix*. Course participants worked in teams for one week to *build* a substantive program (2363 LoC on average), following a provided specification (See below for specification description). Teams earned points based on the number of correct functions they implemented (evaluated via instructor-defined test cases) and their code's performance (running time of instructor test cases).

After the build phase, the course used a hybrid break-fix phase. All teams' code was made available to the other teams, which could then attempt to *break* their classmates' submissions by producing test cases demonstrating vulnerabilities. Teams were also allowed to submit vulnerability reports that clearly stated the insecure line(s) of code in the target program and explained the vulnerability, as well as why it was not feasible (within class constraints) to demonstrate it. This could occur due to competition time constraints (e.g., brute force exploitation not feasible in the

course timeframe without specialized equipment) or exploit complexity. We added this option because in previous contests it was not clear whether teams had failed to find vulnerabilities, or had found them but been unable to exploit them [20, 23]. None of these reports were submitted.

Teams gained points for each valid break submitted. To incentivize unique breaks, different teams exploiting/reporting the same vulnerability shared its points evenly. As breaks were identified, teams could update their code to *fix* vulnerabilities; teams lost points for every 24 hours that a known break went unfixed. This hybrid break-fix setup is a departure from prior BIBIFI exercises, where the break and fix phases occur sequentially [23]. We made this change to better reflect real-world scenarios in which developers often have to handle fixing issues in code while managing other responsibilities and encourage teams to perform fixes, which was a problem in previous competition iterations [20, 23].

3.1.2 Project Specification

The build-it project was a lightweight IoT smart home controller that manages a smart home by receiving updates from sensors and controlling output devices. The specified controller can run user scripts, written in a domain-specific language, that update output devices (e.g. lights, thermometer, etc), retrieve information about the current state of the system (e.g. lights on/off, thermometer temperature, etc), or store information for future use (e.g. variables, permissions for users, new users, etc). Users of this smart home submit scripts in the domain-specific language. Sensors, output devices, and data are protected by role-based access control policies customizable by special users (admin) and the data's owner. Other users may receive delegated access, directly or transitively. This role-based access control was expected to be recursive,

such that if a parent (delegator) lost permission on a variable, its children (delegatees) also lost permission. Further, the controller must validate user-provided scripts so that they will only run if properly authorized. Teams were expected to implement authentication using passwords and usernames. For extra build points, teams could implement some commands that operate similarly to required commands, such as for loops, indexing variable history, or resetting variable history. The full specification can be found at https://osf.io/s8ztb/?view_only=f8b9fb7804ab46648000ff1f52ca0d6c.

3.1.3 Class Instruction

Since participants were not required to have prior security training, the course started with a short introduction to security and threat modeling taught by one of the researchers. This training ensured a minimum level of security knowledge among participants and covered examples and approaches for thinking about possible attack surfaces and mitigations when developing a program. Teams were given time in class (about 11 hours total, over the 2.5-week course) to build their system, break other systems, and fix breaks. This ensured dedicated time for teams to work collaboratively. Teams were also expected to work on the competition outside class time, in lieu of other homework. During class time, teams were able to ask the instructors—the researcher who provided the original lecture and another research team member—to clarify vagueness in the specification and resolve issues teams had working with the course infrastructure. Any specification clarifications were announced to the entire class and updated in the specification document to ensure consistency between teams. When teams asked for help or advice with system design or implementation, the instructors directed students through problem introspection asking

teams first to describe what is correct about their approach and how they know, then to identify differences from correct outputs. This series of introspective questions is taken from prior work investigating successful CS tutoring approaches [97].

3.1.4 Course-centered research design

For this classroom-based research, we drew on methodologies common to CS and engineering education research [98, 99]. In this complex setting, education research suggests collecting three types of data—*activity* (e.g., observations [100]), self-reported *accounts* (e.g., interviews [101, 102] and surveys [103, 104]), and *artifacts* (e.g., program designs [105, 106] and code [107])—and triangulating results among these data points [98, pg 181-186]. We adopt this approach, collecting *accounts* and *artifacts* in each BIBIFI phase, as detailed in the following sections (Sections 3.1.6-3.1.8.1). We considered collecting *activity* data, but chose not to as it did not make sense in the setting of our study. Much of our teams' work was completed outside of class, so direct observations were limited and self-reporting of this kind would likely be vague or unreliable [108]; create significant additional work for participants, potentially causing response fatigue [109]; and would offer limited additional information from the other data sources. Because we collect multiple types of *account* and *artifact* data, we believe we have sufficient information to provide useful triangulation.

3.1.5 Data analysis

We analyzed most of this data through *iterative initial coding*, sometimes also known as *open coding* [110, 111]. This analysis produces a set of labels called a *codebook*. Our codebook

Round	Data	Subcomponent	Description	Frequency	N
Build	Build submission	Code	–	When added single feature or fixed single bug	676
		Commit message	Description of change Reason for change Associated requirement How did the team come up with this change? Did teams change their design?	Each build commit	676
	Design documents	–	Team design of system Detail of potential threats Detail of potential mitigations	Before build, after build	27
Break	Break submission	Attack submission	Test to be run on target Description of exploit	Each attack	275
		Commit message	Description of issue in target How was the issue found? Requirement broken by issue Fixed version of failed break Difficulty to find break Difficulty to exploit break	Each attack	275
Fix	Fix submission	Code	Addresses vulnerability in their code	Each fix submission	48
		Commit message	Issue with implementation Cause of the vulnerability How did the team fix the vulnerability Confidence in security of their code Difficulty to identify the vulnerability Update to design required?	Each fix submission	48

Figure 3.1: Description of data collected throughout the study.

provides labels for the different elements and can be found in Table 3.1. Inter-rater reliability (IRR) was generally not calculated, as the small number of responses and submissions for many aspects of the data did not allow for it. When we did calculate IRR, we used Krippendorff's α, a conservative measure that considers how much better coders are than simply guessing [112]. A threshold of $\alpha > 0.8$ is recommended as a sufficient level of agreement [112]. We specifically highlight cases where we calculate IRR in the following sections. For most of the data analysis, the first author and one other team member coded the data. We specifically highlight the one case where this is different. The following subsections describe the coding processes and codebooks for each different major data category.

Team	Member	Year	Major	Dev Experience	Dev Skill	Sec Skill	Sec Training
3	1	Junior	Math	0 years	Novice	Awareness	Coursework
	2	Senior	CS	1 year	Intermediate	Novice	Coursework
5	1	Senior	Physics	0 years	Intermediate	Awareness	Coursework
	2	Junior	CS	4 years	Intermediate	Novice	Coursework
6	1	Graduate student	ENGR	2 years	Novice	Novice	Coursework
7	1	Junior	CS/SEC	–	–	–	Coursework
	2	Sophomore	CS	1 year	Novice	Novice	Coursework
8	1	Senior	CS	2 years	Intermediate	Intermediate	Coursework
	2	Senior	CS	1 year	Intermediate	Novice	Coursework
9	1	Senior	CS	7 years	Intermediate	Intermediate	Coursework
10	1	Senior	CS	1 year	Intermediate	Awareness	Coursework
	2	Senior	CS	0 years	Awareness	Awareness	Coursework
11	1	Senior	CS	3 years	Advanced	Awareness	Coursework
13	1	Senior	CS	0 years	Intermediate	Novice	Coursework
	2	Senior	Criminal justice	0 years	Novice	Awareness	Coursework
14	1	Senior	CE	2 years	Intermediate	Novice	Coursework
16	1	Junior	CS/SEC	–	–	–	Coursework
	2	Senior	CS	1 year	Intermediate	Novice	Coursework
18	1	Junior	CS	1 years	Intermediate	Intermediate	Coursework
	2	Senior	CS	4 years	Intermediate	Awareness	Coursework
19	1	Senior	CS	3 years	Intermediate	Intermediate	Coursework
	2	Senior	CS	0 years	Intermediate	Intermediate	Coursework
21	1	Junior	CS/SEC	6 years	Intermediate	Intermediate	Coursework
	2	Junior	CS	1 year	Intermediate	Novice	Coursework

Table 3.1: Demographics of the teams

3.1.6 Build data

During the build phase, we collected and initial-coded data from three sources: build submissions (*artifact*), commit messages (*account*), and design documents (*artifact*).

3.1.6.1 Build submissions

While writing project code, participants were instructed to produce regular, atomic commits (i.e., a single change per commit) to a designated gitlab repository accessible by the teaching and research staff. To incentivize regular submissions, each commit automatically triggered performance and correctness testing to update the team's performance and functionality scores, giving participants immediate feedback on their efforts. With regular, atomic commits, we were able to track participants' progression, implementation strategies, and areas of focus.

Students were also instructed to discuss the context of the commit (e.g., what changed, reason for change, associated requirement, impact on design) in each commit message. The full set of items required in build commit messages can be found in Table 3.1. These messages provide an important window into participants' development process: what they were working on, the reasoning behind their approach, and how it fits into their design. By collecting this data throughout the competition, we also learn about the development timeline and what events trigger changes. Perhaps most importantly, detailed commit messages frequently allow us to differentiate vulnerabilities arising from misunderstanding of security concepts from implementation errors, helping us to understand why different vulnerabilities are being introduced.

3.1.6.2 Design documents

To provide additional insight into teams' understanding of security requirements, we asked them to produce design documents. Teams were required to submit documents detailing their system's design, and enumerating potential threats and associated mitigations, as described in Table 3.1. Using this data, we can learn more about the causes of vulnerabilities, and distinguish

errors in design from implementation errors. Students submitted initial design documents individually (16 students) before development began (teams had not yet been formed), and then teams submitted final design documents at the end of the build phase (11 teams), providing insight into how their designs change as they work together to develop, as well as how and whether security issues are introduced and/or eliminated over time.

3.1.6.3 Analysis of Build data

We analyzed the commit messages and the code submissions using iterative initial coding, as described in Section 3.1.5. For the build submissions, the first author and an additional author—different from the second coder discussed in subsequent sections—performed the analysis. This second coder was selected specifically because expert security knowledge was required to identify vulnerabilities in the build submissions. The two coders looked for vulnerabilities independently in each team's code; later, break submissions were also reviewed to ensure no vulnerabilities were missed and all were appropriately categorized. A vulnerability was only labeled within the codebase once a team attempted to implement that functionality or security. Teams did not start with *No Implementation* vulnerabilities because their code did not have any access control (because they hadn't built anything yet), but, rather, as teams built individual pieces of functionality, without any access control, vulnerabilities were labeled accordingly.

IRR was calculated for all build submission variables other than binary variables (the presence of a vulnerability or change in design document), since coding binary variables requires little to no interpretation, as recommended in prior work [113]. All vulnerabilities were confirmed by both coders, and consensus for all codes was reached via discussion. We categorize different

vulnerability types using our prior codebook [20]. The final codebook and associated IRR values are given in Table 3.2.

Associated data	Analysis focus	Codes	Description	Agreement
Build code submission	Associated requirement	Setup	Setup team's environment	0.844
		Parser	Parsing commands	
		Prim Cmds	Non-security functionality of primitive commands	
		Authentication	Password checking/initialization	
		Access control	Access control checks	
		Networking	Network communication	
		System state	Managing state of program variables	
		Input handling	Command line/config file input	
		Optional cmds	Non-security functionality of optional features	
		Testing	Testing specific inputs	
		Security fix	Fix of a security bug	
	# lines changed in the code	–	–	–
	Introduced vulnerability	–	Number of vulnerabilities introduced	–
	Vulnerability type	See Table ??	Type of vulnerability introduced	–
	Vulnerabilities fixed	Yes, No	Vulnerabilities fixed in future commit	1
	Fix any prior vulnerabilities		Number of vulnerabilities fixed	1
Build commit message	Associated requirement	Same as code	–	0.899
	Reason for change	Initial	First commit for this piece	1
		Bug fix	Fixed bug	
		Missing functionality	Added feature that was missing	
		Simplification	Making code easier to work with	
		Performance	Increased efficiency	
		Extra feature	Optional cmds	
		Integration	Integrate multiple components	
		Testing	Adding testing to look for bugs	
	How team came up with change	Instructor	Info from instructor	0.888
		Prior exp	Based on knowledge from before	
		Spec	Info from specification	
		Personal preference	Code is more readable/easier for later	
		StackOverflow	Read answers from SO	
		Other tutorials	Other online tutorial (not SO)	
		Teamwork	Discussion with teammate	
		Class forum	Discussion from class forum	
		Oracle	Oracle to determine output	
		Brute force	Tried many things	
		Testing	Realized when testing	
	Design document updated	Yes,No	Whether team updated their design	1

Figure 3.2: Descriptions and agreement of codes for build round

To analyze the initial design documents, we merged the 16 individual students' submissions into 11 teams, using a conservative process where if one team member's document included

something (e.g., a design decision or a vulnerability) we considered it present for the team. When considering whether a design was in-depth, we used the deepest of the team members' designs.

Due to the small sample sizes for the design documents, we coded these collaboratively rather than independently. Two coders analyzed the documents iteratively, coding two documents per round independently and then meeting to discuss and resolve differences. The codebook is provided in Table 3.3. We use this analysis to explore trends connecting design depth and the resulting security of the code. This allowed us to better understand the cause of different types of vulnerabilities.

Associated data	**Analysis focus**	**Codes**	**Description**	**Agreement**
Design documents	Thoroughness of the design	1 - 5	–	–
	Included components	See Table **??**	What requirements did they include	–
	Included relationships	Functionality	Relationships between functionality components	–
		Security	Relationships between security and functionality components	
	Visualizations included	Yes,No	Included visualization in design document	–
	Text included	Yes,No	Descriptive text in design document	–
	Components changed	Yes,No	Did components change from version 1 to 2	–

Figure 3.3: Descriptions and agreement of codes for participant data

3.1.7 Break data

During the break phase, we collected break submissions (*artifact*) and commit messages (*account*). While the break and fix phases occurred simultaneously, we describe them separately for simplicity.

3.1.7.1 Break submissions

Break submissions include test cases demonstrating insecure behavior (as compared to an oracle implementation provided by the instructors), as well as written vulnerability reports (see Section 3.1.5). As in the build phase, teams were asked to include detailed commit messages with each submission. Teams were prompted to supply a description of the break, their reasoning, and their perception of the difficulty both of finding the vulnerability and generating a working exploit (test case).

Taken together, these two data sources illuminate participants' testing strategy: what do they try, in which order? Do they fully understand the vulnerability they are exploiting, or are they brute-forcing some aspect of it? Do they focus on finding all problems in one target, or try the same (or similar) exploit against multiple teams? This data also provides insight into what types of vulnerabilities are (not) easy to find and to exploit. Details about the break submissions and commit messages are given in Table 3.1.

3.1.7.2 Analysis of Break data

Despite the large number of submitted breaks (N= 275), only a subset were successful and unique (N=52). The two coders analyzed breaks and associated commit messages in rounds of 20, meeting in between to discuss and resolve differences. The codebook is shown in Table 3.4.

3.1.8 Fix data

In the fix phase, we collected fix submissions (*artifact*) and commit messages (*account*).

Associated data	Analysis focus	Codes	Description	Agreement
Test to run	Target	Team name	Name of target team	–
	Successful	Binary	Was the attack a success	–
	Type	Availability	Type of vulnerability	–
		Confidentiality		
		Integrity		
	Issue	Skipped algorithmic step	Team skipped a step	–
		Insufficient error checking	Vulnerability caused by lack of error chekcing	
		Control flow mistake	Flow of code is unintended	
		Incorrect input assumptions	Assume input is formatted certain way	
		Incorrect use of library	Misuse library function	
		Incorrect access control assumptions	Wrong understanding of access control rules	
		No access control	No access control implemented	
		Uncaught runtime server error	Don't catch exception	
		Resource exhaustion	Don't plan for possible resource exhaustion	
		No timeout	Do not implement a timeout	
	Single/multiple	Single	Vulnerable code in one spot	–
		Multiple	Vulnerable code in multiple spots	
	Difficulty to exploit	Single-step	Exploitable out of box	–
		Multi-step, small	Small manipulations to system needed before it is exploitable	
		Multi-step, large	Many manipulations needed	
	Difficulty to find	Execution	Found by testing spec	–
		Execution and source	Dig into source	
			Goes beyond spec	
		Execution, source, an underlying concepts	Deep understanding of concepts needed	
	Spray	Binary	Vulnerability part of a group of vulnerabilities submitted against every team	–
Description of exploit	Clarity	1 - 5	How clear is provided description	–
	Lines of code	Code included in description	–	
	Example output	Example output included in description		–
	Length	–	# of characters	–
	Readability	–	FRES readability [**?**]	–
Commit message	Description	See Table **??**	Description of what	–
		requirement the vulnerability breaks	–	
	How found	Used spec	Found vulnerability by using specification	–
		Break against themselves	Vulnerability was a break against them	
		Testing	Found vulnerability through testing	–
		Vulnerability during build	Vulnerability in their system during build round	
		Intuition	Used intuition to find vulnerability	
		Brute force	Tried many thigns	
		Exploiting other vulnerability	Found this vulnerability while exploiting another	
		Oracle	Used the Oracle	
		Vulnerability in many systems	Found in many systems	
		Code review	Found by reviewing code	
		Previous issue	Found other exploits for this functionality before	
	Associated requirement	Availability	–	–
		Confidentiality		
		Integrity		
	Fix of a failed break	Binary	Fixed a vulnerability that didn't work before	–
	Difficulty to find	1 - 5	How hard to find vulnerability	–
	Difficulty to exploit	1 - 5	How hard to exploit vulnerability	–

Figure 3.4: Descriptions and agreement of codes for break data

3.1.8.1 Fix submissions

Participants patched their code in response to a break by committing an update to their git repository. Teams were incentivized to produce fixes quickly, as they lost more points the longer a break went unfixed. As before, participants were prompted to provide detailed commit messages, this time describing how they fixed the vulnerability, its underlying cause, their confidence the fix resolved the vulnerability, and whether they changed their design (versus just their implementation). Details are given in Table 3.1.

Submitted fixes and commit messages allow us to understand whether participants understand the vulnerability and can identify the cause in their code. This data allows us to understand where and why participants struggled to address vulnerabilities and provides further insight into whether vulnerabilities were caused by poor design or implementation.

3.1.8.2 Analysis of Fix data

The small number of unique breaks led to a correspondingly small number of fix submissions (N=48). As such, we again coded this data collaboratively, with two coders meeting after every 10 submissions to discuss and resolve differences. Codebook details are given in Table 3.5.

3.1.9 Ethics

This study was approved by the University of Maryland Institutional Review Board (IRB). Participants were informed that a study was being run within the course and that by signing up for the course, they were consenting to being a part of the study. This information was clearly presented on the course website, in the course posting, and directly via email when students

Associated data	Analysis focus	Codes	Description	Agreement
Submission	Fix significance	Low	Amount of change to code/design for fix	–
		Mid		
		High		
	Vulnerabilities introduced	Binary	Other vulnerabilitiesintroduced during fix	–
	Break type	Availability	–	–
		Confidentiality		
		Integrity		
	Issue	See Table ??	–	
Commit message	What was wrong	–	Team description of vulnerability	–
	What caused problem	Table ??	Cause of vulnerability	–
	How fixed	Added missing piece	Added missing thing that causes vulnerability	
		Fixed programming mistake	Fixed mistake that caused vulnerability	
		Adjusted code to reflect spec	Altered code to match specification	
	Design updated	Binary	Did fix require update to design	–

Figure 3.5: Descriptions and agreement of codes for fix data

registered for the course. We also presented this information to students on the first day of class to ensure all students were clearly informed.

To avoid coercion, this 1-credit course was offered during an off-peak semester and was not major-required. Because students were assigned a grade for the course, we tailored the requirements so that a majority of the points were given if the student met all functionality requirements and was active throughout the break/fix phase. Specifically, students could achieve an A even if they were lower-ranked in the competition. Remaining points were allocated based on competition rank to ensure motivation.

3.1.10 Limitations

Our goal in this study was to understand the challenges faced by developers when pursuing secure development, using students as proxies for professionals. While students often have less experience than professionals, several recent studies conclude they can be adequate substitutes in secure software development studies [27, 85, 87, 88], as many skilled professionals have limited

experience with security specifically [20, 23, 49, 114–119]. However, it is likely students have lower ability to find vulnerabilities as compared professionals. As such, we consider the vulnerability hunting done by students as a lower bound and, as security experts, provide additional review for missed vulnerabilities. Still, we believe lessons from this work can apply to more experienced developers.

In some ways, our study represents a best-case scenario for security considerations. Because participants were explicitly asked to reflect on their progress and design, they likely put more effort into considering design and evaluating their development process than they otherwise might have. It is possible that encouraging regular, atomic commits could improve code security, as this is recommended best practice [120], but prior studies of this relationship have been inconclusive [30]. Additional reflection about the reasoning behind commits is helpful for development [121] and learning generally [122,123], so our results likely demonstrate an upper bound. However, this allows us to understand development progression, misunderstandings of security and requirements, and design approaches. Relatedly, participants were aware they needed to include security in their system from the outset, which may not accurately reflect general developer experiences. However, this allows us to understand how vulnerabilities that occur even when developers explicitly consider security from the beginning.

Our results also have the normal limitations of field work, including that we can only observe associations, not causality. Our study was conducted with a limited sample size (14 teams) at only one institution. Despite its limitations, field work is a common practice in HCI research [124] and has been shown to be useful in security research [54, 55, 125, 126]. The BIBIFI contest does not perfectly simulate realistic development settings. Our participants were able to choose their own development environments and programming languages, which is often

not the case in company settings. Our participants were tasked with building an entire (small) system from scratch rather than the common developer tasks of modifying or managing existing codebases. Despite these differences, studying the reason for (in)secure development from this lens gives us insights that may prove useful for improving broader practices.

3.2 Development and Vulnerabilities

Our analysis of code submissions identified 145 unique vulnerabilities introduced throughout the build round and 79 unique vulnerabilities remaining in participants' code at the conclusion of the build round, with a median of 12.5 vulnerabilities remaining per team. Here, *unique* refers to non-repeating-per-team vulnerabilities; the same type of vulnerability could count for more than one team. In contrast, throughout Sections 3.1-3.3, we use *distinct* to refer to distinct types of vulnerabilities, such that two teams could have the same *No Implementation*, *Misunderstanding*, or *Mistake*, but it is only counted once. Vulnerabilities in build submissions were only labeled once teams attempted to implement the missing functionality, as mentioned in Section 3.1.6. These vulnerabilities can be categorized into 10 different issues. We employed a similar codebook to the one we used when analyzing prior BIBIFI submissions [20] , re-deriving codes from the original codebook. Our codebook includes one addition, as our specification required a mechanism for timing out when input is no longer received from a user script, where prior iterations did not.

This section presents vulnerability descriptions and examples, as well as relationships between the software development techniques and strategies teams employed and the vulnerabilities they introduce. Throughout this section, we use V to represent the number of vulnerabilities

present and T to represent the number of teams. We select illustrative examples in each of the three major vulnerability categories (*No Implementations*, *Misunderstandings*, and *Mistakes*), but do not comprehensively describe all vulnerability types in each category; as such, the V values within each subsection do not always sum to the total for that category. A comprehensive accounting of all vulnerabilities for each phase of the competition is given in Table 3.2.

We coded vulnerabilities based on three high-level categories established in our prior work [20]: *No Implementation*, *Misunderstanding*, and *Mistake*. Our results closely mirror the distribution found in the multiuser database problem (most similar to our project) with slightly fewer *No Implementation* vulnerabilities in our iteration. The tighter control of the classroom setting in our study likely influenced teams to implement needed access control checks, but teams in our study still missed many checks, as described below. We are able to further expand the description of the vulnerabilities from the prior paper by explaining the reasons for, and the process of introducing, the different vulnerabilities.

3.2.1 No Implementation

A vulnerability was labeled as *No Implementation* when participants failed to attempt to implement necessary security mechanisms (i.e., access control, authentication, or timeout mechanisms) at all. We further divide this type into three sub-types depending on whether the requirements were mentioned directly in the specification. Specifically, the *All Intuitive* and *Some Intuitive* codes were used when teams failed to implement all or some, respectively, of the stated security requirements (e.g., missing all or some access control commands). *Unintuitive* requirements were not as explicit within the specification (e.g. recursive delegation).

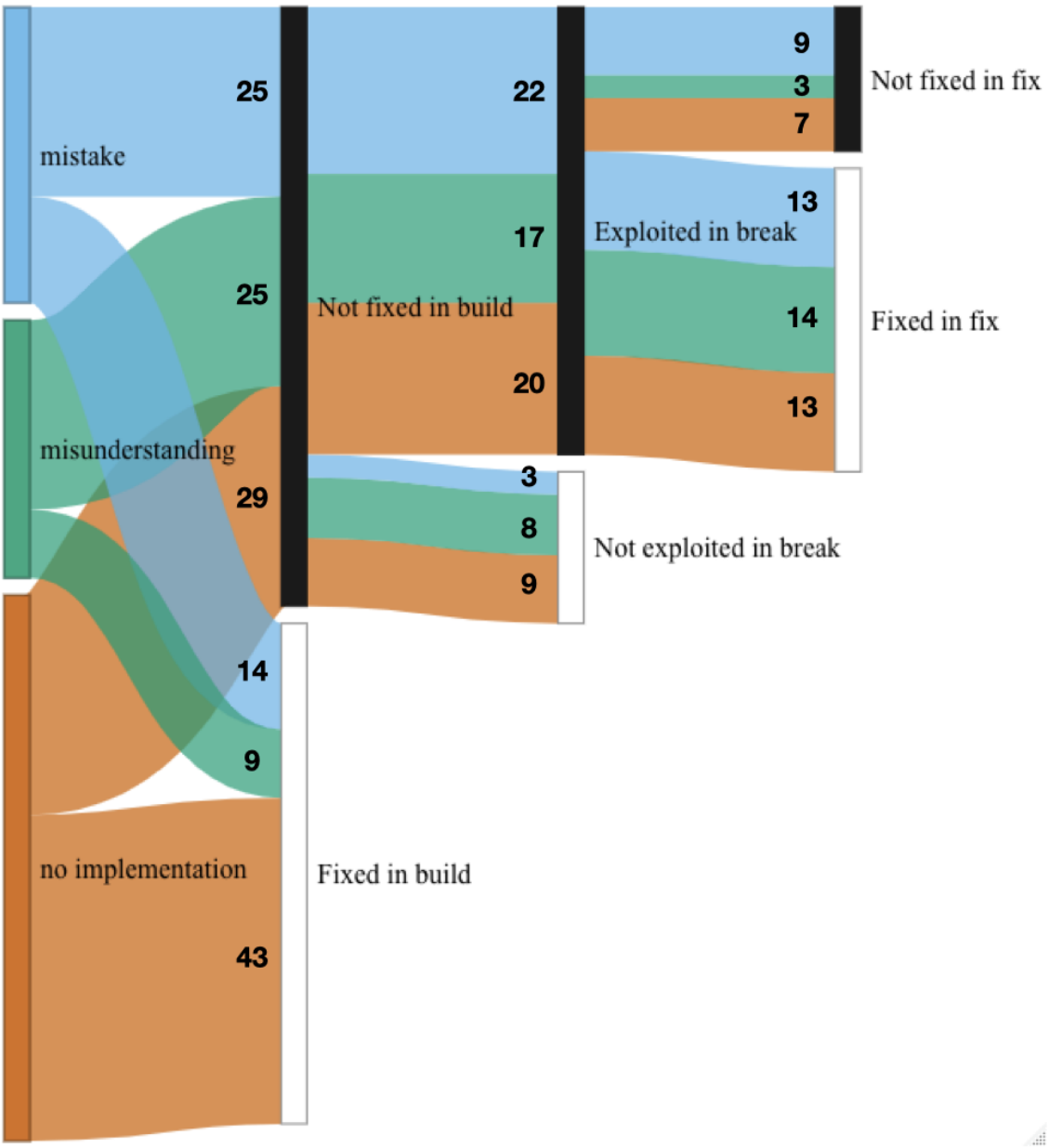

Figure 3.6: Vulnerability types throughout each phase.

3.2.1.1 Teams implement more obvious security features, only missing them when they are rushed

Six teams had a total of 29 distinct *No Implementation* vulnerability types present in their projects after the build phase. Teams were explicitly told to implement access control and issue

Type	Issue	Build		Break		Fix	
		Introduced	**Fixed**	**Exploited**	**Unexploited**	**Fixed**	**Unfixed**
No Implementation	No access control	22	22	–	–	–	–
	Missing some access control	10	3	3	4	1	2
	No timeout	6	3	3	–	1	2
	No recursive delegation check	34	15	14	5	11	3
	Total	72	43	20	9	13	7
Misunderstanding	Incorrect access control assumptions	29	9	13	7	10	3
	Incorrect input assumptions	5	0	4	1	4	–
	Total	34	9	17	8	14	3
Mistake	Control flow mistake	16	12	2	2	2	–
	Insufficient error checking	14	1	13	–	7	6
	Skipped algorithmic step	8	1	6	1	3	3
	Uncaught runtime server error	1	0	1	–	1	–
	Total	39	14	22	3	13	9

Table 3.2: Vulnerabilities introduced, (un)fixed, and (un)exploited throughout each phase.

a timeout when the IoT controller does not receive an indication of termination or input from a user-issued script after 30 seconds (making them *All Intuitive* requirements). Three teams missed this *All Intuitive* requirement (V = 3, T = 3). Three teams missed some required access control checks (V = 7, T = 3). primitive commands until late in the build phase and then forgot to add the necessary access control checks, introducing these vulnerabilities within the last 3 days. These teams built their access control checks into each individual primitive command function, rather than building one function to handle all access control checks, so they would have needed to write an entire new check for these last-minute additions (T = 3). We further describe the effect of development timeline on their code security in Section 3.2.4.

3.2.1.2 Teams fail to implement less obvious security features

Most *No Implementation* vulnerabilities resulted from teams missing the *Unintuitive* requirement of a recursive delegation check (V = 19, T = 5). According to the specification, a principal p has a right r on a variable x if there exists some principal q that has r on x and the current security state includes an assertion that q is delegating r on x to p. Based on this description, teams were expected to manage delegations in a tree-like structure, where loss of permissions from a parent results in loss of permissions to the child. This means that teams must check and manage the rights of children and parents simultaneously at time-of-use. This requirement was not explicitly stated in the specification, meaning that teams had to infer its implementation from the requirements. As mentioned by T6, the reason for this vulnerability in their system was because they "didn't realize [the] system needed to search through all assertions," which was common among the teams that addressed this vulnerability. Teams that introduced this vulnerability often built their code using individual access control checks for each primitive command rather than creating one function to manage it (T = 5).

3.2.2 *Misunderstanding*

A vulnerability was labeled as a *Misunderstanding* when teams attempted to implement necessary security requirements but misunderstood requirements or security concepts during implementation.

3.2.2.1 Teams conceptually misunderstood access control requirements

Eleven teams had a total of 34 distinct *Misunderstanding* vulnerability types throughout the build phase, with 25 remaining at the end. The most pervasive *Misunderstanding* was caused by incorrect access-control assumptions (V = 20, T = 10). For example, two teams attempted to implement recursive rights but did not properly remove rights for participants in a delegation chain. When a parent delegator lost a right on a variable, these teams would remove the this right from all of its delegatees, correctly implementing recursive rights. However, these teams would remove every instance of this right from the delegatee, incorrectly removing delegations of this right from other parent delegators, causing an availability violation. T4 said this vulnerability was caused by poor "assumptions .... I assumed we always needed to revoke the permission recursively, but I never actually checked if the user still had access." This is a conceptual *Misunderstanding* of the access control requirements, because they thought that they should remove all rights rather than just a subset. In this case, the team had a fundamental *Misunderstanding* of the requirements in the specification, which was fairly common among teams (T = 3). Other vulnerabilities related to special accounts in the system and their restricted capabilities, how to handle the permissions for a specific primitive command, and how to handle recursive delegation.

3.2.2.2 Teams implement most of the access control requirements, but fail to consider all cases

The most common cause of *Misunderstanding* vulnerabilities was a team missing a specific subcase of a requirement (T = 4). These included building recursive delegation but failing to

account for circular delegations, failing to check the running principal during delegation, and allowing another principal to delegate on behalf of the delegator. These teams did attempt to implement recursive delegation (unlike *No Implementation*) and did not fundamentally misunderstand the underlying concepts. They just misunderstood a single facet of the requirement.

3.2.2.3 Teams overgeneralized security requirements based on provided tests

Participants also struggled with building input-handling code to correctly handle user scripts (V = 5, T = 4). For example, four teams assumed all user scripts would end with a newline character and wrote their code accordingly (by reading input line by line). This resulted in an availability violation, as an attacker could generate a script that did not end in a newline — a correct script according to the specification—causing the parser to hang and prevent future connections. T2 identified the cause of this vulnerability in their code as being "an assumption about the input cases. I assumed it was not valid if a newline was not present at the end. Since all tests contained one." Note that the specification makes no mention that the scripts must end with a newline. This team overgeneralized assumptions from the provided examples, which caused them to not fully consider how the timeout setup should work.

3.2.3 *Mistake*

A vulnerability was labeled as a *Mistake* when participants attempted to correctly implement security checks and functionality, but made a programming mistake that resulted in a vulnerability.

3.2.3.1 Teams failed to test for edge cases beyond provided tests

Eleven teams had a total of 25 distinct *Mistake* vulnerability types present at the end of, and 39 throughout, the build phase. The most common issue was insufficient error checking (V = 13, T = 8), commonly a lack of null value checking during script parsing. This caused the server to crash on certain attacker-crafted malicious scripts, leading to availability violations (V = 7, T = 5). These crashes were restricted to availability exploits because teams exclusively used memory-safe programming languages. However, if teams were under performance or cultural pressure from their employer to use an unsafe language, these vulnerabilities could have led to even more serious exploits. Teams often attributed these *Mistakes* to missing an edge case when implementing functionality or security: specifically not checking for certain values (T = 3). T8 attributed a *Mistake* in their code to the fact that they "did not have a null check before attempting to access an element." These *Mistakes* were often introduced within the last 3 days of the build phase (V = 7), making them more difficult to catch.

3.2.3.2 Teams failed to test for security

Other *Mistakes* included teams skipping a step while trying to implement security (V = 7, T = 6) and incorrect implementations of control flow (V = 4, T = 4). Our participants noted two causes for these vulnerabilities: code that runs functionally but not securely (T = 1), and code that runs securely except for a specific input (T = 2). For example, teams were required to roll back all changes in the event of a failure or security violation while processing a user script. T6 set up their system so that when a status of "FAILED" was returned, they would not commit any of the changes from this set of primitive commands. However, while testing, to simplify

debugging, they changed the code to print "FAILED" rather than return it and simply forgot to revert this change. The code containing this vulnerability would run functionally but would not meet the security requirements of the specification. Given that the tests provided were centered on functionality, teams did not uncover these vulnerabilities unless they specifically tested their own code for security.

3.2.4 Development approach's security impact

We next review trends relating different development strategies to the introduction of different types of vulnerabilities).

3.2.4.1 Detailed design is common with fewer vulnerabilities

Analyzing the final submitted design documents, we found four teams submitted a detailed design document, four submitted a design document containing minimal design considerations, and three submitted a design document containing few to no design considerations. Teams with detailed design explicitly described how they were planning to implement access control. For example, T1 designed explicitly for access control and transitive rights by using "a directed graph to represent the access rights available to each principal for each variable or rule. Nodes in the graph represent each principal, while edges, labeled with one of the four access types, represent access delegation from one principal to another." Teams with minimal design considerations mention of the necessity of a security feature but do not describe how they plan to address it in their code. In their design, T8 notes that "tampering could be prevented by keeping a strict record of what users have what rights," but they do not mention any details about how they will store

or manage this. Teams with little to no design considerations do not mention access control or authentication at all.

When comparing a team's design depth to the security of their submission, we observed teams with detailed designs also tended to introduce fewer *No Implementation* and *Mistake* vulnerabilities. The four teams with detailed designs only accounted for one (5%) of the *No Implementation* vulnerabilities and 33% of the *Mistake* vulnerabilities. Conversely, the four teams with minimal designs accounted for 56% and 42% of the vulnerabilities respectively. For example, T8 describes one possible threat: "tampering could happen if a user writes to data values it does not have write access to." However, they provide very little detail about how they intend to mitigate this attack: "tampering could be prevented by keeping a strict record of what users have what rights to each variable, sensor, output device, or rule." T8 makes no mention of how they will manage this record or the need for recursive delegation checks, and they failed to implement recursive delegation checks. Similarly, we note that teams that failed to implement the *All Intuitive* timeout requirement (T = 3), missed some access control checks (T = 2), and failed to implement recursive delegation checks (T = 4) failed to make mention of these requirements within their design documents.

It seems plausible that developer experience, instead of just better planning, might cause both better planning and better outcomes. However, course teams exhibited little variation in experience: every team had at least one member self-rated as "novice" in secure development experience and "intermediate" in software development experience. All teams but one had at least one member (most teams both) with academic secure development training. Demographics can be seen in Table 3.1.

3.2.4.2 Teams with detailed designs did not revisit their design even if vulnerable, especially for *Misunderstandings*

We observed that teams with a detailed design tended to stick with those designs, which sometimes encoded initial *Misunderstandings* into their projects. Teams that had a fundamental *Misunderstanding* of security requirements designed in detail for these features within their design documents (T = 3). For example, T7 had a detailed design document in which they describe how they will handle the access control requirement described in the specification through the use of "an access control component that controls authentication and monitors which principal should have access to what" in which they "maintain a list of 'direct' objects that keep track of which variables principals can access directly and 'delegate objects' that keep track of rights that have been delegated... and ensures that principals delegating rights are actually permitted to do so." However, despite their detailed design, T7 failed to consider the possibility of circular delegation chains, in which the parent delegator and the last account to receive permissions are the same. This results in a *Misunderstanding* vulnerability, which caused an infinite loop during a delegation check. This suggests that teams that misunderstand the system's security needs from the beginning (i.e., design-time) are unlikely to catch these issues later.

3.2.4.3 Building security early correlates with secure development

When we consider teams' development timelines in comparison to the number of vulnerabilities introduced during the build phase, we note several trends. We plot the number of functionality

and security commits each day for several notable teams in Figure 3.7.

As seen in Figure 3.7, the team that had the fewest vulnerabilities, T1, did no security work on the very last day of the build phase (day 9). T1, which also had no vulnerabilities at the conclusion of the build phase, started implementing their access-control code on day 2 of the build phase and edited it slowly over the following 6 days. In contrast, T11, which had the fourth most vulnerabilities at the conclusion of the build phase, had security commits up until the last day and did not start implementing their access control code until day 6 of the build phase. They built the majority of the access control functionality on day 8. On day 8, they introduced 20 different vulnerabilities, only fixing about half of them before the conclusion of the build round (12).

T11 concluded the build round with 9 vulnerabilities in their code, 8 of which were introduced on day 8. Further we note that teams that waited to implement access control until late in the build phase often ran out of time to implement recursive delegation checks (T = 3). For example, T11 was still implementing basic access control on the last day of the build phase, causing them to run out of time to implement recursive delegation checks despite building this requirement into their design.

3.2.4.4 Good development practices can help with security, but do not make up for no design

We note some interesting ways in which good development practices seem to interact with the resulting code security. Teams T2 and T7 had a good design, but both waited until late in the build phase to add security to their codebase, rather than building it in incrementally.

T7 introduced 5 different vulnerabilities related to access control, 4 of which occurred on the same day. On the other hand, T9 provided only a minimal design document but built security in gradually over time, ending the build-phase with the second fewest vulnerabilities.

Despite this general trend, we can see that T8 seemingly did everything correctly. They started building for security early and continued steadily throughout the build phase. So what went wrong? The vulnerabilities introduced by T8 early in the build phase relate to poor design decisions or a lack of design. T8 introduced 9 different access-control-related vulnerabilities during the build phase, including 7 before day 6. These vulnerabilities were conceptual in nature, and they were slowly introduced over each small security commit. Building incrementally does not make up for not designing for security in depth. Other T8 vulnerabilities caused by a *Mistake* in implementation were introduced in the last 3 days of the build phase when T8 was likely rushing to complete their project.

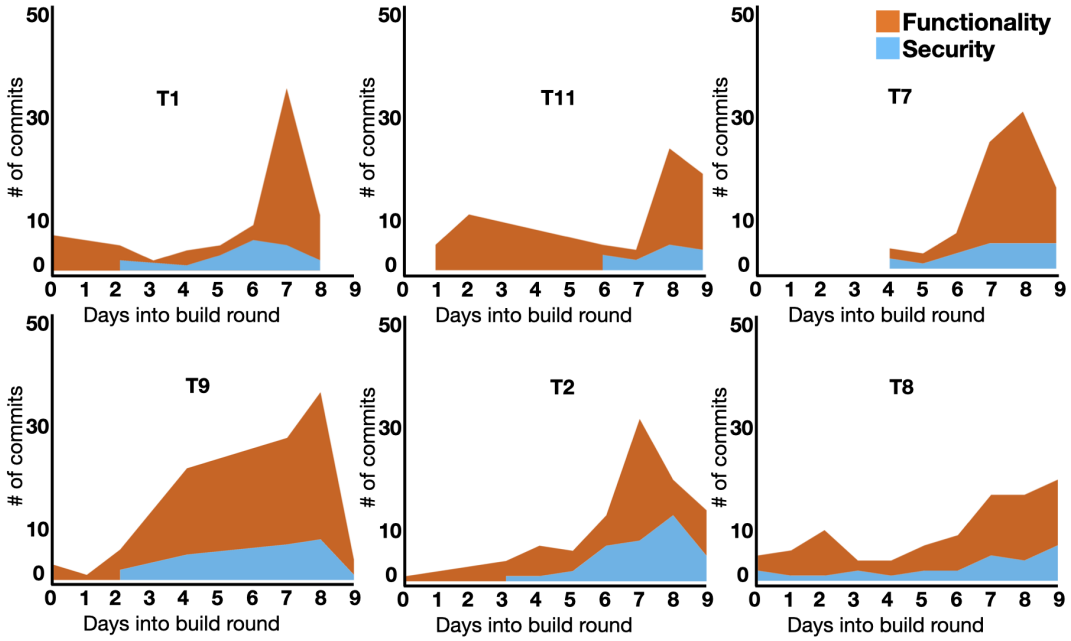

Figure 3.7: Number of functionality and security commits through the build phase for select teams.

3.3 Analysis of (Un)exploited Vulnerabilities

This section details the vulnerabilities that were (not) exploited during the break round. We use the same type descriptions in Section 3.2 to describe the exploits (not) uncovered.

Teams submitted 104 total valid breaks, resulting in 52 non-repeating breaks (where the breaking team did not exploit the same thing more than once againt the same team). Teams left 19 instances of vulnerabilities unfound and not exploited. When discussing vulnerabilities (not) exploited, we use E to represent the number of distinct vulnerabilities exploited, T to represent the number of teams that found this exploit, and U to represent the number of vulnerabilities unexploited. We note that one team can exploit the same vulnerability in multiple teams, as the same vulnerability can exist in multiple programs. For example, two teams might both fail to implement recursive delegation checks, so it is possible for E to exceed T. Throughout this section, we select illustrative examples within the three main vulnerability categories rather than discuss every specific type of vulnerability in detail, so E and U do not always sum to the total for *No Implementations*, *Misunderstandings*, and *Mistakes*.

3.3.1 *No Implementation*

Of the 29 distinct *No Implementation* vulnerabilities present after the build phase, 20 were exploited in the break phase. These related to missing some access control (E = 3, T = 5), not timing out (E = 3, T = 2), and missing recursive delegation checks (E = 14, T = 2).

3.3.1.1 Missing access control found when checking related issues

In general, teams exploited these missing access control and recursive delegation checks while testing a tangential, but related, access control requirement, rather than via targeted attacks where they reviewed code for a specific vulnerability. For example, T8 exploited a specific missing access control check on a primitive command by testing the target code for a vulnerability related to circular delegation, a tangential access control requirement. They were not specifically testing to see if the target team failed to implement access control on this primitive command. Rather, they were testing for a different access control requirement and found this vulnerability. However, breaks exploiting a lack of timeout were targeted: discovered explicitly by testing timeouts. For example, T3 exploited two timeout vulnerabilities by "testing [malformed user-scripts] and seeing if that caused [programs] to hang."

3.3.1.2 Teams target more glaring issues rather than complex vulnerabilities

Nine *No Implementation* vulnerabilities were left unexploited, pertaining to missing some access control checks (U = 4) and missing recursive delegation checks (U = 5). Notably, of the 9 unexploited *No Implementation* vulnerabilities, none were the *All Intuitive* no timeout vulnerability. The no recursive delegation vulnerabilities that were left unexploited were in code that had other, more glaring issues such as problems with input handling or timing out; teams generally favored attacking these issues rather than the slightly more complex vulnerabilities. As breaks that exploited missing access control checks were often found incidentally while targeting other vulnerabilities, these vulnerabilities were likely left unexploited because they were not accidentally triggered. Teams favoring glaring issues rather than attacking more complex issues

may be an artifact of the study, as teams knew that the developers of the code were students and the code was likely to contain bugs. However, we expect that code review and testing in the software development lifecycle also commonly target more obvious problems, and we incentivized specific targeting by giving more points for the discovery of novel bugs.

3.3.2 *Misunderstanding*

There were 25 distinct *Misunderstanding* vulnerabilities present after the build phase, and 17 were exploited in the break phase. The exploited vulnerabilities included incorrect access control assumptions (E = 13, T = 5) and incorrect assumptions about possible user script inputs (E = 4, T = 6). Specifically, vulnerabilities due to incorrect access control assumptions were overwhelmingly due to a misunderstanding of requirements associated with recursive delegation checks, where teams attempted to implement the recursive check but misunderstood a sub-piece or conceptual piece of the requirement (E = 6, T = 4).

3.3.2.1 *Misunderstanding* vulnerabilities were exploited with targeted testing

Breaks exploiting these *Misunderstandings* were often crafted to test for a specific *Misunderstanding* rather than testing for a broader, related requirement or being found incidentally. For example, T4 tailored their break to test "that there are no availability issues when creating/deleting a circular chain" rather than broadly attacking access control or recursive delegation checks.

Eight *Misunderstanding* vulnerabilities were left unexploited, mostly related to incorrect access control assumptions (U = 7). These vulnerabilities were related to rules around the delegation abilities and restrictions of the running principal of the user script (U = 3), rights

management of special principals (U = 2), managing rights when the recursive delegation chain is broken (U = 1), and conceptual misunderstandings of rights management for specific primitive commands (U = 1). Given exploiting *Misunderstandings* requires deeper knowledge and specifically crafted exploits, it is perhaps unsurprising that no single break submitted targeted these vulnerabilities in any project.

3.3.3 *Mistake*

Of the 25 *Mistake* vulnerabilities present at the end of the build round, 22 were exploited in the break round making vulnerabilities due to a *Mistake* the most likely to be exploited. The exploits of vulnerabilities from the build round were attributed to insufficient error checking (E = 13, T = 7), teams skipping a step when implementing security or functionality (E = 6, T = 8), mistakes associated with the control flow of the application (E = 2, T = 5), and an uncaught runtime server error (E = 1, T = 3). This mirrors results from prior work [20], in which we found that *Mistake* vulnerabilities are almost always exploited.

3.3.3.1 *Mistake* vulnerabilities were exploited incidentally

Teams exploited nearly all *Mistake* vulnerabilities incidentally, while targeting an unrelated vulnerability. T6 exploited a control flow issue in a team's code, in which they forgot to remove debugging statements, while attacking the implementation of transitive delegation in a few teams' codebases. That *Mistake* vulnerabilities were widely caught incidentally using high-level, broad testing points to the ability for fuzzers to uncover these vulnerabilities during the development phase. Teams only needed to test for basic functionality, akin to the testing performed by fuzzers,

to uncover these *Mistakes*. Several teams did not comprehensively test in the build phase, likely due to time constraints, but were able to build a set of comprehensive tests during the break phase, uncovering many vulnerabilities in other teams' code (T = 4). This suggests that with sufficient time and effort, developers could test for and uncover most *Mistake* vulnerabilities even with minimal security training. This suggests that in principle developers could use (and generate seeds for) tools like fuzzers, if the tools were sufficiently available and usable.

Only 3 *Mistake* vulnerabilities went unexploited. Two were vulnerabilities resulting from control flow issues in teams' programs and one was from a team skipping a step when implementing security checks. Compared to the exploited control flow mistakes and skipping steps, the unexploited mistakes were buried more deeply within teams' code and required very specific tests to exploit.

3.4 Analysis of Fixed Vulnerabilities

This section describes the details and characteristics of vulnerabilities that were fixed and not fixed. As in the previous section, we use the vulnerability type descriptions from Section 3.2.

In total, our participants fixed 66 vulnerabilities in their code during the build round (T = 12) and 30 vulnerabilities in their code during the fix round (T = 11). 39 vulnerabilities were left unfixed (19 exploited) at the study's conclusion. When discussing the (un)fixed vulnerabilities throughout this section, we use V to represent the number of vulnerabilities and T to represent the number of teams that this issue was (not) fixed by. As before, we select illustrative examples for the three main vulnerability categories, such that V does not always sum to the total for *No Implementations*, *Misunderstandings*, and *Mistakes*.

3.4.1 *No Implementation*

During the build phase, *No Implementation* vulnerabilities were fixed the most (V = 43, T = 11). Most were related to a lack of implemented access control (V = 22, T = 7). Several teams started by building their code focused on functionality rather than access control (T = 7) and added access control further into development. Generally, these teams started by adding basic access control and then revisited their code to add checks of the delegation chain (V = 15, T = 6). Five of the six teams that addressed recursive delegation checks did so within the last 2 days of the build phase. *No Implementation* vulnerabilities were largely introduced and fixed as teams worked through the build phase to implement the requirements of the specification, starting with *All Intuitive* requirements and getting to *Unintuitive* requirements if there was time left.

3.4.1.1 *No Implementation* fixes require restructuring the program

While many *No Implementation* vulnerabilities were fixed during the build phase, over half of them were left unfixed at the conclusion of the study (V = 16, T = 4). Of those left unfixed, seven were exploited during the break phase. The unfixed vulnerabilities were caused by missing access control checks (V = 2, T = 1), missing recursive delegation check (V = 3, T = 3), and missing a timeout in the code (V = 3, T = 3). Three of these vulnerabilities were associated with T12, including the missing recursive delegation check. T12 did not implement their access control checks with transitive rights in mind, failing to include recursive delegation checks and implementing many access control checks throughout the codebase rather than in one single access control function. Despite exploitation, this vulnerability remained unfixed within their code, as a fix for this would have required T12 to significantly alter their codebase, changing each

access control check throughout their codebase and redoing their entire access control system to account for parent-child rights. The two other teams with this vulnerability present in their code at the conclusion of the study required significant alterations to their codebase to address this vulnerability owing to the fact that their access control was spread throughout their codebase. This points to the importance of designing in depth for access control requirements from the outset, as designing in detail from the start prevents heavy redesign to address issues later.

3.4.2 *Misunderstanding*

During the build phase, vulnerabilities caused by *Misunderstanding* access-control requirements were the least likely to be fixed (V = 9, T = 6). The only *Misunderstanding* vulnerabilities fixed in this phase were due to incorrect access control assumptions (V = 9, T = 6), such as handling on special principals (V = 3, T = 2) and requirements around who can change a principal's password (V = 1, T = 1). Similarly, these were the most likely to be fixed *Misunderstandings* in the fix round (V = 10, T = 7).

3.4.2.1 *Misunderstanding* vulnerabilities are typically only fixed when pointed out

Vulnerabilities caused by *Misunderstanding* the security requirements were often not found and fixed until pointed out by either instructor-provided tests (during the build phase) or submitted exploits against a team's codebase (during the break phase). For example, T6 misunderstood the requirements for parent child rights, but found and fixed this vulnerability during the build phase because it "didn't run on a test."

However, instructor-provided tests only covered fairly basic functionality and failed to test for more complex access-control requirements. As a result, more complex *Misunderstandings* of access control were often not found until they were exploited in the break phase. For example, T9 had a vulnerability in their code involving the delegation of rights. When a user script is submitted to the system, the script starts by logging in as a principal. This principal (PA) may delegate rights to another principal (PC) through the use of a third principal (PB). This is not a problem as long as the permissions of PB are checked and they have the necessary permissions. However, T9 misunderstood this requirement and only checked the permissions of the running principal (PA). This vulnerability was not fixed by this team until it was exploited during the break round, despite the fact that T9 directly altered the code containing this vulnerability twice before the conclusion of the build round. Receiving detailed input about security misunderstandings in their code allowed T9 to address this issue and understand where they went wrong. Overwhelmingly, *Misunderstanding* vulnerabilities were fixed once they were pointed out and explained to teams (V = 13, T = 8), pointing to the benefit of including explanation of security *Misunderstandings* in the development process. Teams demonstrated the ability to learn from these explanations by crafting tests for other teams based on what had been exploited in their own code. For example, T6 began testing other teams for a vulnerability related to incorrectly removing rights once a delegation chain had been broken after this *Misunderstanding* was exploited and fixed in their own codebase. They said they found the vulnerability because it was a "break [in my code] found by [an] enemy team."

3.4.2.2 Addressing *Misunderstandings* requires time

Similar to the build phase, vulnerabilities caused by a security requirement *Misunderstanding* were left unfixed the most at the conclusion of the study (V = 11, T = 9). Three of these vulnerabilities were exploited during the break phase. These exploited vulnerabilities were due to incorrect assumptions about the access control requirements (V = 3, T = 4). Teams did not design at all for these vulnerabilities (V = 3, T = 3), and two of these vulnerabilities were submitted in the last day of the break phase, leaving teams little time to address something they had not designed for.

3.4.3 *Mistake*

3.4.3.1 *Mistake* vulnerabilities are found and fixed with testing

During the build phase, vulnerabilities caused by an implementation *Mistake* were mostly left unfixed (V = 14, T = 6). The most frequently fixed were related to control-flow errors (V = 12, T = 5). These vulnerabilities were easy to find through testing, as seen by break teams exploiting them incidentally using a variety of tests.

Conversely, vulnerabilities related to insufficient error checking (V = 6, T = 4) and missing an implementation step (V = 3, T = 3) were the most fixed *Mistakes* in the fix phase. Teams that did not think to handle edge cases related to insufficient error checking or skipped algorithmic steps were, unsurprisingly, also unlikely to think to test for these cases while building. Five of these vulnerabilities were exploited in the last day of the break phase, leaving teams little time to address them despite how easy many fixes could be. For example, a vulnerability in T7's

code caused by insufficient error checking was exploited on the second-to-last day of the break phase. T7 had failed to check certain commands for syntactic validity, leading to a crash if these commands were improperly formatted. T7 already had a function to check for syntactic validity, so this vulnerability could have been fixed by adding a single line of code invoking this already-existing function in an additional place. However, this vulnerability went unfixed in the short available time window.

3.4.3.2 Addressing more complex vulnerabilities leaves little time to address simple *Mistakes*

Eleven *Mistake* vulnerabilities were unfixed in our participants' code (V = 12, T = 6) at the conclusion of the study. Nine of these *Mistakes* were exploited. These exploited vulnerabilities were divided among a vulnerability due to insufficient error checking (V = 6, T = 3) and skipping an algorithmic step while implementing security (V = 3, T = 3). Two of the teams with unfixed *Mistakes* had the most and second most successful breaks against them during the break phase leaving them with many issues to address with limited time. Additionally, two teams fixed vulnerabilities caused by a lack of timeout and recursive delegation check, which took considerable time to address. Despite the relative ease of addressing mistakes, teams may not have had time to get to them before the conclusion of the study.

3.4.3.3 Most unfixed vulnerabilities are easy to exploit

To explore the possible repercussions of unfixed vulnerabilities, we look at how attacking teams and members of the research team (with security experience) rated the difficulty to find and

exploit these unfixed vulnerabilities. Overall, we find that, of those that they did rate, teams rated many unfixed *Mistake* vulnerabilities as easy to find and exploit (Find = 7/7 and Exploit = 6/6). Given that *Mistakes* were the least unfixed of any vulnerability, this is not particularly concerning. However, vulnerabilities associated with *Misunderstandings* were also overwhelmingly rated as easy to find and exploit by our teams (Find = 8/12 and Exploit = 10/12). Our experts rated *Misunderstanding* vulnerabilities as less easy to find and exploit than participants (Find = 5/12, Exploit = 7/12), but this points to a possibly alarming trend in which vulnerabilities that are difficult to fix and often weren't fixed until exploited are an easy point of exploitation for possible attackers. Once the vulnerability was discovered, it was easy to craft an exploit. For example, T9 had an unfixed vulnerability associated with removing rights transitively. An exploit for this could easily be crafted in two user-issued scripts, whereas the fix would require rewriting the access control functionality of the system, meaning the effort required from an attacker is minimal, but addressing the vulnerability requires significant changes to the design and implementation of the system.

3.5 Discussion and Recommendations

Our results emphasize the importance of existing recommendations and solutions.

3.5.1 Vulnerability classes differ in more than just content

Throughout our data we note that the different types of vulnerabilities (*No Implementation*, *Misunderstanding*, and *Mistake*) fall into distinct classes. These vulnerabilities occur differently from one another: *No Implementations* result from a failure to realize that a feature was needed

whereas *Mistakes* result from a failure to thoroughly test the codebase. These vulnerabilities are found differently from one another: *Misunderstandings* were found through explicitly crafted tests whereas *Mistakes* were found incidentally, while testing for a completely unrelated vulnerability. Each of these vulnerabilities needs their own strategy to prevent and address: thorough, well thought out design to address *No Implementations*, consultation and assistance from security experts to address *Misunderstandings*, and broad, complete testing to address *Mistakes*. Employing a single secure development strategy is not sufficient enough to prevent all three classes of vulnerabilities, and a variety of strategies and solutions will be necessary to promote secure development. Throughout the discussion, we'll talk in detail about solutions for the three different classes of vulnerabilities.

3.5.2 Importance of using security tools and comprehensive testing to uncover *Mistakes* confirmed

Nearly every team had at least one *Mistake* vulnerability at the conclusion of the build round (many teams more than one). These vulnerabilities were nearly all exploited during the break round, often incidentally through testing for other vulnerabilities, and were largely attributed to teams not testing for or considering an edge case or null value. These vulnerabilities can generally be caught or prevented by existing secure development tooling such as secure programming languages, fuzzers, and static analyzers [10, 11, 127–134].

Teams often struggled to thoroughly test their code for vulnerabilities during the build round but comprehensively tested other teams' code during the break round, finding and exploiting a variety of vulnerabilities. This points to a lack of priority for testing their own code thoroughly,

but not a lack of knowledge to do so. Possibly if teams were given more time to complete their project, they would have used this time to perform thorough testing on their own codebase. This highlights that developers likely do not lack the knowledge to thoroughly test their code, given the time and encouragement to do so.

3.5.3 Importance of incremental development and minimally trusted code reaffirmed

Our results suggest the importance of following several security best practices. Teams that worked incrementally on security and built for security and functionality simultaneously, rather than focusing on functionality and then security, introduced less vulnerabilities in their code throughout the build round. Notably, teams with a good design that waited to implement security had more vulnerabilities than some teams with minimal or no design that started early on security. However, starting early on security did not make up for conceptual misunderstandings of security requirements.

Additionally, how teams built their security code played an important role. We note that teams that failed to fix vulnerabilities related to the transitivity of the access control system had individual access control checks for each command rather than building one function. This meant that when these teams wanted to fix this vulnerability they had to redesign their entire system. Moreover, every team that failed to handle the transitive nature of the access control system implemented their access control checks in this way. This fails to follow the security recommendation of minimal trusted code [135] and points to the benefit of of following best practices in implementing security code.

3.5.4 Detailed design is important to secure development, but not a silver bullet

Design appeared to play an important part in the security of our participants' code. More broadly, we found that teams with a detailed design were less likely to miss implementing access control and make mistakes. Teams that misunderstood a sub-case or sub-requirement of an access control check did not design for the requirement at all. However, detailed design isn't a silver bullet. If teams encoded an initial *Misunderstanding*, they aren't likely to catch the vulnerability at implementation as they were likely to stick with their designs. A detailed design is important to the secure development process as it helps prevent missing security requirements or introducing programming mistakes, but it is important to ensure the design is secure from the start.

3.5.5 Importance of including security experts in development lifecycle to help uncover *Misunderstandings*

Our results suggest that participants struggle more with misunderstanding security concepts than implementing them. This echoes prior work [20] and points to a need for security expertise within the development process. Teams encoded these misunderstandings into their design documents and failed to catch them at implementation time. They often attributed these vulnerabilities to a fundamental misunderstanding of the security requirements, and they were unable to understand them and address them until they were pointed out and explained to them (exploited during the break round). Once the breaking team pointed out and explained the vulnerability, *Misunderstandings* were almost unanimously fixed. The ability to have the vulnerability explained to them allowed teams to address these issues, improving the resulting security of the code. The presence of

a security-knowledgeable developer or security professional during the design process to help point to these vulnerabilities and explain their importance and severity can improve the security knowledge of developers, while simultaneously improving the security of the codebase.

Chapter 4: Exploring the Adoptability of Secure Programming Languages

Given the importance of secure tooling in preventing and discovering vulnerabilities, we next explore the barriers and benefits to adopting secure tools: specifically, the memory-safe programming language Rust.

Many current, real-world vulnerabilities arise from highly dangerous violations of memory safety, such as use-after-frees, buffer overflows, and out-of-bounds reads/writes [136–140]. Despite their long history and the many attempts aimed at mitigating or blocking their exploitation, such vulnerabilities have remained a consistent, and sometimes worsening, threat [141], with estimates that 60-70% of critical vulnerabilities in Chrome [142], Microsoft products [143] and in other large critical systems [144] owe to memory safety vulnerabilities.

Overwhelmingly, memory safety vulnerabilites occur in C and C++ code—while most popular languages enforce memory safety automatically, C and C++ do not [23, 145]. Relatively recently, Google developed Go [11] and Mozilla developed Rust [10] to be practical but secure alternatives to C and C++; these languages aim to be fast, low-level, *and* type- and memory-safe [146, 147]. Rust and Go have been rising in popularity—IEEE's 2019 Top Programming languages list ranks them 17 and 10, respectively—but C and C++ continue to occupy top spots (3 and 4). We might wonder: What are the factors fueling the rise of these secure languages? Is there a chance they will overtake their insecure counterparts, C and C++, and if so, how?

In this chapter, I, along with my co-authors, attempt to answer these questions for Rust, in particular. While Go is extremely popular, Rust's popularity has also risen sharply in the last few years [146, 148–151]. Rust's "zero-cost abstractions" and its lack of garbage collection make it appropriate for resource-constrained environments, where Go would be less appropriate and C and C++ have traditionally been the only game in town.

We conducted semi-structured interviews with professional, primarily senior developers who have actively worked with Rust on their product teams, and/or attempted to get their companies to adopt Rust (n = 16). We also surveyed participants in Rust development community forums (n = 178). We asked participants about their general programming experience and experiences using and adopting Rust both personally and at a company. We also asked about the benefits and drawbacks of using Rust in both settings. By asking these questions, we aim to understand the challenges that inhibit adoption, the net benefits (if any) that accrue after adoption, and what tactics have been (un)successful in driving adoption and use.

The rest of this chapter is organized as follows: Section 4.1 reviews necessary background information about the Rust Programming Language; Section 5.1 describes the interview and survey protocol, recruitment strategies, and data analysis approaches; Section 4.3 describes the demographics of our interview and survey participants; Section 4.4 describes the experiences of our participants using Rust; Section 4.5 discusses the benefits and drawbacks of Rust as experienced by our participants; Section 4.6 discusses organizational adoption of Rust, Section 4.7 discusses the implications of our work as it pertains to the promotion and adoption of secure programming languages.

4.1 Background

Rust is an open-source systems programming language created by Mozilla, with its first stable release in 2014. Rust's creators promote its ability to "help developers create fast, secure applications" and argue that Rust "prevents segmentation faults and guarantees thread safety." This section presents Rust's basic setup and how it aims to achieve these benefits. For those with a deeper interest, we recommend the tutorial offered in the official Rust Programming Language Book [152].

4.1.1 Core features and ecosystem

Rust is a multi-paradigm language, with elements drawn from functional, imperative, and object oriented languages. Rust's **traits** abstract behavior that types can have in common, similarly to interfaces in Java of typeclasses in Haskell. Traits can be applied to any type, and types need not specifically mention them in their definitions. Objects can be encoded using traits and structures. Rust also supports **generics** and **modules**, and a sophisticated **macro system**. Rust's variables are **immutable by default**: once a value is bound to a variable, the variable cannot be changed unless it is specifically annotated as mutable. Immutability eases safe code composition, and plays well with ownership, described shortly. Rust also enjoys **local type inference**: types on local variables are generally optional, and can be inferred from their initializer. Rust also supports **tagged unions** ("enums") and **pattern matching**, which allow it to, for example, avoid the need for a *null* value (the "billion dollar mistake" [153]).

Rust has an integrated build system and package manager called **Cargo**, which downloads library packages, called **crates**, as needed, during builds. Rust has an official community package

registry called crates.io. At the time of writing, Crates.io lists more than 49,000 crates.

4.1.2 Ownership and Lifetimes

To avoid dangerous, security-relevant errors involving references, Rust enforces a programming discipline involving ownership, borrowing, and lifetimes.

4.1.2.1 Ownership

Most type-safe languages use garbage collection to prevent the possibility of using a pointer after its memory has been freed. Rust prevents this without garbage collection by enforcing a strict *ownership*-based programming discipline involving three rules, enforced by the compiler:

1. Each value in Rust has a variable that is its ***owner***.
2. There can only be ***one owner at a time*** for each value.
3. A value is ***dropped*** when its ***owner goes out of scope***.

An example of these rules can be seen in Listing 4.1. In this example, `a` is the owner of the value "example." The scope of `a` starts when `a` is created on line 3. The scope of `a` ends on line 5, so the value of `a` is then dropped. In the second block of code, `x` is the initial owner of the value "example." Ownership is then transferred to `y` on line 11, which is why the print on line 13 fails. The value cannot have two owners.

4.1.2.2 Borrowing

Since Rust does not allow values to have more than one owner, a non-owner wanting to use the value must *borrow* a reference to a value. A borrow may take place so long as the

```
{
//make a mutable string and store it in a
let mut a = String::from("example");
a.push_str(", text"); //append to a
}
//scope is now over so a's data is dropped

{
//make a mutable string and store it in x
let x = String::from("example");
let y = x; //moved ownership to y
println!("y is {}", y); //allowed
println!("x is {}", x); //fails
}
```

Listing 4.1: Examples of how ownership works in Rust

following invariant is maintained: There can be (a) just one mutable reference to a value `x`, *or* (b) any number of *immutable* references to `x` (but not both). An example of the rules of borrowing can be seen in Listing 4.2. In this example, a mutable string is stored in `x`. Then, immutable references are made ("borrowed") on lines 6 and 8. However, the attempt to mutate the value on line 10 fails since `x` cannot be mutated while it has borrowed (immutable) references. Line 12 fails in the attempt to make a mutable reference: `x` cannot have both a mutable and an immutable reference. Once we reach line 13, the immutable references to `x` have gone out of scope and been dropped. `x` is once again the owner of the value and possesses a mutable reference to the value, so line 14 does not fail. In the second code block, starting on line 15, a mutable reference is made to the value. Attempts to make a second mutable reference on lines 19 and 21 fail because only one mutable reference can be made to a value at a time.

The ownership and borrowing rules are enforced by a part of the Rust compiler called the *borrow checker*. By enforcing these rules the borrow checker prevents vulnerabilities common to memory management in C/C++. In particular, these rules prevent dangling pointer dereferences and double-frees (only a sole, mutable reference may be freed), and data races (a data race

```
{
//make a mutable string and store it in x
let mut x = String::from("example");
    {
    //make immutable reference to x
    let y = &x; //allowed
    //make second immutable reference to x
    let z = &x; //alllowed
    println!("x is {}. y is {}", x, y) //allowed
    x.push_str(", text"); //fails
    //make mutable reference to x
    let mut a = &mut x; //fails
    } //drops y and z; x owner again
x.push_str(", text"); //allowed
    {
    //make mutable reference to x
    let mut a = &mut x; //allowed
    a.push_str(", text"); //allowed
    x.push_str(", text"); //fails
    //make second mutable reference to x
    let mut b = &mut x; //fails
    } //drops a; x is owner again
}
```

Listing 4.2: Examples of how borrowing works in Rust

requires two references, one mutable).

Unfortunately, these rules also prevent programmers from creating their own doubly-linked lists and graph data structures. To create complex data structures, Rust programmers must rely on libraries that employ aliasing internally. These libraries do so by breaking the rules of ownership, using unsafe blocks (explained below). The assumption is that libraries are well-vetted, and Rust programmers can treat them as safe.

4.1.2.3 Lifetimes

In a language like C or C++, it is possible to have the following scenario: (1) you acquire a resource; (2) you lend a reference to the resource; (3) you are done with the resource, so you deallocate it; (4) the lent reference to the resource is used. Rust prevents this scenario using a

concept called *lifetimes*. A lifetime names a scope, and a lifetime annotation on a reference tells the compiler the reference is valid only within that scope. For example, the lifetime of variable `a` in Listing 4.1 ends on line 5 where the scope of `a` ends. Similarly, the lifetime of `a` in Listing 4.2 ends on line 22.

4.1.3 Unsafe Rust

Since the memory guarantees of Rust can cause it to be conservative and restrictive, Rust provides escape hatches that permit developers to deactivate some, but not all, of the borrow checker and other Rust safety checks. We use the term *unsafe blocks* to refer generally to unsafe Rust features. Unsafe blocks allows the developer to:

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable global variable
- Implement an unsafe trait
- Access a field of a union

Unsafe functions and methods are not safe in all cases or for all possible inputs. Unsafe functions and methods can also refer to code that a developer wants to call that is in another language. Unsafe traits refer to traits with at least one unsafe variant. Lastly, unions are like structs but only one field in a union is used at a time. To use unsafe blocks, the relevant code construct is labeled with keyword `unsafe`.

4.2 Method

To understand the benefits and drawbacks to adopting Rust, we conducted semi-structured interviews with senior and professional software engineers working at technology companies who were using Rust or attempting to get Rust adopted. To examine the resulting findings in a broader ecosystem, we then distributed a survey to the Rust community through various online platforms.

4.2.1 Interview protocol

From February through June 2020, we conducted 16 semi-structured interviews via video-conferencing software.

Each interview included two phases. Phase one asked the participants how they discovered and learned Rust, as well as about instances when they or their company decided to use Rust for projects (or not) and why. In the second phase, we asked more technical questions about programming with Rust, including what features of the language/ecosystem they like/dislike compared to other familiar languages, as well as opinions about features relating to Rust's security, such as ownership and unsafe blocks.

Each session lasted about an hour, giving participants a chance to share detailed experiences. The full interview protocol is given in Appendix A.1.

4.2.2 Survey

The survey was designed to mirror the interviews, with closed-item answer choices inspired by answers from the open-ended interview questions. The survey was broken into seven sections;

Sect.	Description and Example Questions
1	Technical Background (General, and Rust) • How long have you been programming? • How long have you been programming in Rust?
2	Learning and using Rust • How easy or difficult did you find Rust to learn? • How would you rate the quality of available Rust docs? • When I encounter a problem or error while working in Rust, I can easily find a solution to my problem?
3	Work (general), and using Rust for work • Did anyone at your employer have apprehensions about using Rust? • What one piece of advice would you give to someone who is trying to get Rust adopted?
4	Comparing Rust to other familiar languages • How would you rate the quality of Rust compiler and run-time error messages compared to [*chosen language*]?
5	Rust likes/dislikes & unsafe blocks • Which of the following describes your use of unsafe blocks while programming in Rust?
6	Porting and interoperating with legacy code • What language(s) have you ported from?
7	Demographics about participants • Please select your highest completed education level

Table 4.1: Survey sections and example questions.

Table 4.1 tabulates the sections and provides some example questions. The full survey is given in Appendix A.2. It was active from July to September 2020.

4.2.3 Recruitment

To recruit for the interviews, we contacted a longtime member of the core Rust team and asked them to connect us with software engineers who were active members or leaders of teams

using or adopting Rust at their employers. From these initial referrals, we snowball-sampled more interviewees (asked participants to refer us to peers). We also recruited participants referred to us by colleagues, and contacted people and companies quoted or listed on the Rust website [154]. We focused on recruiting participants with senior, leadership, or other heavily involved roles in the Rust adoption process. We interviewed participants until we stopped hearing substantially new ideas, resulting in a total of 16 participants. This sample size aligns with qualitative best practices [155].

To recruit participants for the survey, we advertised on several Rust forums and chat channels: Reddit channel `r/rust`; Rust Discord community channels `embedded`, `games-and-graphics`, `os-dev`, `gui-and-ui`, and `science-and-ai`; Rust Slack beginners channel; Rust Facebook Group; and the official Rust Users Forum. Those who wanted to participate were directed to follow a link in the notice that took them directly to the survey.

4.2.4 Ethics

Both the interview and the survey were approved by University of Maryland's ethics review board. We obtained informed consent before the interview and the survey. Given that we were asking questions about their specific companies and the work they were doing, participants were informed that we would not disclose the specific company they worked for. They were reminded that they could skip a question or stop the interview or survey at any time if they felt uncomfortable.

4.2.5 Data analysis

Once the interviews were complete, two team members transcribed the audio recordings and then analyzed them using iterative open coding [156]. The interviewer and the other team member independently coded the interviews one at a time, developing the codebook incrementally and resolving disagreements after every transcript. This process continued until a reasonable level of inter-rater reliability was reached measured with the Krippendorff's α statistic [157]. After seven interviews, the two researchers achieved a Krippendorff's $\alpha = 0.80$, calculated using ReCal2 [158]. This level of agreement is above the commonly recommended thresholds of 0.667 [112] or 0.70 [159] for exploratory research and meets the more general minimum threshold recommended by Krippendorff of 0.8 [157]. Once a reliable codebook was established, the remaining nine interviews were evenly divided among the two researchers and coded separately.

We report the results of our closed-response survey questions using descriptive statistics. While we did not have any questions as specific attention checks, we evaluated the responses for completeness to ensure that we removed all low-quality responses. We did not remove many responses as can be seen in Section 4.3. Since our work is exploratory, we did not have any hypotheses, so we do not make any statistical comparisons. Free response questions from the survey were analyzed by one researcher using the same codebook from the interview. When new codes were added to the codebook, they were back-applied to the interviews.

Throughout the following sections, we use *I* to indicate how many interview participants' answers match a given statement, and use *S* to denote how many survey participants' answers do, either as a percentage (closed-item questions) or count (open-ended ones). We report on interview and survey results together, as the results generally align. We report participant counts

from the interviews and open-ended items for context, but not to indicate broader prevalence. If a participant did not voice a particular opinion, it does not necessarily mean they disagreed with it; they simply may not have mentioned it.

4.2.6 Limitations

Our goal with the interviews was to recruit people who had substantial experience, and preferably a leadership role, in attempting to adopt Rust at a company or team. We believe we reached the intended population. Only one interviewee failed to see Rust adopted at their employer, but all interviewees faced similar adoption challenges.

For the surveys, our goal was to reach a broad variety of developers with a range of Rust experiences, in order to capture the widest range of benefits and drawbacks. We did reach participants with a wide range of Rust experience, in part because we targeted many Rust forums, including some specifically for beginners. However, because all of these forums are about Rust, we may not have reached people who have tried Rust but abandoned it, or those who considered it but decided against it after considering potential pros and cons. In addition, these forums are likely to overrepresent Rust enthusiasts compared to those who use the language because they are required to. Further, there could be self-selection bias: because we stated our goal of exploring barriers and benefits to adopting Rust when recruiting, those with particular interest in getting Rust adopted may have been more likely to respond.

Taken together, these limitations on our survey population suggest that our results may to some extent overstate Rust's benefits or may miss some drawbacks that drive people away from the language entirely. Nonetheless, our results uncovered a wide variety of challenges and

benefits that provide novel insights into the human factors of secure language adoption. Given the general difficulty of recruiting software developers [160], and the particular difficulty of reaching this specific subpopulation, we consider our sample sufficient.

4.3 Participants

This section describes the experience, job titles, and organizational structure of our interview and survey participants.

4.3.0.1 Interview participants

We interviewed 16 people who were active members of teams using or adopting Rust at their company. Our participants mostly held titles related to software development (I = 12) and worked at large technology companies (more than 1000 employees, I = 9), as shown in Table 4.2. Most of them had worked in software development for many years and were members of, several leading, teams building substantial project(s) in Rust at their employers. Many were Rust evangelists at their companies. Their companies develop social media platforms, bioinformatics software, embedded systems, cloud software, operating systems, desktop software, networking software, software for or as research, and cryptocurrencies.

4.3.0.2 Survey respondents

We received 203 responses to our survey. We discarded 25 (12%) incomplete surveys, which left 178 complete responses. Respondents were predominantly male (88%), young (57% below the age of 30 and 88% below the age of 40), and educated (40% had a bachelor's degree

ID	Title	Company Size (# employees)
I1	aid in Rust adoption	≥ 1000
I2	group director	100 - 999
I3	software engineer	≥ 1000
I4	software engineer	≥ 1000
I5	senior engineer	100-999
I6	principal software engineer	≥ 1000
I7	system engineer	≥ 1000
I8	software engineer	≥ 1000
I9	co-founder and CTO	< 100
I10	instrumentation engineer	100 - 999
I11	engineering manager	≥ 1000
I12	research software engineer	100 - 999
I13	research software engineer	100 - 999
I14	software engineer	≥ 1000
I15	principal engineer	< 100
I16	software engineer	≥ 1000

Table 4.2: Interviewee demographics.

and 28% had a graduate degree). Our participants were relatively experienced programmers (53% had more than 10 years of programming experience and 85% had at least 5 years). Seventy-two percent of our participants were currently employed in the software engineering field and worked in a variety of application areas, as shown in Table 4.3. Additionally, our participants had used a variety of languages in the prior year, as shown in Figure 4.1.

Our survey participants were fairly experienced at programming in Rust (37% had been programming in Rust at least 2 years and 74% at least 1 year). Ninety-three percent of respondents had written at least 1000 lines of code and 49% had written at least 10,000 lines of code. Forty-six percent had only used Rust for a hobby or project, 2% had only used it in a class, 14% had maintained a body of Rust code, and 38% had been paid to write Rust code. Most of our respondents were currently using Rust (93%), while some had used it on projects in the past but were not currently using it (7%). While our survey participants had a broad variety

Area	# of participants (%)
Web development	73 (54%)
Library development	51 (38%)
Network programming	44 (32%)
DevOps	33 (24%)
Databases programming	28 (21%)
Data science	27 (20%)
Embedded systems programming	27 (20%)
Desktop/GUI apps development	26 (19%)
OS programming	25 (18%)
Other	24 (18%)
Mobile application development	19 (14%)
Firmware development	16 (12%)
CS/Technical research	14 (10%)
Game development	9 (7%)
CS/Technical education	7 (5%)

Table 4.3: Survey participants who worked in each area of software development. Multiple selection was allowed.

of experiences, they may underrepresent people who tried and turned away from Rust—such people would probably not be members of the Rust forums in which we advertised. As such, our results may overstate Rust's benefits or may miss some drawbacks that drive people away from the language entirely. Nevertheless, we believe our results offer practical insights relevant to the adoption of secure programming languages.

4.3.0.3 Survey respondents' companies

Nearly half of our respondents were using Rust for work (49%). Of those using Rust for work, most were using Rust as a part of a company or large organization (84%), rather than as a freelance assignment. We gathered further details about these 87 respondents' companies. They were primarily small (53% of the 87 worked for companies with 100 or fewer employees and 74% worked for a company with less than 1000 employees). They mostly developed alone (50%) or in small teams of two to five people (40%) at their companies, and their companies had

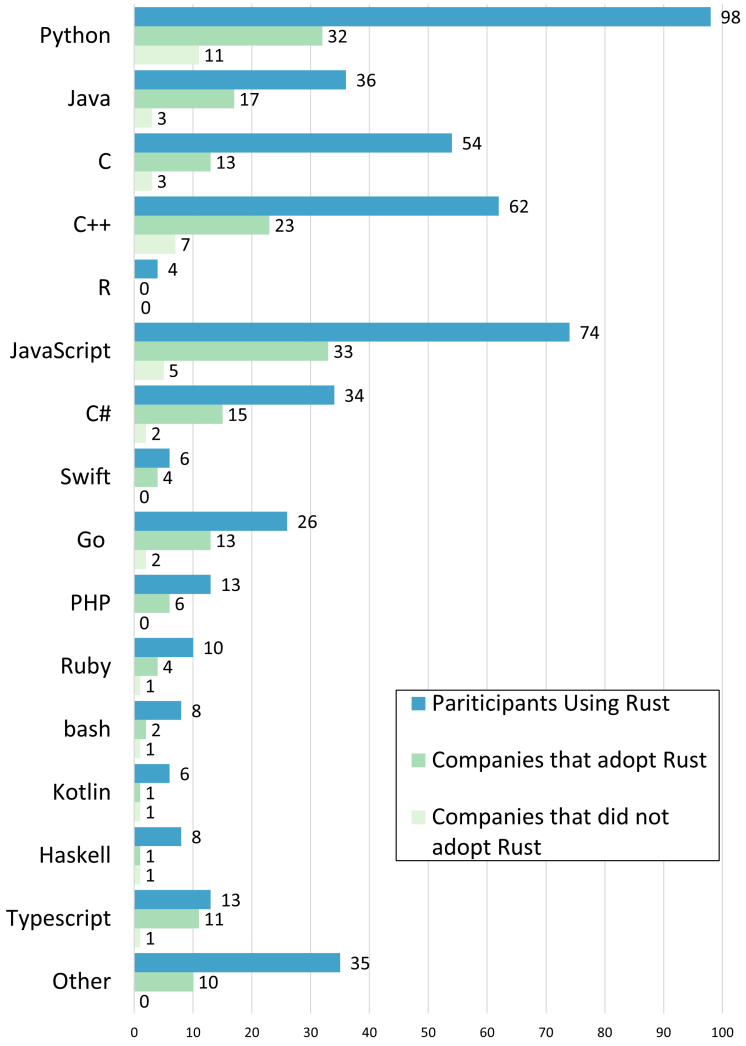

Figure 4.1: Languages used in the past year by survey participants (counts), companies that had adopted Rust, and companies that considered but didn't adopt Rust. Ordered according to the IEEE 2019 top programming languages list [161].

legacy codebases of varying sizes (88% had 500,000,000 or fewer lines of code and 64% had 1,000,000 or fewer lines of code). A variety of languages were used at respondents' companies (whether they had adopted Rust or not), as shown in Figure 4.1.

4.4 How is Rust being used?

This section and the next two analyze our interview and survey results. We first examine how our participants are using Rust.

4.4.1 Applications

Interview participants reported using Rust in a variety of application areas, including databases (I = 3); low-level systems such as operating systems, device drivers, virtual machine management systems, and kernel applications (I = 5); data processing pipelines (I = 1); software development applications such as monitoring resource usage (I=2); and compilers and programming languages tools (I = 2).

Participants did not always consider Rust the best tool for the job. When asked to select application areas for which Rust is not a strong fit, they most frequently mentioned mobile (I = 1, S = 44%), GUI (I = 3, S = 37%), and web applications (I = 3, S = 17%). For example, I9 said "*Strongly typed languages like Rust... lend themselves much more to systems programs... and less to web applications and things that you want to be very flexible.*" Interestingly, 13 survey participants who selected web development as a bad fit for Rust also chose web development as one of the things they do for work. Several participants mentioned that Rust is a poor fit for prototyping or one-off code (I = 6, S = 3%). I4 explained, "*I still prototype everything in C++ because it just works faster ... [Rust's] not a great prototyping language.*"

4.4.2 Porting and interoperating

Because Rust is relatively new, using it often requires porting or interoperating with legacy code written in other languages.

Most participants had ported code from another language into Rust (I = 14, S = 69%). They had ported from a variety of languages (Figure 4.2). Interview participants found porting code from Python to be easy (I = 5); similarly, where 70% of survey respondents who had ported from Python (n=33) found it either somewhat or extremely easy. In contrast, fewer participants found porting from C (I = 2; S = 54%, n = 41) and C++ (I = 2; S = 52%, n = 44) somewhat or extremely easy. I11 said porting from C++ is "*much harder because... you structure your data with movability [mutability].*"

Many participants had written code to interoperate with Rust (I = 13, S = 47%), starting from a variety of languages (Figure 4.2). Ease of interoperation varied by language somewhat differently than ease of porting. Almost three-quarters of participants who had interoperated with C found it at least somewhat easy (I = 6; S = 70%, n = 44). A majority also rated Python somewhat or extremely easy (I = 2; S = 53%, n = 17). Less than half considered C++ at least somewhat easy (I = 2; S = 43%, n = 23). I6 attributes this to the fact that "*the C++ side is just the Wild West. There's rampant aliasing ... and none of that is going to play by Rust's rules.*"

4.4.3 Unsafe blocks

As described in Section 4.1, unsafe blocks allow the programmer to sidestep borrow-checking, which can be too restrictive in some cases. Because unsafe blocks may potentially compromise Rust's safety guarantees, we investigate how they are used and what if any error-

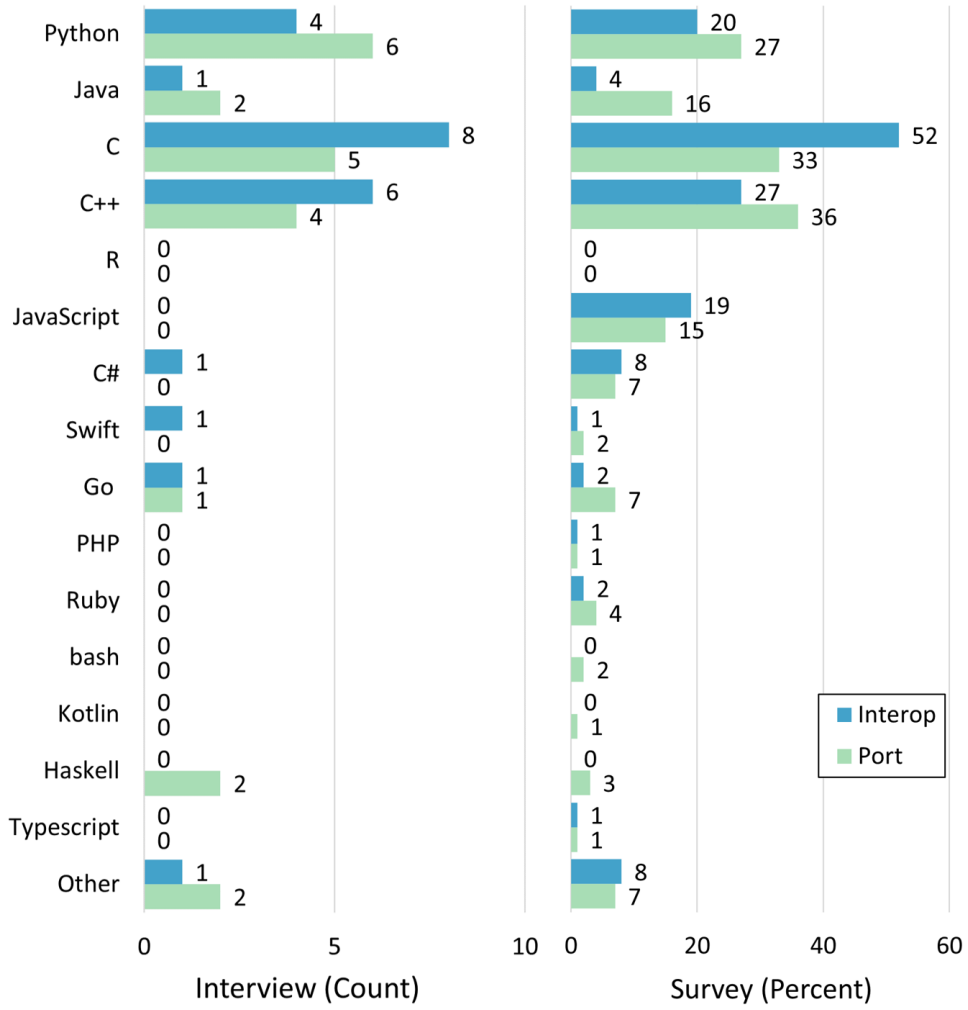

Figure 4.2: Interview and survey participants porting (survey n = 123) to Rust from and interoperating (survey n = 84) Rust with other languages. Languages are ordered via ranking on the IEEE 2019 top programming languages list [161].

mitigation strategies exist.

4.4.3.1 Unsafe blocks are common and have a variety of uses

Most participants had used unsafe blocks (I = 15, S = 72%). Use-cases included foreign-function interfacing (I = 11, S = 70%), increasing code performance (I = 3, S = 40%), kernel-level interaction (I = 1, S = 35%), hardware interaction (I = 4, S = 34%), and memory management (I = 4, S = 28%). For example, I14 uses unsafe blocks to "*wrap all of our... code for accessing*

hardware," since they had to do things like "*write values into this offset relative to the base address register,*" which is prohibited by Rust ownership rules.

4.4.3.2 Few companies have unsafe-code reviews

To avoid introducing problems Rust otherwise guarantees against, companies may implement a procedure to check that "unsafe" code is actually safe. However, unsafe-review policies were uncommon at our participants' employers (I = 7, S = 28%). Where specific policies do exist, the most common approach is a thorough code review (I = 2, S = 68%). For example, at S118's company, the review policy is "*pretty simple: pay extra close attention to unsafe blocks during code review.*" To help code reviewers, developers use comments to explain why the code is actually safe (I = 5, S = 21%). I5 commented, "*I guess the only formal thing is that every unsafe block should have a comment saying why it is in fact safe.*" These comments may aim to explain important safety invariants [162].

4.5 Benefits and drawbacks of Rust

This section explores benefits and drawbacks of Rust related to technical aspects of the language, learning the language, the Rust ecosystem, and Rust's effect on development.

4.5.1 Technical benefits of Rust

Participants largely are motivated by, and agree with, Rust's claims of performance and safety [146].

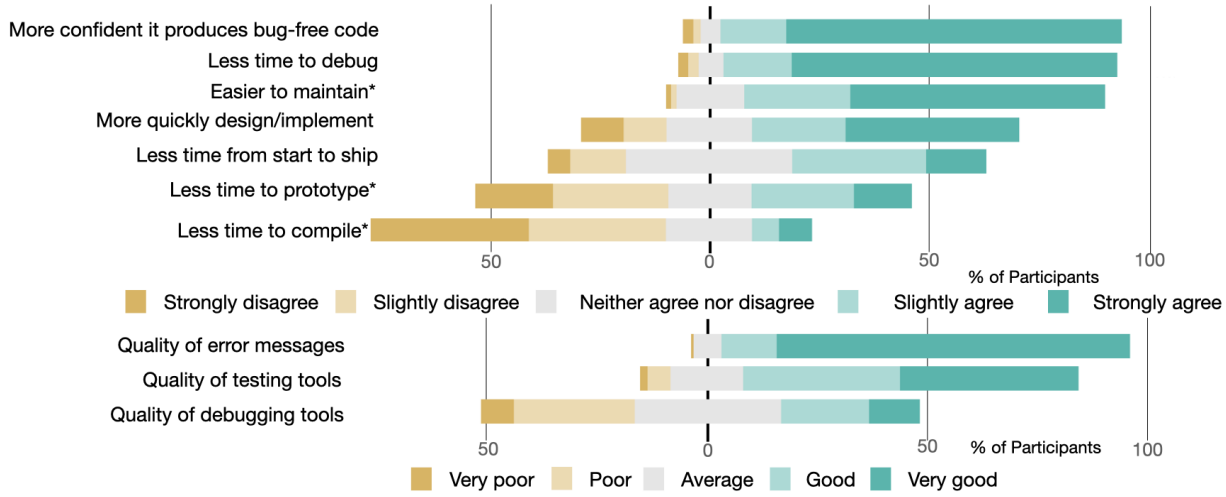

Figure 4.3: Likert-style responses comparing Rust to a language survey participants were most comfortable with. Green bars advantage Rust; gold bars advantage the other language. Questions with a * have been flipped in polarity for consistency.

4.5.1.1 Safety is important

Many participants identified Rust's safety assurances as benefits. They listed memory safety (I = 10, S = 90%), concurrency safety (I = 6, S = 84%), immutability by default (I = 4, S = 74%), no null pointers (I = 3, S = 81%), Rust's ownership model (I = 2, S = 75%), and lifetimes (I = 2, S = 55%). As I5 said about Rust's strengths, "*The safety guarantees, like 100%.... That's why I use it. That's why I was able to convince my boss to use it.*"

4.5.1.2 So is performance

Participants were also drawn to Rust's promise of high performance. Respondents explicitly listed performance (I = 7, S = 87%) and, less explicitly, lack of garbage collection (I = 3, S = 63%) as reasons to like the language. I1 describes the appeal of Rust: "*it gives you the trifecta of performance, productivity, and safety.*"

4.5.2 Learning Rust: Curiosity vs. reality

We next review participants' experiences learning Rust.

4.5.2.1 Most chose to learn Rust because it is interesting or marketable

Most participants selected, as their primary reason(s) to learn Rust, curiosity (I = 2, S = 90%). Other said they had heard about it online or it was suggested by a friend (I = 12, S = 25%). Participants also believed knowing Rust was a marketable or useful job skill (I = 7, S = 22%).

4.5.2.2 Rust is hard to learn

Possibly the biggest drawback of Rust is its learning curve. Most participants found Rust more difficult to learn than other languages (I = 7, S = 59%). I14 said Rust has "*a near-vertical learning curve.*"

Asked how long it took to learn to write a compilable program without frequently resorting to the use of unsafe blocks, a plurality of participants said one week to one month (I = 2, S = 41%), less than one week (I = 0, S = 27%), or one to six months (I = 3, S = 25%). Notably, six survey participants were not yet able to do this. Interviewees had similar experiences. I3 "*didn't feel fully comfortable with Rust until about three months in, and really solid programming without constantly looking stuff up until about like six months in.*" Five interviewees said it takes longer to get Rust code to compile than another language they are comfortable with. Survey participants agreed (S = 55%, Figure 4.3). S161 commented, "*You spend 3–6 months in a cave, breathing, eating and sleeping Rust. Then find like-minded advocates who are prepared to sacrifice their first born to perpetuate the unfortunate sentiment that Rust is the future, while*

spending hours/days/weeks getting a program to compile and run what would take many other 'lesser' languages a fraction of the time."

4.5.2.3 The borrow checker and programming paradigms are the hardest to learn

Seven interviewees reported that the biggest challenges in learning Rust were the borrow checker and the overall shift in programming paradigm. A few survey participants (S = 3) noted this in free-response as something they explicitly did not like about Rust. S136 did not like "*having to redesign code that you know is safe, but the compiler doesn't.*" I8 echoes this frustration: "*There are new paradigms that Rust sort of needs to teach the programmer before they can become super proficient. And that just makes the learning curve a little bit higher, and that did frustrate a number of people, . . . because it's something that's sufficiently different from other things they're used to.*" This could pose a problem for adoption, if the frustration of learning these new paradigms turns developers away from Rust altogether.

4.5.3 Rust ecosystem: Good and getting better

The Rust ecosystem influences organizational adoption, because it provides needed support for large projects. Participants identified a variety of current benefits and drawbacks.

4.5.3.1 Tools are easy to use and well supported, but slow

Asked how Rust's tooling compared to the other language they were most comfortable programming in, 75% of survey participants found it either very good or good (Figure 4.3). Also

in Figure 4.3, most survey participants found Rust's compiler and runtime error messages to be good or very good compared to their reference language (S = 92%). Beginners (less than one year of experience, n = 47) felt this way, too (S = 87%). A large majority (I = 8, S = 97%) listed the compiler's descriptive error messages as a major problem-solving benefit. I9 comments: "*Most of the time the compiler is very, very good at telling you exactly what the problem is.*" When it doesn't, "*Rust is an exercise in pair programming with the compiler,*" wrote S176. Participants also liked the crates ecosystem (I = 4, S = 83%). For example, I7 said, "*I also just love the cargo tooling; it's so easy to get crates.*" While participants like the tooling, they dislike that Rust has a long build time (I = 4, S = 55%). I16 said, "*Compile times are pretty bad... I don't think Rust will ever get close to like Go level of compile speed.*"

4.5.3.2 Easy to find solutions

Despite the challenging learning curve, participants report it is easy to find solutions to problems they encounter when developing in Rust (I = 14, S = 79% overall, 70% of beginners). Participants attribute this to good compiler errors (discussed above), good official documentation (I = 3, S = 91%), and the helpfulness of the Rust developer community, in-person and online (I = 5, S = 46%). I5 notes the "*very accessible documentation and kind of an active community... on Stack Overflow and so forth. I feel like if I have a problem with Rust, I Google it and there's always an answer.*"

4.5.3.3 Rust lacks libraries and infrastructure and causes dependency bloat

Despite the high quality of available tools and libraries, Rust still lacks some critical libraries and infrastructure, perhaps in part because it is fairly new. When asked what they dislike about the language, many participants noted the lack of available libraries (I = 3, S = 39%). I4 agrees:"*It feels like you're reinventing a lot of infrastructure, right? So, I've felt that it's slower [to develop with].*" Additionally, participants complained about a tendency toward dependency bloat (I = 4, S = 34%). I4 agrees: "*You know (cargo) goes and pulls every dependency ever.... That part's bad. It encourages dependency bloat, which, in a security focused area, is also the exact opposite of what you want.*"

4.5.4 Mostly positive impact on development

We find Rust offers development-cycle benefits that may in part offset its learning curve and upfront adoption costs.

4.5.4.1 Rust improves confidence in code

A key benefit mentioned by participants is that once Rust code compiles, developers can be fairly confident that the code is safe and correct. Four interview participants mentioned that they spend less time debugging in Rust than in other languages; this was supported in the survey, when 89% of respondents (87% of beginners) slightly or strongly agreed they spend less time debugging compiled Rust code than code in another language they are comfortable with. Interviewees also mentioned that Rust makes them more confident their production code is bug-free (I = 9); 90% of survey respondents slightly or strongly agreed. I16 said, "*The thing that I*

like the most about Rust overall is the fact that if the compiler is okay with your code then it will probably mostly be working."

4.5.4.2 Rust improves productivity in the development cycle

While the initial time to design and develop a solution in Rust is sometimes long and/or hard to estimate due to unforeseen conflicts with the borrow checker, interview participants felt — and survey participants agreed or strongly agreed — that Rust reduced development time overall, from the start of a project to shipping it, compared to other languages they were comfortable with (I = 7, S = 45%). I1 said, "*They see how well these projects go in comparison to the C++ projects;... and they've seen quantitatively that the Rust projects they've been working on have been a dramatically better experience and more predictable and a faster lifecycle.*"

Additionally, five interview participants noted that they could more quickly design and implement bug-free code in Rust than in another language they were comfortable with. This is echoed in the survey, where 61% of participants agreed or strongly agreed, as shown in Figure 4.3. 81% of survey participants also strongly or slightly disagreed that maintaining code is more difficult in Rust than in other languages. The reported improvement in developer productivity and code quality resulting from the use of Rust means that companies and organizations can ship better-quality code in less time.

4.5.4.3 Rust improves safe development in other languages

Most participants report Rust has had at least a minor positive effect on their development in another language they're comfortable with (I = 10, S = 88%). These participants said Rust

causes them to think about ownership (I = 5, S = 68% of 155), data structure organization (I = 6, S = 59%), use of immutability (I = 0, S = 48%), iteration patterns (I = 1, S = 45%), memory lifetimes (I = 4, S = 37%), and aliasing (I = 2, S = 25%). This is encouraging, as it shows developers carry over the safety paradigms that they are forced to consider in Rust when working in other languages. This is exemplified by S40, who said, "*Once you learn Rust, you are one with the borrow checker — it never leaves you. I now see many of the unsafe things I have been doing in other languages for years, (but probably not all of them, as I am human and not a compiler).*"

Notably, a few participants volunteered that Rust has even made them stop using C++ altogether (I = 2, S = 2). S26 said, "*It has made me stop working with C++. I really do feel that Rust replaces C++'s use cases well.*"

Overall, these results hint that the high cost of learning Rust can be worth it, providing longer-term benefits in other applications. This means developers may write more secure code in other languages, and organizations may get benefit out of investing time in their developers learning Rust.

4.6 Organizational adoption of Rust

While the Rust benefits identified by our participants may also apply to organizations, many participants mentioned experiencing pushback from teammates or managers (I = 9, S = 41%). Notably, some participants' attempts at adoption were unsuccessful (I = 1, S = 20%).

Participants identified several organization-level apprehensions about adopting Rust. We divide these into two categories: those that may apply to any change in programming language, and those that are specific to Rust. Rust-specific concerns closely mirror the drawbacks of

adopting Rust individually our participants identified above.

4.6.1 Apprehensions about any new language

We detail the organization-level apprehensions that participants' companies had that might be true when adopting any new tool or language.

4.6.1.1 Unfamiliarity with the language

Many participants cited unfamiliarity with Rust as one reason people were worried about adopting or did not adopt Rust at their company (I = 2, S = 69%). Any change to an unfamiliar language could create uncertainty or apprehension.

4.6.1.2 Avoiding unnecessary changes

Participants also reported a general desire to avoid unnecessary change. In particular, several participants' companies are reluctant to add any new languages (I = 2, S = 46%). As I2 explained, "*Not wanting to have too many languages in play at the company simultaneously, and so just a general conservatism there around not wanting to pick up new languages willy nilly.*"

4.6.1.3 Business pressures

Some participants said their companies were concerned about using or did not want to use Rust because there was time pressure to deliver a product, and they did not want to invest the time to get a new language and its infrastructure up and running (I = 1, S = 38%).

4.6.1.4 Lack of fit with existing codebase and ecosystem

Participants reported that lack of compatibility with the existing development ecosystem (I = 2, S = 27%) or interoperability with the existing codebase (I = 4, S = 27%) were concerns for their employers. As I6 said, "*It's very different for your developers or your managers who are managing a large, mature C++ ecosystem. They're much more skeptical of Rust. Maybe not on its merits, but just in the practical terms of how do I integrate this with my huge existing ecosystem?*"

4.6.2 Apprehensions specific to Rust

We detail the organization-level apprehensions that participants' companies had that were specific to the design and ecosystem of Rust.

4.6.2.1 Rust's steep learning curve

The difficulty of learning Rust was among the biggest concerns participants encountered at their companies (I = 3, S = 50%). I14 said one worry was "*how are we going to ramp programmers up? And I think Rust in particular has this reputation of having a very steep learning curve.*" Some participants' companies were also concerned about potential reduction in developer productivity (I = 3, S = 29%) or difficulty maintaining Rust code (I = 1, S = 23%). At I7's company, for example, "*the main concern was that it would be taking too long to use Rust.*"

4.6.2.2 Rust's maturity and maintenance

Since Rust is relatively new, some participants cited company concerns about the maturity and maintenance of its tooling and ecosystem, as well as whether it would be around long-term (I = 4, S = 29%). As I1 said, "*If [developers are] launching a new codebase in a new language, it's going to take them a year, maybe three years, to develop the things, and they care where the ecosystem will be at that point.*" Other comments reflected a lack of trust in the Rust toolbase (I = 3, S = 8%). I14's company worried, "*How well supported is the tool chain? How mature is the compiler? . . . Rust is a new language.*"

4.6.2.3 Difficulty hiring Rust developers

Stemming possibly from the newness and the difficulty of learning Rust, some participants reported their companies worried about the ability to hire Rust developers (I = 5, S = 42%). I11, for example, said, "*Do we really want to keep this thing in Rust? It's hard to find a new person for the team. . . because we don't have . . . a huge pool of Rust programmers.*"

4.6.3 Ways to encourage adoption

Despite these apparent apprehensions, many participants' companies still adopted Rust (I = 15, S = 49%). We report their suggestions for enabling adoption.

4.6.3.1 Pick projects carefully

Participants suggest that advocates pick initial projects for Rust carefully. Projects should fit Rust's strengths (I = 5, S = 2), both in terms of language design and available tooling. S62

recommended, "*Pick projects that are suitable for Rust, based on how mature the ecosystem (crates) is at supporting that type of project.*" S99 similarly commented, "*Don't try to port paradigms or design patterns from other languages.*" Participants also advise starting small (I = 5, S = 12). S95 said, "*Start small. There are many little problems that Rust programs solve well, which builds trust.*"

4.6.3.2 Demonstrate value

Participants argue that adoption hinges on demonstrating the value of using Rust. Most importantly, participants said advocates must argue that Rust offers a measurable improvement over the company's current language (I = 6, S = 10). While Rust touts its guarantees for safety and correctness, companies want to know the time and effort they allot to tackle the Rust learning curve will result in a major benefit. For example, S65 suggests, "*If you give a presentation about Rust, focus on concepts unique to Rust and what they offer; what matters is the idea that somehow it's possible to write safe, concurrent & fast software thanks to those concepts.*" This echoes results from Haney et al. suggesting security advocates must demonstrate value to motivate people to take appropriate security actions [163].

Participants emphasize being clear and straightforward about Rust's drawbacks, while arguing that the benefits outweigh them. I3 recommends "*rewrit[ing] [code] in Rust and swap[ping] it in.... And then you say look, this provides the same API. You didn't even know.*" If advocates can show their managers and teammates that using Rust had no negative effect on the codebase, they may be less apprehensive about its effect on productivity and timelines. Other participants recommend using a prototype to show that Rust is worth adopting (I = 4, S = 4). As I11 said,

"*We're gonna do a prototype. If doesn't work we'll just kill it*"

4.6.3.3 Account for upfront costs

Another strategy suggested by participants is to be clear about, and attempt to mediate, upfront costs, including additional time to design for the ownership paradigm as well as challenges related to tooling and dependencies (all discussed above). Participants suggest advocates spend time significant time planning tooling (I = 4, S = 2). I14 specifically advised to "*invest in your tooling upfront. Everybody starts out with Cargo, and Cargo is wonderful for what it does, but it has problems.*" Due to the steep learning curve, participants also suggest that advocates budget enough time to get started (I = 3, S = 1). For example, I5 advised, "*Factor in the learning curve and ramping up period that you're going to need to do. Because with initial adoption, you're probably not going to be able to hire like Rust programmers . . . for a decent size project, and . . . it does take a long time to kind of become productive in Rust, especially compared to some other languages, but if you are expecting that then over the long term you're gonna get big advantages.*"

4.6.3.4 Be helpful and have a good support system

Given the steep learning curve, participants emphasize the need for advocates to be willing and able to help new developers (I = 4, S = 2). They recommend the advocate themselves be a knowledgeable Rust developer (I = 2, S = 8): S118 suggests "*Make yourself an expert (e.g., via personal projects and study) and share your expertise generously. People will feel more comfortable with an unfamiliar language if they have a friendly, helpful expert on their team. Finally, be patient. . . . Being friendly, helpful, and humble usually works better than being*

pushy, righteous, and evangelical." Similarly, S73 said, "*Make sure you are willing to mentor aggressively for a long time.*" Further, some participants suggest a formal support system for teaching and mentoring new Rust developers (I = 3, S = 3). I8 advised, "*Try to just have some good support for newer engineers.... If you do happen to have a couple engineers who are more proficient in Rust and are willing to help, ... have those engineers help the newer ones.*"

4.6.3.5 Be persistent but patient

Companies may not always buy in to adopting Rust immediately. Some participants suggest advocates for Rust be persistent (I = 1, S = 3). S84 suggested adopters should "*keep at it and try to get coworkers to pick it up as well. Strength in numbers.*" However, participants also suggested advocates be patient (I = 3, S = 5) and not "*expect [their employer] to agree to making any changes at first.*" Advocates need to "*give Rust time and be patient, the memory model and lack of OOP combine to make it difficult for existing programmers to jump into.*" This advice — which ties into the steep learning curve and lack of language maturity discussed in Section 4.6.2 above — aligns well with Haney et al.'s finding that building relationships and trust helps with the adoption of secure systems and technologies [163].

4.7 Discussion and recommendations

Our results demonstrate that there are drawbacks to adopting Rust but, at least for our participants (many of whom are Rust enthusiasts), the benefits appear to outweigh them. This section summarizes what we can learn from Rust's success to date, and recommends steps toward improving adoption or use of Rust itself, as well as other secure languages and tools.

4.7.0.1 Making secure tools and languages appealing

All but one survey respondent said they would either probably or definitely use Rust again in the future (S = 99%) and many survey participants felt that their employer would likely use Rust again (S = 88%). This mirrors the results of the Stack Overflow Developer Survey, where Rust has been the "most loved" language for the last five years in a row [148].

Our results shed some light on why this might be. We confirmed that to a large extent, Rust is perceived to meet its motivating goals of security and performance. Further, Rust's tools provide high-quality feedback (e.g., error messages), the language boasts good documentation, and it has an active and helpful online community; all of these were deemed important in prior studies of language adoption [164]. Good documentation and a responsible and attentive community are also known to be important for encouraging adoption of secure APIs and programming patterns [28, 165].

4.7.0.2 Flatten the learning curve

Participants overwhelmingly report that learning Rust had a positive effect on their development skills in other languages, including by internalizing memory safety-relevant concepts such as ownership and lifetimes. Rust caused participants to shift their programming mental models, which echoes prior work showing that "mindshifts are required when switching paradigms" [166].

Unfortunately, our participants also report that Rust can be very difficult to learn (Section 4.5.2) precisely because of the difficulty of adhering to these concepts (as enforced by Rust's ownership and lifetime rules). As observed with other security tools, Rust's learning curve may be turning some developers and/or organizations away from using it [167].

Finding ways to flatten this curve could have a big impact. For example, it may make sense to develop a version of Rust that allows users to incrementally learn the difficult concepts of ownership and borrow checking, rather than forcing them on users all at once. We speculate that Go may be easier to learn for developers given its garbage collected memory model, which removes some of the burden of memory management from the developers. Could we create a version of Rust with garbage collection as a learning tool?

4.7.0.3 Reduce the risk of investment

Several of the drawbacks we identified interact in ways that may multiply the perception of risk related to adoption. Much of the cost of adoption occurs up front: the steep learning curve, the relative immaturity of the ecosystem, the slower initial development time, and the inherent challenge of making a large change. Benefits accrue later: improvements in security-minded programming, shorter debugging time and eventually shorter development time overall, and enforced avoidance of key security problems, since Rust is type- and memory-safe. The perceived difficulty of hiring experienced Rust developers, as well as concerns about longevity and future maintenance, may make these future-term benefits seem too uncertain to be worth the risk.

Educators and security advocates who want to incentivize secure programming languages should look for ways to improve this calculus, perhaps by investing in a pipeline of trained Rust developers (reducing learning curve and improving hiring prospects), by developing libraries to contribute to the increasing stability of the ecosystem, or perhaps by developing models and templates for common porting and interoperability challenges. Our participants offer suggestions

for action within organizations, such as "starting small" to demonstrate value, and implementing mentoring support for transitioning to Rust. Security advocates could help, by creating and publishing detailed case studies that illuminate benefits and costs of adopting secure tools in real systems, and by creating and supporting mentoring networks for these tools.

4.7.0.4 Improve the culture around unsafe code

Rust's memory safety-related security benefits come simply by virtue of using the language, but only as long as `unsafe` blocks are used correctly; the more often and more carelessly they are used, the greater the risk of a security hole. Our participants report that `unsafe` blocks in Rust code are being used frequently, often with only rudimentary vetting processes; prior studies have come to similar conclusions [162, 168]. While many participants/companies do recognize the risks of unsafe code, we encourage the adoption of more, and more formal, review procedures to more thoroughly mitigate these risks.

4.7.0.5 Reaching non-enthusiasts

While our participants had a variety of general- and Rust-specific development experience, our sample overrepresents people who show active interest in Rust (as indicated by their adoption efforts and/or membership in a community forum). Future work could explore recruitment strategies to target developers who failed in their adoption efforts and/or lost interest in Rust programming.

Chapter 5: Exploring Methods for Secure Development Studies

Our results highlight that studying secure development from a human-centered perspective remains an important research focus. While the usable security community has been studying how to improve tools for and the mental models of end-users for about 25 years, only in the last 8 years has the community turned significant efforts to studying these phenomenon for developers and security professionals. This means that there are many community best practices for conducting studies with end-users, but less so for studies with developers. Studies with developers often involve interviews [45,53–55,73,169], surveys [36,46,69,72,169] and programming tasks [20, 28, 85, 170]. In 2016, Acar et al. called for studies to further explore methodology for developers [25].

Conducting secure development studies is often difficult due to the time consuming nature of code-writing studies and recruitment of a specialized populations [25, 26, 119]. Recruiting participants for developer studies is often challenging, as they participate in the study outside of their normal work hours [27]. Additionally, the pay to participate in studies is often substantially lower than what developers make in their daily jobs, lowering the incentive to participate. This can lead to high drop-out rates in studies resulting in smaller and less significant sample sizes [28]. However, if we can reduce the time spent by developers while maintaining the same quality and validity of the results, we may be able to yield larger and better sample sizes for these studies.

As a first step to address this problem, Danilova et al. explored the possibility of using code review as a methodology in place of writing code by having participants write code reviews about code snippets from a prior study about secure password storage [96]. They are able to draw some comparisons between their results and the results of the prior work [27], but they were not able to compare directly across experimental conditions. We aim to expand on their work by exploring the use of reading insecure code and fixing insecure code as methodological substitutes for writing code. We compare participant time spent, participant frustration levels, and results both across conditions and with prior work.

To this end, we conducted a remote experimental study with Python programmers, who were asked to complete two secure-development tasks. The study was adapted in part from a prior study comparing cryptographic libraries [28]. Each participant was assigned to one cryptographic library and one method: writing secure code from scratch ("write"); reading existing code and then finding and explaining any vulnerabilities or functionality issues ("read"); or reading existing code and then finding and fixing any vulnerabilities or functionality issues ("fix"). Participants were then directed to share their experiences while participating in the study.

In particular, we were interested in understanding whether the read and fix conditions provide comparable results about functionality, security, and usability as the write condition; what types of tasks (functionality and/or security) participants in the fix and read condition could complete successfully, and whether participants in the read and fix conditions experienced fewer negative effects (drop-out rate, frustration, time to complete) than those in the write condition.

5.1 Method

To understand how reading and fixing code compare as methods to writing code, we conducted a remote lab-study with Python programmers in which they completed two secure development tasks.

5.1.1 Recruitment

We recruited from Upwork and through computer science student mailing lists at multiple universities from May 2022 - February 2023, following best practices [87, 88].

Upwork allows researchers to filter participants based on their skillset. We filtered participants for experience writing at least one small project in Python and age (18 or older). For recruitment from computer science student mailing lists, we created a short screening survey to ensure that participants in the main study would have programming and Python experience. We started with a few questions to understand their general programming experience and occupation and concluded by using the questionnaire created by Danilova et al. [91] to determine if a participant actually had programming experience. A full copy of the screening survey can be found in Appendix A.4.1.

For the full study, participants from Upwork were invited if, from their profile, they had completed at least one small project in Python. For participants recruited via CS mailing lists, we exclusively invited participants who had Python experience and were able to correctly answer all of the questions from the Danilova et al. questionnaire on a first come, first served basis. Participants were not compensated for the screening survey, but all main study participants were compensated $35 for completing the study with a possibility for a $5 bonus if their solution met a high correctness threshold. This bonus was meant to encourage people to give their best effort

Task	Type	Vuln/bug	Description
Encrypt/decrypt	Security	Fixed IV	IV was set to static value
	Functionality	Return plaintext data	Return plaintext instead of ciphertext
	Functionality	Encrypt key instead of plaintext	Send key to encryption function instead of plaintext
	Functionality	Use plaintext as encryption key	Use plaintext as key to create cipher object
	Functionality	Return ciphertext data	Return ciphertext instead of plaintext
Key generation and storage	Security	Fixed salt	Salt was set to static value
	Security	Weak KDF	PBKDF1 used
	Security	Weak hash algorithm	Sha1 used
	Security	Bad mode selection	Insecure mode used to encrypt key
	Functionality	Wrong name to open	Send wrong variable to open command when writing key to file
	Functionality	Incorrect length for password check	Use wrong value to check length of password before using in keygen

Table 5.1: Vulnerabilities and bugs that were included in read and fix code snippets when participating in the study.

5.1.2 Study Design

To ensure the validity of our study conditions, we decided to partially replicate a prior study. This allowed us to compare the results of our write condition with the original study's results, while also allowing us to compare the results of our read, write, and fix conditions.

We chose to partially replicate the 2016 study from Acar et al. in which they explored the usability of various Python cryptography APIs [28]. We chose this paper because it offered self-contained and short code writing tasks with results that lent themselves well to being compared across different conditions. Additionally, we had familiarity with Python, which would make the process of identifying vulnerabilities and writing solutions easier.

The original study [28] used five libraries and two sets of tasks (symmetric and asymmetric encryption), for a total of 10 conditions. Comparing three different methods would triple this to 30 conditions, requiring a sample size that would not be feasible when recruiting developers. As such, we opted to only replicate a subset of the original conditions (two libraries, one task), crossed with our three methods, for a total of six conditions.

We had participants only work on symmetric encryption tasks rather than both symmetric and asymmetric tasks. We chose symmetric over asymmetric tasks as it served as the baseline for all of the regressions performed in the original paper, and participants in the symmetric condition produced more functional solutions. Participants were assigned to work in either PyCrypto or Cryptography.io. We selected PyCrypto as it was the baseline library for all the models in the original paper. We selected Cryptography.io as it proved to be significantly better in regards to security results in the original paper and was designed with usability in mind. Further, we used the versions of each library used in the original study rather than their current versions because many insecure defaults and security issues identified in the original study have been addressed in modern versions of the libraries. Finally, participants were assigned to one of three experimental conditions: writing code from scratch ("write"); reading existing code, determining its correctness, and describing anything they wanted to change ("read"); or reading existing code, determining its correctness, and fixing any vulnerabilities or bugs in the code ("fix").

5.1.2.1 Task selection

Participants were randomly assigned to one of the six conditions and tasked with completing an encrypt/decrypt task and a key generation and storage task. The order they were presented in

was randomized.

For the write task, participants were given stub code and asked to write code that completed the task described. For the encrypt/decrypt task, this meant writing code that encrypted or decrypted plaintext or ciphertext, respectively, using a provided key. For the key generation and storage task, participants were tasked with creating an encryption key using a provided password, storing it encrypted to a file in a provided directory name, and recovering the key from the same file. They were provided with tests for each function and a set of tests at the very end to test all of their code. The tests covered the basic functionality of each task and ensured that the code ran without failure and returned the correct value. The key generation and storage function stub, encrypt/decrypt function stub, and provided tests for both tasks can be seen in Listing A.1, Listing A.2, and Listing A.5 in Appendix A.3.

For the read task, participants were given already completed code and asked to read the existing code, determine its correctness, and add comments to identify what they would change and how. For the fix task, participants were given already completed code and asked to read the existing code, determine its correctness, and fix it if necessary. In both read and fix, the code provided contained four unique functionality bugs and one unique security vulnerability for the encrypt/decrypt task and two unique functionality bugs and four unique security vulnerabilities for the key generation and storage task. The functionality bugs ranged in type from those that would cause the code to crash when run to those that would not cause the provided tests to fail. The security vulnerabilities were based on vulnerabilities identified in the original paper [28], to best allow for comparing results between studies. We included a comprehensive subset of the types of vulnerabilities they found in the original study, picking a variety that covered key generation and storage and encryption and decryption. A full list of the functionality bugs and

security vulnerabilities present in the provided code for participants in the read and fix conditions can be found in Table 5.1. Participants in the read condition and the fix condition were provided with the same tests as participants in the write condition. The key generation and storage code, encrypt/decrypt code, and provided tests for participants can be seen in Listing A.3, Listing A.4, and Listing A.5 in Appendix A.3.

5.1.2.2 Study environment

Our participants completed the study remotely through a heavily modified version of the Developer Observatory system [171]. The developer observatory allows participants to complete coding tasks from the comfort of their home. This means that participants do not need to come to a lab to complete a "lab study," allowing for easier recruitment.

The original Developer Observatory was first adapted to run on a collection of Docker containers, as opposed to Amazon Web Services virtual machines, to accommodate more concurrent participants with shorter startup times. The internal representation of programming tasks was expanded to properly represent our varied conditions and requirements for each (e.g., not being able to run code in the Read condition). Various other features were added such as the ability to leave and return to an ongoing study, automatic tracking of consent forms, and system statistics collection. The study infrastructure code can be found on our Github page.

Once participants consented, they were taken to the instructions for the study. From there, they were able to click start and begin participating. Participants in the fix and write conditions were able to run their code as often as they liked. Participants in the read condition were not provided with a run button for their code, but they were still able to use hot keys to run code in

Jupyter notebooks. This became an issue within the study, so the interface was edited to remove the ability to run the code using hot keys as well. Participants were able to switch between tasks (forward or backward) at any point. Once participants completed all the tasks, they clicked a *finish* button which took them to the final survey. An example of the Developer Observatory can be seen in Figure 5.1.

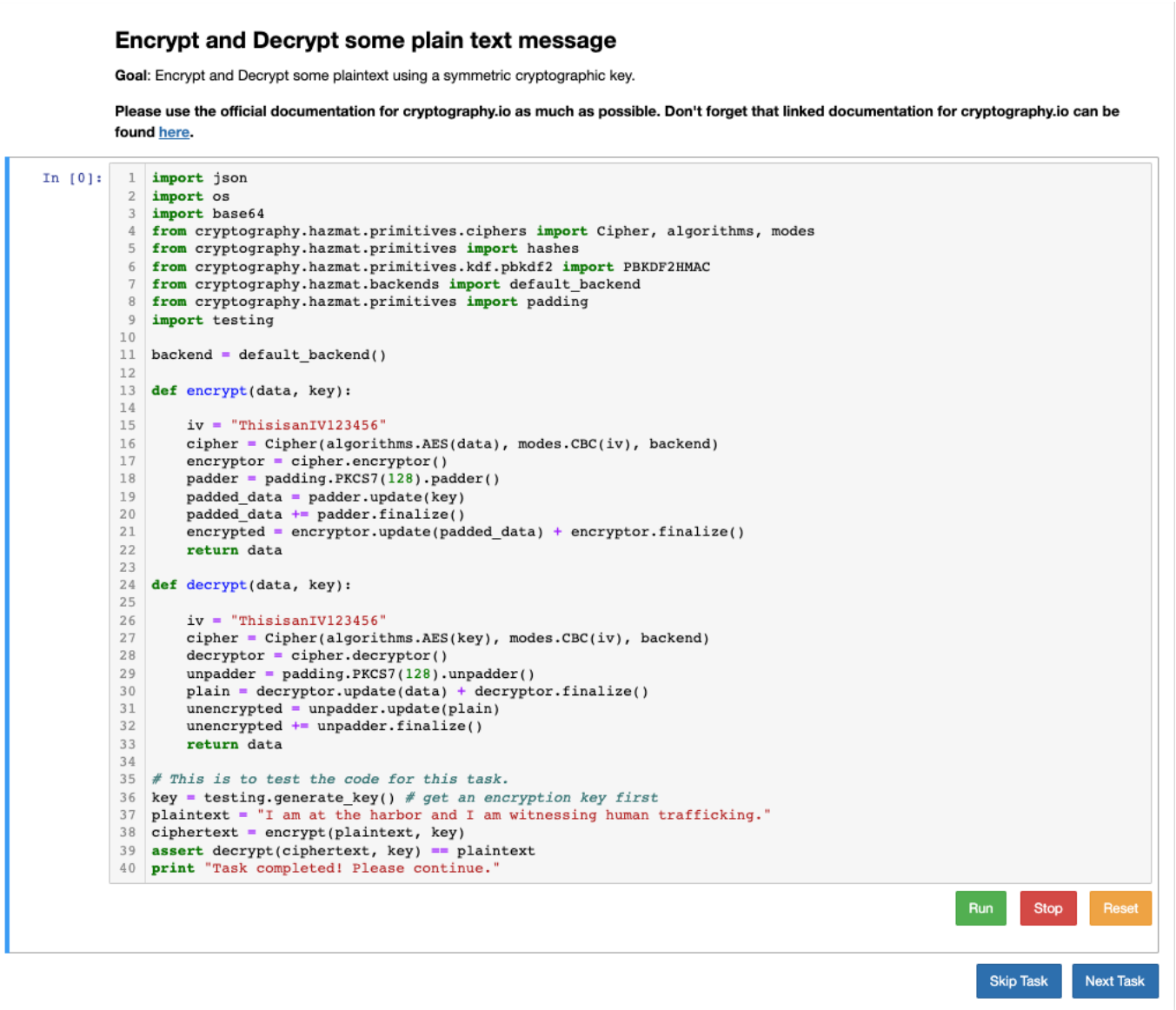

Figure 5.1: Example of a task in the Developer Observatory

5.1.2.3 Exit survey

Once participants indicated that they were finished with the tasks, they were directed to the final survey. The final survey first asked about the perceived security and correctness of their solutions, how frustrating, fun, tedious, and challenging they found the tasks, and what was easy and hard about the tasks. The next section of the survey contained the System Usability Scale (SUS) [172] and the usability scale developed and used in the original paper [28]. The next section contained the Secure Software Development Self-efficacy Scale, which measures a participants' perceived ability around various security tasks [93]. Finally, the survey concluded by asking participants about their security experience, general development experience, Python development experience, and general demographics. A full copy of the survey can be found in Appendix A.4.2.

5.1.3 Data Analysis

To determine what vulnerabilities and functionality bugs were introduced by participants in the write condition and not identified or fixed by participants in the fix condition and the read condition, we manually reviewed the submissions following the same process as in our prior work [20,173]. Two coders reviewed each task for functionality bugs and security vulnerabilities. For the write condition, the coders reviewed both tasks to identify any bugs or vulnerabilities present, using the vulnerabilities present in the prior work as a reference point. Each bug or vulnerability was labeled for type (functionality or security) and the specific vulnerability. In addition, we categorized the vulnerabilities into "issue" classes based on the classifications used in our prior work [20].

For the read and fix conditions, the two coders reviewed submissions using a list of the bugs and vulnerabilities they knew to be present. For the fix condition, vulnerabilities or bugs included in the study setup were labeled for whether participants were able to correctly identify and fix it, what the participant said they changed in the code to address the vulnerability/bug, how the participant said they changed the code to address the vulnerability/bug, and why the participant said they changed the code to address the vulnerabilities/bug. For the read condition, vulnerabilities or bugs included in the study setup were labeled for whether participants were able to correctly identify the existence of the vulnerability/bug and whether the changes they proposed would fix the vulnerability/bug, what the participant said they would change to address the vulnerability/bug, how the participant said they would change the code to address the vulnerability/bug, and why the participant said they would change the code to address the vulnerabilities/bug. We also labeled any additional issues identified by participants that were not actual bugs and vulnerabilities (false positives).

IRR was calculated for all variables using Krippendorff's α statistic, a conservative measure that considers coders' agreement as an improvement over randomly guessing. We met the recommended threshold for Krippendorff's α of 0.8 [112]. All vulnerabilities and bugs were confirmed by both coders, and consensus for all codes was reached via discussion. The final codebook and associated IRR values are given in Table 5.2.

5.1.4 Ethics

Both surveys (pre-screen and final) and the full study were approved by the University of Maryland's Institutional Review Board (IRB). We obtained informed consent once before the pre-

Data labeled	Description	Values	Agreement
Type	Type of bug/vulnerability	Security	0.982
		Functionality	
Function	Which function the issue occurred in	Encrypt	0.988
		Decrypt	
		Key generation	
		Read key from file	
Vulnerability	Vulnerability/bug that occurred	See Table 5.3	0.991
Issue	Issue that caused vulnerability/bug	See Table 5.3	0.989
Identified	Was bug/vulnerability identified by participant	Yes/No/NA	0.984
Actual issue	Was bug/vulnerability an actual bug/vulnerability	Yes/No/NA	0.983
Fixed	Did the participant fix the bug/vulnerability	Yes/No/NA	0.981
What	What the participant changed	Type of KDF	0.967
		Change encryption mode	
		Value of the salt	
		Number of iterations	
		Initialization vector	
		Hash function used	
		Name passed to open	
		Variable returned	
		Length of password check	
		Variable passed to crypto init	
		Variable being padded	
		Variable being encrypted	
How	How the participant changed the code	Change to secure KDF	0.933
		Change to more secure mode	
		Randomly generate salt	
		Change iterations to 10,000+	
		Randomly generate IV	
		Change to more secure hash function	
		Change iterations to 1000	
		Change file to filename	
		Change it to 16	
		Change variable name to data instead of key	
		Change variable name to key instead of data	
		Return unencrypted instead of data	
		Return encrypted instead of data	
Why	Why the participant changed the code	Documentation recommended other KDF	0.815
		Current mode is insecure	
		Salts need to be random	
		PBKDF2 prevents against guessing	
		Increasing iterations reduces guessability	
		IVs need to be random	
		Example in documentation	
		Function depreciated	
		SHA1 is insecure	
		Example from webpage	
		Increasing iterations improves security	
		Example in documentation (misunderstood)	
		Example in documentation (insecure)	
		Need to pass the right name	
		Reserved Python keyword	
		Runtime error	
		Need to return correct variable	
		Need consistent length checks for password	

Table 5.2: Descriptions and agreement of codes for participant submissions

Type	Issue	Vulnerability
Security	Weak cryptography choice	Insecure mode selection
		Weak KDF
		Custom KDF
		No KDF
		Insufficient iterations in KDF
		Weak hash algorithm
		Insecure algorithm selection
	Fixed/static value	Static salt
		Static IV
		Empty/missing IV
		Use password as encryption key
		Use password as salt
		Static key
		Use file key as key
	No encryption	Store key unencrypted
	Homemade crypto	XOR only for encryption
Functionality	Incorrect variable name used	Variable name for file sent to open
		Key being padded
		Data sent to encryption init function instead of key
		Key being encrypted instead of data
		Store file key as encryption key
		Write AES.new object not key
	Inconsistent bounds checking for function	Check for length of password
	Incorrect value returned	Return (un)encrypted data
		Return file key
		Don't return (un)encrypted data
		Decrypt data in encrypt function
	Insufficient error checking	Called non-existent module
		Use variable that does not exist
		Incorrect number of variables passed to function
	Used asymmetric encryption	Used RSA to generate key
	Don't store encryption information correctly	Don't store key to file
		Don't use same IV to decrypt
		Don't store key in keydir
		Don't use same salt across gen/read
	Incorrect code formatting	Incorrect indentation

Table 5.3: Vulnerabilities/bugs and issues that caused them

screen survey and again before the full study (tasks and final survey). Participants were informed that they could skip any task or question and drop out of the study at any time.

5.1.5 Limitations

A key limitation of our work is age of the study that we are replicating. The cryptographic libraries themselves, as well as their documentation, have changed drastically since the study originally ran. This means that any search for assistance on Google or StackOverflow will likely result in information that does not match the version of the library used in our study. We were able to provide participants with a version of the documentation that matched the versions of libraries they were using. Since we are not actually concerned with evaluating the current usability or security of the libraries, but rather with understanding the effect of the method used to study them, this limitation does not reduce the validity of our results.

The original study aimed to understand the usability of cryptography APIs for professional developers. We aim to semi-replicate this study. However, about half of our participants for this study were students, differing from the original study which used GitHub developers. While students often have less experience than professionals, several recent studies conclude they can be adequate substitutes in secure software development studies [27, 85, 87, 88], as many skilled professionals have limited experience with security specifically [20, 23, 49, 114–119].

Our goal in this study was to understand the feasibility of using reading and/or fixing code as experimental substitutes for writing code in secure software development studies. This study serves as a single data point in this exploration. We explore whether reading and/or fixing code works to compare the usability of cryptography APIs. Additionally, the tasks in this study were

deliberately small and self-contained to allow for easy comparison amongst the experimental conditions. Thus, our results may not generalize to all kinds of secure software development studies, such as those exploring other secure development issues or tasking participants with building larger projects. However, we believe this study is a good first step into exploring this phenomenon.

A final possible limitation of this work is the reliance on some self-report data to measure negative experiences of participants, such as reported frustration or challenge. While self-report data can be biased or inaccurate, this is the best proxy we have for measuring frustration levels. We do attempt to mitigate some of this self-report bias by also collecting other measurements of negative effects, such as time spent or dropout rates.

5.2 Participants

In total, 99 participants started our study, with 30 assigned to the fix condition, 41 to the read condition, and 28 to the write condition. In total, 91 participants completed our study with 30 assigned to the fix condition, 37 to the read condition, and 24 to the write condition. However, we removed 14 participants for a variety of reasons such as running or editing code in the read condition prior to removing the use of hot keys (N = 9), skipping every task in the study (N = 4), and not understanding what to do (N = 1). After exclusion, we ended with 77 participants who completed the study; 27 in the fix condition, 26 in the read condition, and 24 in the write condition. Details of participants assigned to each experimental condition and library can be found in Table 5.4.

	Write			Read			Fix		
	Started	**Completed**	**Valid**	**Started**	**Completed**	**Valid**	**Started**	**Completed**	**Valid**
Crypto.io	14	11	11	21	18	12	16	16	13
PyCrypto	14	13	13	20	19	14	14	14	14
Total	***28***	***24***	***24***	***41***	***37***	***26***	***30***	***30***	***27***

Table 5.4: # of participants that started the study, completed the study, and were excluded in each condition

5.2.1 Demographics

In general, our participants trended heavily toward male (68%), young (with ages ranging from 18 - 43 and 84% of participants being younger than 40), and educated (55% had at least a bachelor's degree). Our participants came from a variety of ethnic backgrounds, with a majority identifying as Asian (46%).

Our participants had 6.3 years of programming experience on average ($\eta = 5$ years) and 3.8 years of Python experience ($\eta = 3$ years). About 62% of our participants were employed in a professional role that required programming, with the most common job roles being developer and engineer. Of that 62%, 63% used Python in their job. Our participants had 3.4 years of professional programming experience ($\eta = 2.8$ years), and 1.8 years of professional Python experience ($\eta = 1$ year).

Finally, our participants had fairly little security experience, with an average of 1 year of security experience ($\eta = 0.5$ year). However, our participants were self-confident in their security abilities, with 62% rating their security knowledge at average or above average.

We had 46 participants from Upwork and 36 participants from CS mailing lists. Overall, we don't see much difference in the experience between the two groups of participants. Participants from Upwork and CS mailing lists had an average of 6.7 and 5.8 years ($\eta_{upwork} = 6$ years,

$\eta_{cs} = 5$ years) of programming experience respectively. Upwork participants had 4.2 years and CS mailing list participants had 3.2 years of Python experience on average ($\eta_{upwork} = 4$ years, $\eta_{cs} = 3$ years). About 72% of participants from Upwork and 50% of participants from CS mailing lists were employed in a job that required programming with 73% and 44% of those participants using Python at their job respectively. Participants from Upwork had 4 years of professional programming experience including 2.2 years of professional Python experience on average ($\eta_{upwork} = 3$ years, $\eta_{cs} = 2$ years). Participants from CS mailing lists had 2.4 years of professional programming experience and 0.8 years of professional Python experience on average ($\eta_{upwork} = 2$ years, $\eta_{cs} = 1$ years). Finally, comparing security experience, participants from Upwork had 1 year and participants from CS mailing lists had 0.8 years of experience on average ($\eta_{upwork} = 0.2$ years, $\eta_{cs} = 0.8$ years). Upwork participants and students were about equally confident in their security abilities with 63% and 61%, respectively, rating their security knowledge at average or above average.

Factor	Task	**[28]**	**Write**	**Read**	**Fix**
Dropouts		$\equiv$	$\equiv$	$\equiv$	$\equiv$
Functionality		$\equiv$	$\uparrow$	$\equiv$	$\equiv$
	Encrypt/decrypt	$\downarrow$	$\uparrow$	$\uparrow$	$\uparrow$
	Key generation and storage	$\equiv$	$\uparrow$	$\equiv$	$\equiv$
Security		$\downarrow$	$\equiv$	$\equiv$	$\equiv$
	Encrypt/decrypt	$\downarrow$	$\equiv$	$\equiv$	$\equiv$
	Key generation and storage	$\downarrow$	$\equiv$	$\equiv$	$\equiv$
SUS		$\equiv$	$\equiv$	$\equiv$	$\equiv$
[28] scale		$\downarrow$	$\equiv$	$\equiv$	$\equiv$

Table 5.5: Trends of results across all conditions where $\uparrow$ indicates more or significance for PyCrypto, $\downarrow$ indicates more or significance in Cryptography.io, and $\equiv$ indicates about equal between PyCrypto and Cryptography.io.

5.3 Replicating results from Acar et al. [28]

To explore the validity of the write condition in our study, we first compare the results of our write condition to the original results from Acar et al. [28]. Rather than running the regressions from the original paper, we ran Fisher's Exact Tests (FET) and t-tests due to our much smaller sample size. We compare our results to the original study results in regard to dropouts, functionality, security, and usability. While the regressions run in the original paper did not see significance for dropouts, functionality, and usability score, we still report and compare these results for completeness and to give context. Trends for each of the tests for both the original study [28] and the write condition can be found in Table 5.5.

5.3.1 Dropouts

We first explore how the dropout rate in the write condition of our study compared to that of [28]. In the original study, 95 (70%) and 97 (71%) participants dropped out from the symmetric PyCrypto and Cryptography.io conditions. In our study, only 4 people dropped out in the write condition, 1 from PyCrypto (7%) and 3 from Cryptography.io (21%). We likely experience fewer dropouts overall due to the fact that we compensated participants for their time, unlike the original study. Details of the number of valid and completed participants can be seen in Table 5.4.

To explore whether the library had any effect on whether a participants dropped out, we ran Fisher's exact test. The test indicates no significant difference between libraries ($p = 0.596$, $OR = 0.294$, $CI = [0.005, 4.29]$). Similar to [28], while substantially smaller, we see a similar number of dropouts between the Cryptography.io and PyCrypto.

5.3.2 Functionality

We next explore the extent to which our participants in the write condition were able to produce functional solutions compares to that of [28]. Solutions were considered functional if they ran, passed the provided tests, and completed the assigned task. If the participants skipped the task, the result was considered as not functional. In Acar et al., participants were able to generate functional solutions for 85% of tasks when using PyCrypto and 90% of tasks when using Cryptography.io. In our study, participants using PyCrypto were able to produce 22 functional solutions (85%), where as participants using Cryptography.io were able to produce 11 functional solutions (50%). The number of functional solutions for each library can be found in Figure 5.2. The percentage of functional tasks for participants using PyCrypto mirrors the results of the original study. To compare our results to the original study, we ran a FET to determine how the assigned library affected the ability of participants to produce functional solutions. We find that participants in PyCrypto were 5.29 times more likely to produce a functional solution ($p = 0.014$, $OR = 5.29$, $CI = [1.22, 28.34]$). This result does not mirror the results of [28] as they found no significant difference between PyCrypto and Cryptography.io.

Seventeen participants were able to produce functional solutions for the encrypt/decrypt task, 11 out of 13 participants using the PyCrypto library and 6 out of 11 participants using the Cryptography.io library. The number of functional solutions the encrypt/decrypt task for each library can be found in Figure 5.3. Sixteen participants were able to produce functional solutions for the key generation and storage task, 11 out of 13 participants using the PyCrypto library and 5 out of 11 participants using the Cryptography.io library. Although only a slight difference, this result mirrors the results from the original study, in which participants were able to produce more

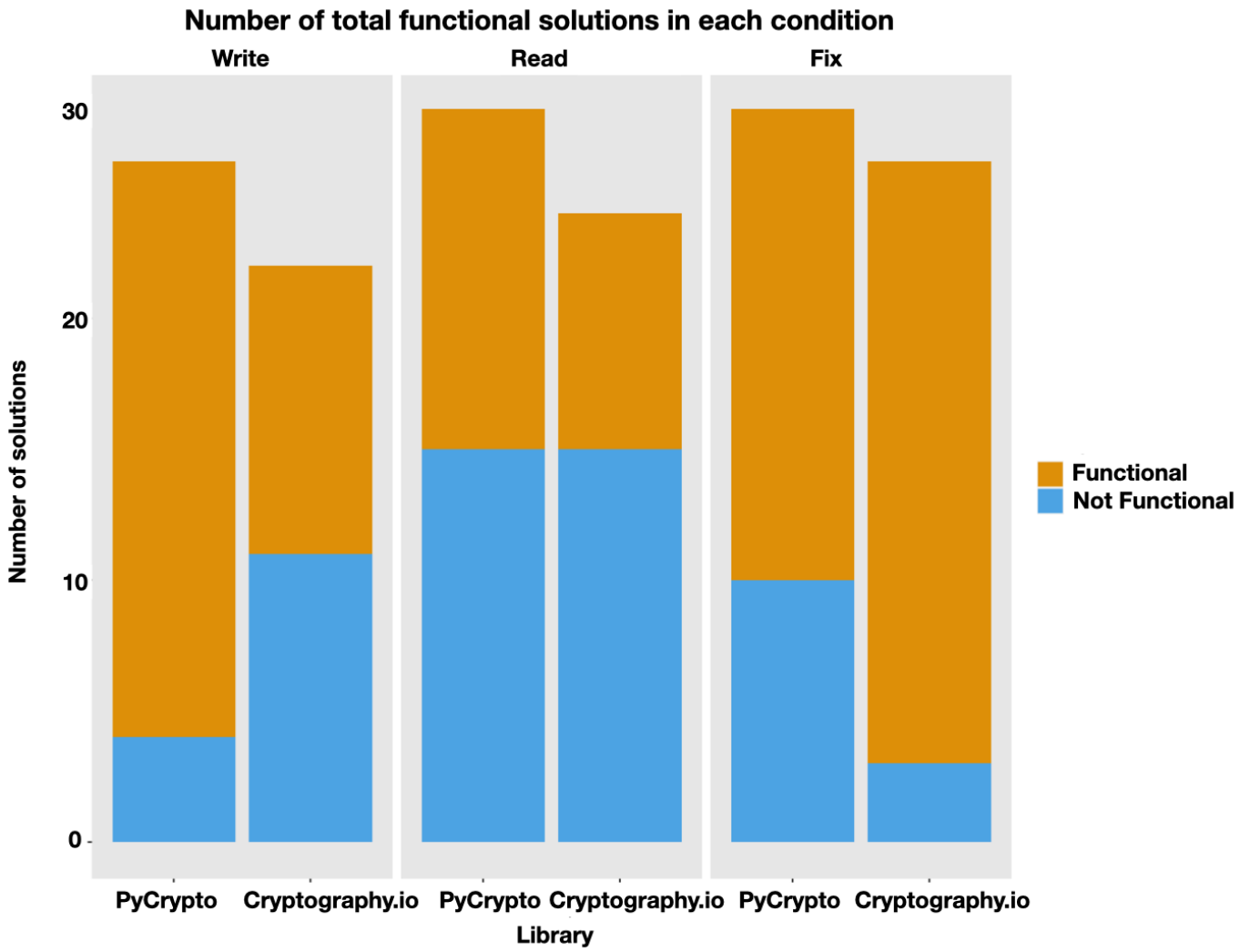

Figure 5.2: Total number of functional solutions in each condition

functional solutions for the encryption and decryption task than the key generation and storage task [28]. The number of functional solutions the key generation and storage task for each library can be found in Figure 5.4.

5.3.3 Security

Next, we explore the extent to which participants who produced a functional solution were able to produce a secure solution and how this compares to [28]. Additionally, we discuss the

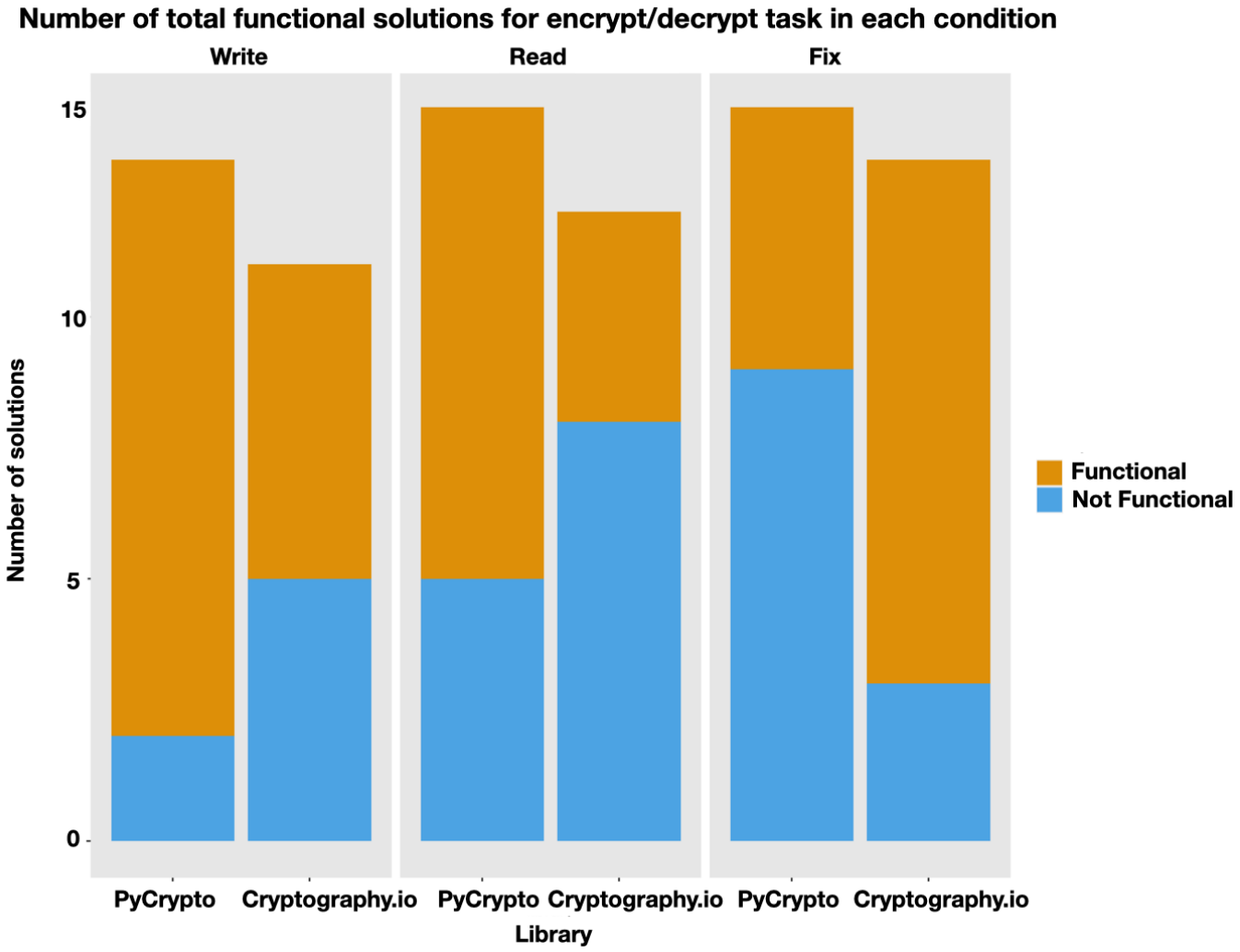

Figure 5.3: Number of functional solutions for encrypt/decrypt task in each condition

types of vulnerabilities introduced in our study and the original study. We use V_w to represent the number of vulnerabilities from our study for participants in the write condition and V_o to represent the number of vulnerabilities from the original study.

In Acar et al., participants were able to generate secure solutions for 15% of tasks when using PyCrypto and 70% of tasks when using Cryptography.io. In our study, participants using PyCrypto were able to produce 5 secure solutions (23%), where as participants using Cryptography.io were able to produce 5 functional solutions (45%). The number of secure solutions for each

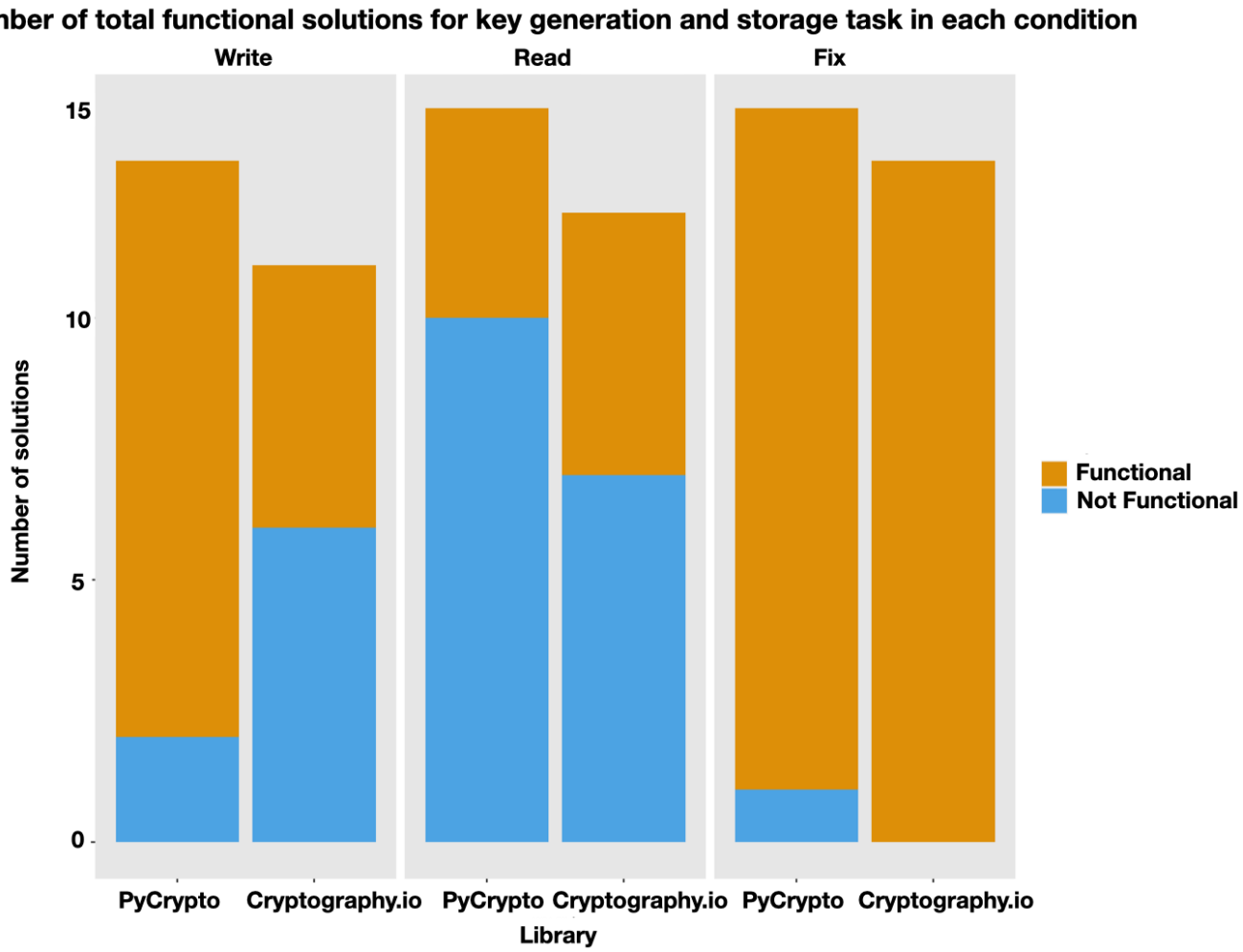

Figure 5.4: Number of functional solutions for key generation and storage task in each condition

library can be found in Figure 5.5.

To explore the effect of the library on security, we ran a FET and found no significant difference between the security of functional solutions for the two libraries ($p = 0.24$, $OR = 2.74$, $CI = [0.45, 17.30]$). This result does not mirror the original paper where they find significance in this relationship, likely owing to the much smaller disparity between the number of secure solutions between participants using the two different libraries. While the result is not significant, we do see a difference between the security of solutions of participant using the two different libraries which mirrors the results from the original study in which participants using Cryptography.io

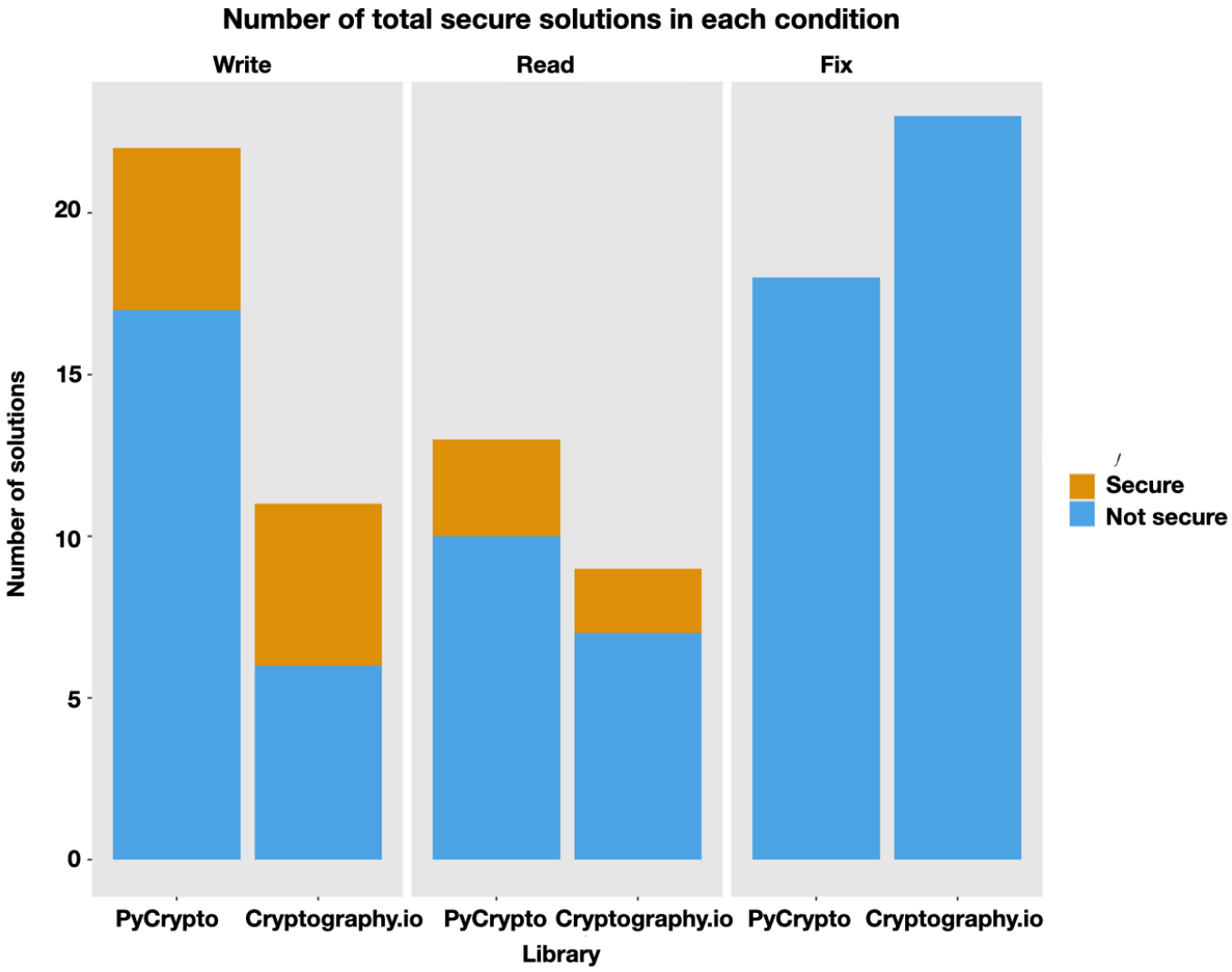

Figure 5.5: Total number of secure solutions in each condition

were able to produce more secure solutions than those using PyCrypto [28].

Of the 17 participants who were able to produce functional solutions for the encrypt/decrypt task, only 9 were able to produce secure solutions, 4 out of 13 participants using PyCrypto and 5 out of 11 participants using Cryptography.io. Of the 16 participants who were able to produce a functional solution for the key generation and storage task, only 1 was able to produce a fully secure solution, using PyCrypto. This again mirrors the results from the original study in which participants were able to produce more secure solutions for the encryption and decryption

task than the key generation and storage task [28]. The number of secure solutions for the encrypt/decrypt and key generation and storage tasks for each library can be found in Figure 5.6 and Figure 5.7 respectively.

The types of vulnerabilities that participants introduced also mirrored the results from the prior study. For the encrypt/decrypt task, the most common vulnerability was using a static initialization vector (V_w = 5, V_o = 29). This was followed closely by using a weak encryption mode (V_w = 3, V_o = 23) and using a weak encryption algorithm (V_w = 4, V_o = 17). For the key generation and storage task, the most common vulnerabilities was failing to use a key derivation function (V_w = 8, V_o = 15). This was followed closely by storing the key unencrypted (V_w = 6, V_o = 4), using a custom key derivation function (V_w = 4, V_o = 11), and using a weak mode to encrypt the key (V_w = 3, V_o = 14). While the distribution of vulnerabilities does not match perfectly, the types of vulnerabilities found in our study matches those seen in the original study.

5.3.4 Usability

Finally, we compare the reported usability of the library in our study and the usability results of [28]. We use a t-test to determine the effect of the library on the usability score measured using the System Usability Scale (SUS) [172] and the scale developed in the original paper [28].

We first explore how the library used affects the SUS score. We start by testing for normality using the Shapiro-Wilk test and are unable to reject the null hypothesis ($p = 0.247$), so we assume normality for the data, so we used a t-test. We find no significant difference between Cryptography.io and PyCrypto ($p = 0.443$, $t = 0.78$). The average SUS score was 56.4 for

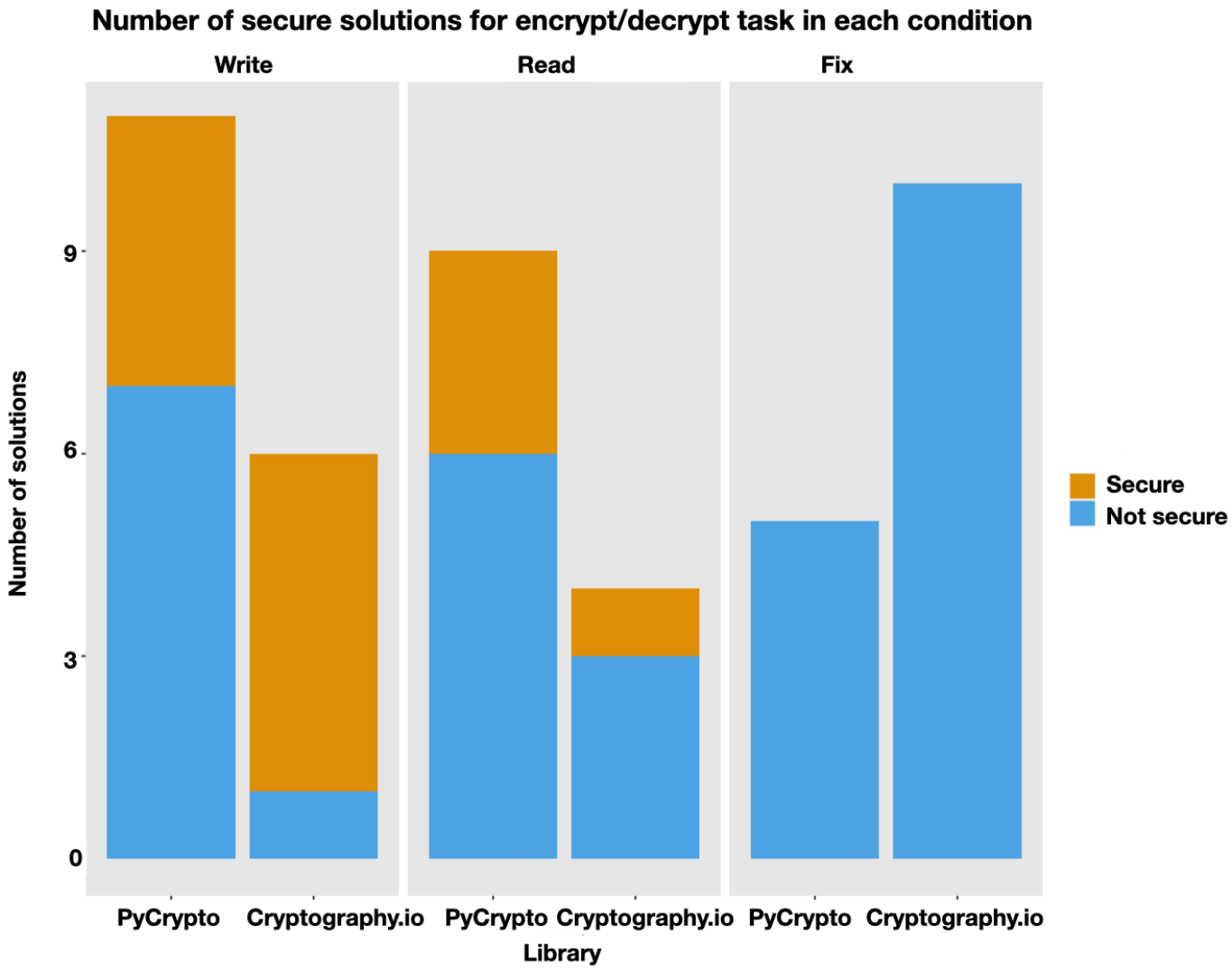

Figure 5.6: Number of secure solutions for encrypt/decrypt task in each condition

participants using PyCrypto and 48.9 for participants using Cryptography.io. This mirrors the nearly identical mean SUS scores (63.9 and 67.2) found in the original paper [28]. Our scores are likely lower than those found in the original paper given the age of the libraries in comparison to when the study was originally run. The distribution of SUS scores for both libraries can be found in Figure 5.8.

Next, we explore the how the library used affects the score on the usability scale developed in [28]. We again start by testing for normality using the Shapiro-Wilk test and are unable to

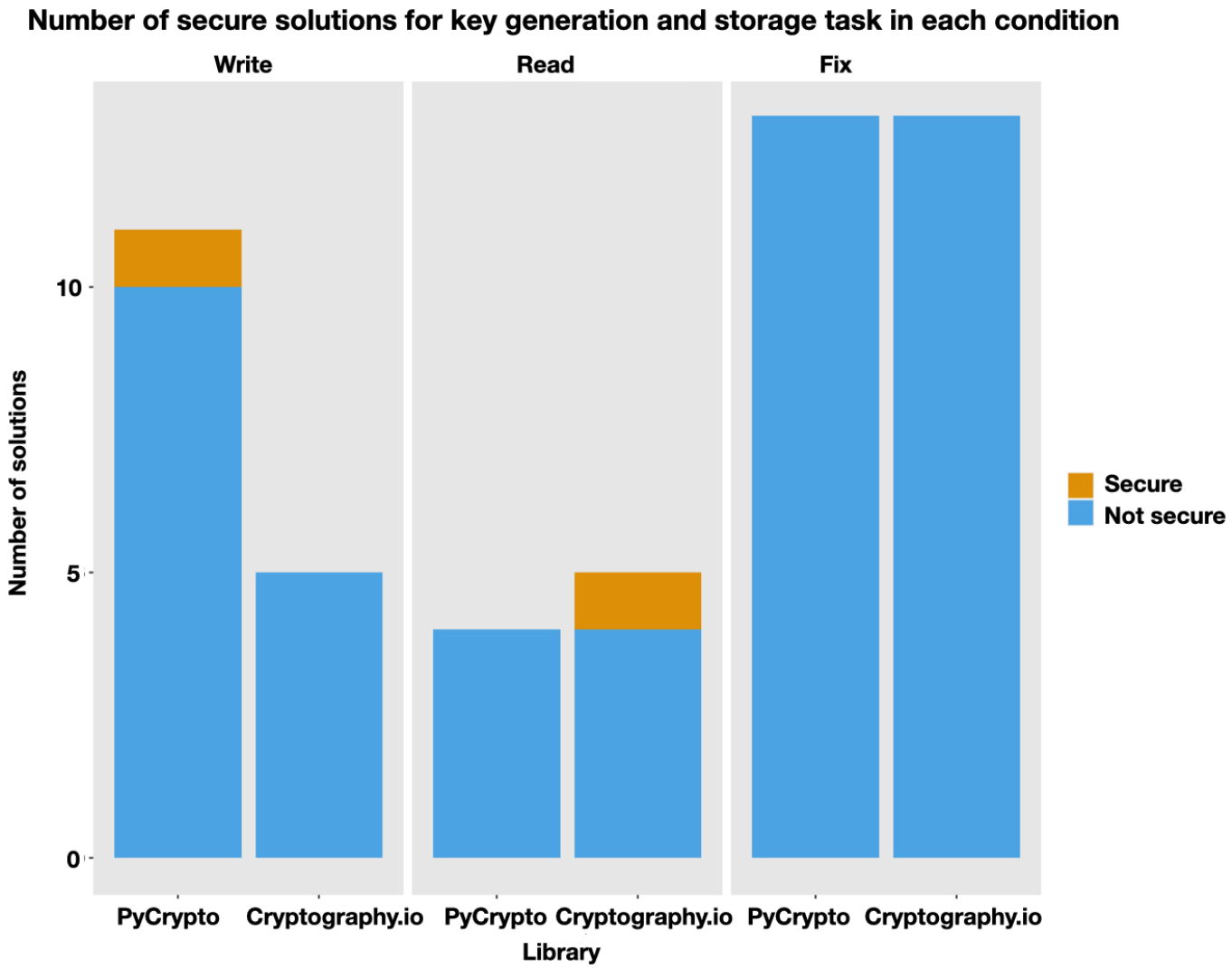

Figure 5.7: Number of secure solutions for key generation and storage task in each condition

reject the null hypothesis ($p = 0.517$), so we assume normality for the data and once again use a t-test. We find no significant difference between the usability scores for Cryptography.io and PyCrypto ($p = 0.956$, $t = -0.06$). The average score for participants using PyCrypto and Cryptography.io was 57.9 and 58.4 respectively, again mirroring the nearly identical means (64.2 and 67.7) found in the original paper. Again, our scores are likely lower than those found in the original paper given the age of the libraries used. The distribution of usability scores from [28] for both libraries can be found in Figure 5.9.

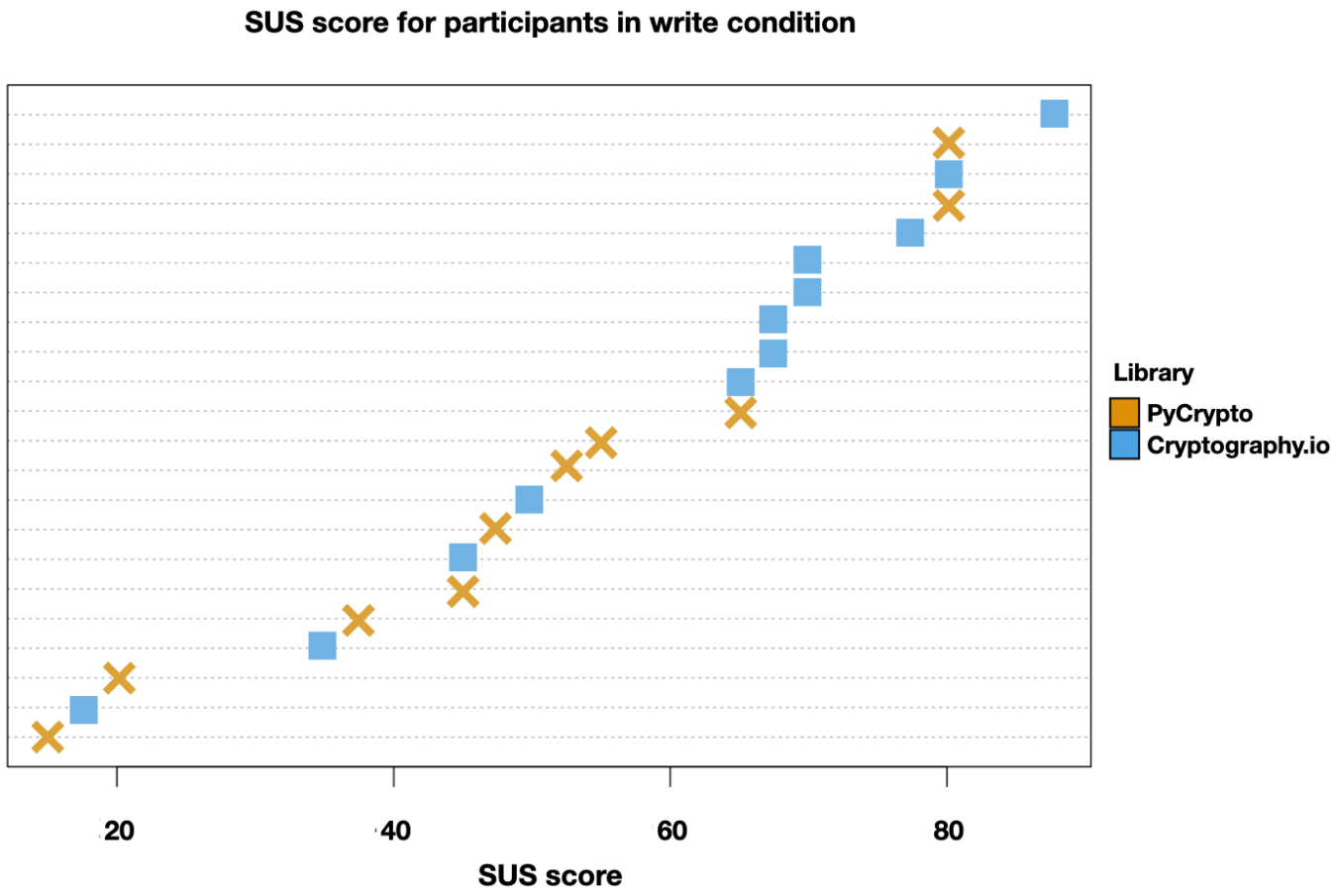

Figure 5.8: Distribution of SUS scores in write condition

5.4 Comparing among conditions

Next, we compare the results of the read and fix condition in our study to the results in the write condition of our study. We again compare across four axes: dropouts, functionality, security, and usability. Trends for each of the tests for each condition can be found in Table 5.5.

5.4.1 Dropouts

We first explore how the library assigned affected the ability of participants in the read condition and the fix condition to complete our study. We used FETs and compared the results

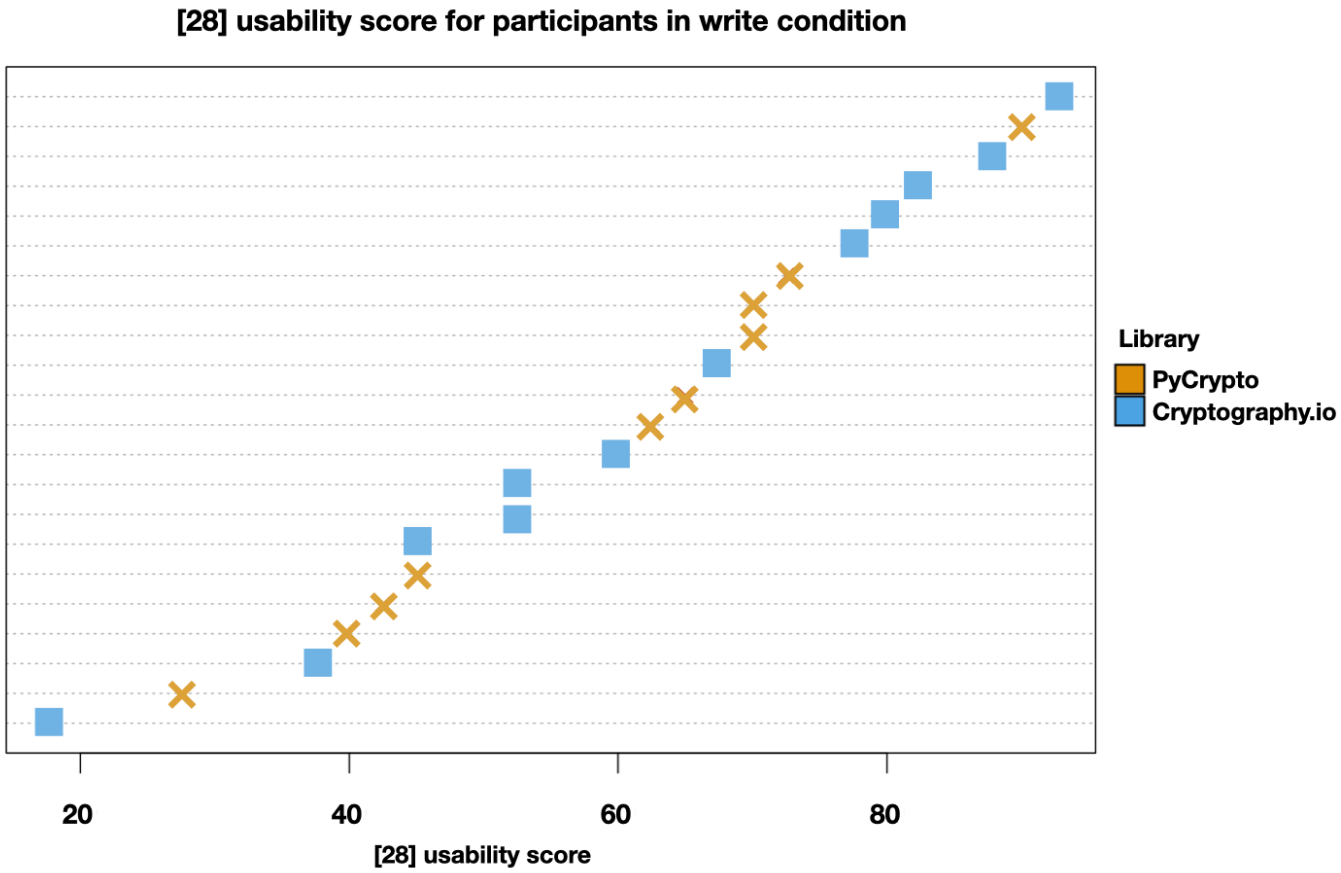

Figure 5.9: Distribution of usability scores from [28] in write condition

among all three conditions. In the read condition, only 1 person dropped out from the study, using Cryptography.io. This mirrors the results from the write condition where slightly more participants (1 from PyCrypto and 3 from Cryptography.io) dropped out from Cryptography.io than PyCrypto. In the fix condition, there were no dropouts. Details of the number of valid and completed participants can be seen in Table 5.4.

We find no significant difference in the number of participants that dropped out from the study in the the read condition between PyCrypto and Cryptography.io ($p = .482$). This is unsurprising given that only one participant dropped out. We did not test the effect of library on dropout for the fix condition as there were no dropouts in this condition.

5.4.2 Functionality

We next explore the extent to which participants were able to produce functional solutions in the fix condition and the read condition compared to those in the write condition. Similar to Section 5.3.2, we considered a solution in the fix condition to be functional if it ran, passed all the tests, and completed the task. For the the read condition, we considered the code to be functional if the participant identified and correctly addressed all the functionality bugs introduced by us as described in Table 5.1.

Comparing Read to Write Participants using PyCrypto were able to produce 13 functional solutions to both tasks, whereas participants using Cryptography.io were able to produce 9 functional solutions to both tasks. The number of functional solutions for each library can be found in Figure 5.2. To explore the effect of the library used on the ability of participants to produce functional solutions in the read condition, we ran a FET. In the read condition, we find no significant difference between participants using PyCrypto and Cryptography.io ($p = 0.581$, $OR = 0.70$, $CI = [0.20, 2.40]$). While the difference is not signifiant (as it is for the write condition), participants producing more functional solutions in PyCrypto mirrors the results found in Section 5.3.2 in which participants using PyCrypto produced 22 functional solutions and participants using Cryptography.io produced 11.

Thirteen participants were able to produce functional solutions for the encrypt/decrypt task. Participants using PyCrypto were able to produce slightly more functional solutions then those using Cryptography.io (9 vs 4 functional solutions). This mirrors the results from the write condition, where participants were able to produce more functional solutions for the encrypt/decrypt

task using PyCrypto (11 participants) than Cryptography.io (6 participants). The number of functional solutions for the encrypt/decrypt task for each library can be found in Figure 5.3.

Nine participants were able to produce functional solutions for the key generation and storage task. Participants using Cryptography.io were able to produce slightly more functional solutions then those using PyCrypto (5 vs 4 functional solutions). This does not mirror the results in the write condition. The number of functional solutions for the key generation and storage task for each library can be found in Figure 5.4.

Comparing Fix to Write Unlike participants in the read and write conditions, participants in the fix condition using Cryptography.io (23 functional solutions) were able to produce more functional solutions than those using PyCrypto (18 functional solutions). The number of functional solutions for each library can be found in Figure 5.2. To explore the effect of the library used on the ability of participants to produce functional solutions in the fix condition, we ran a FET. In the fix condition, we find no significant difference between participants using PyCrypto and Cryptography.io ($p = 0.056$, $OR = 0.24$, $CI = [0.04, 1.12]$).

Fifteen participants were able to produce functional solutions for the encrypt/decrypt task, with more participants using Cryptography.io producing functional solutions than those in PyCrypto (10 vs 5 functional solutions). The number of functional solutions for the encrypt/decrypt task for each library can be found in Figure 5.3.

Nearly all participants were able to produce a functional solution for the key generation and storage task, with only 1 person failing to produce a functional solution. This is the opposite of the results uncovered in the write condition. The number of functional solutions for the key generation and storage task for each library can be found in Figure 5.4.

5.4.3 Security

Next, we explore the extent to which participants who produced a functional solution were able to produce a secure solution in the read condition and the fix condition and how this compares to the write condition. We again used Fisher's exact tests to measure this effect.

Comparing Read to Write Participants using PyCrypto were able to produce slightly more secure solutions than those using Cryptography.io (3 vs 2). The number of secure solutions for each library can be found in Figure 5.5. In the read condition, we find no significant difference between participants using PyCrypto and Cryptography.io ($p = 1, OR = 1.05, CI = [0.09, 15.69]$) using a FET.

For the encrypt/decrypt task, of the 13 participants that were able to produce functional solutions, only 4 were able to produce secure solutions, with participants using PyCrypto able to produce 3 secure solutions and participants using Cryptography.io able to produce 1 secure solution. For the key generation and storage task, of the 9 participants were able to produce functional solutions, only 1 was able to produce a secure solution. This participant used Cryptography.io. This mirrors the result of the write condition where participants were able to produce more secure solution for the encrypt/decrypt task than the key generation and storage task as seen in Section 5.3.3. The number of secure solutions for the encrypt/decrypt and key generation and storage tasks for each library can be found in Figure 5.6 and Figure 5.7 respectively.

Comparing Fix to Write In the fix condition, participants were unable to produce any fully secure solutions, so we did not run at statistical tests comparing libraries. This is likely due to the difficulty of identifying already existent vulnerabilities in code that was written by someone

else. Additionally, participants in the fix condition were very focused on identifying functionality issues over security vulnerabilities as we will discuss in Section 5.7.

5.4.4 Usability

Finally, we compare the reported usability of the library in the read condition and the fix condition and the usability results of the write condition. We again use a t-test to determine the effect of the library on the usability score measured using the System Usability Scale (SUS) [172] and the scale developed in the original paper [28].

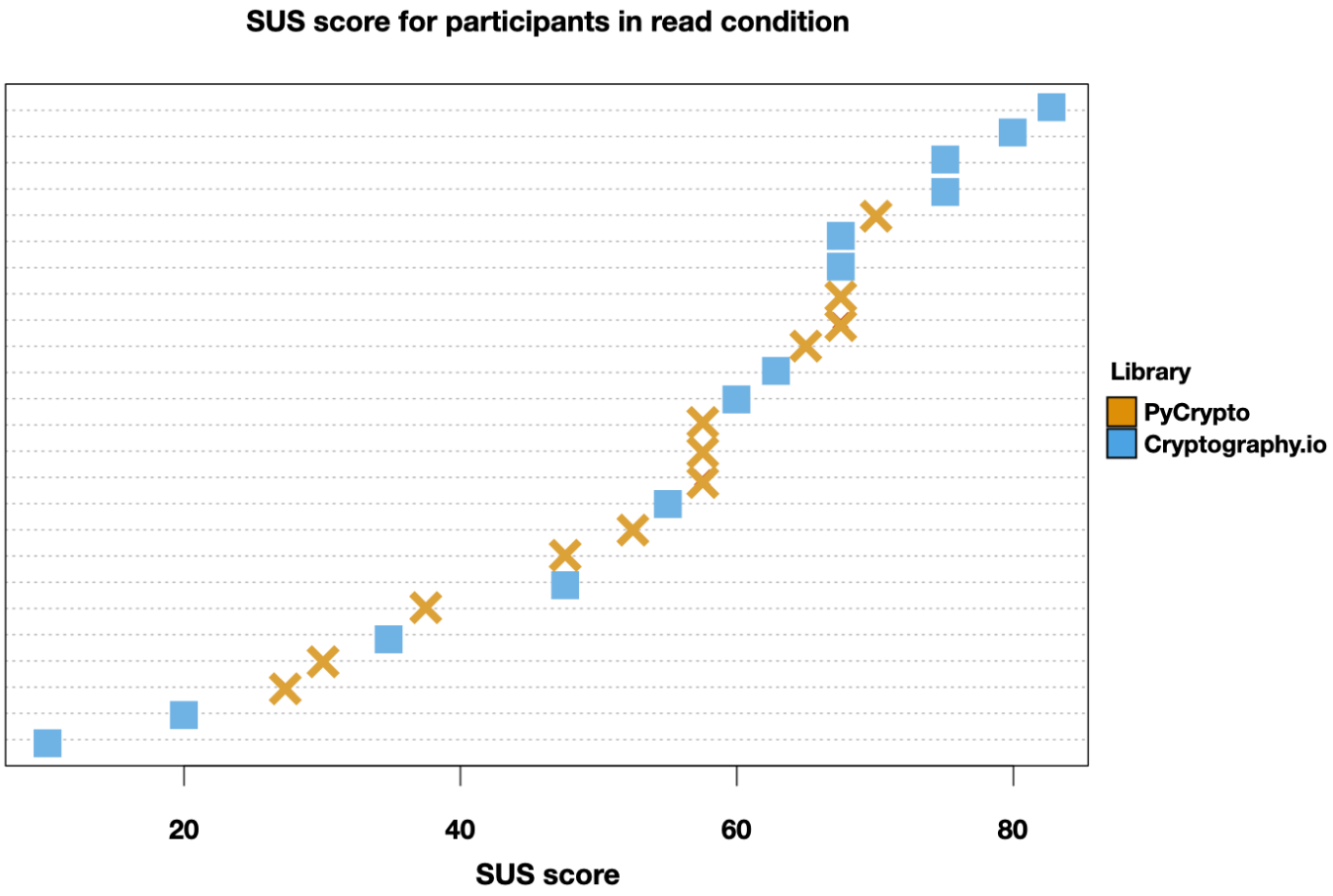

Figure 5.10: Distribution of SUS scores in read condition

Comparing Read to Write We first explore how the library used affects the SUS score. We start by testing for normality using the Shapiro-Wilk test and are unable to reject the null hypothesis ($p = 0.127$), so we assume normality for the data, so we use a t-test. We find no significant difference between Cryptography.io and PyCrypto ($p = 0.641$, $t = 0.78$). The average SUS score was 56.7 for participants using PyCrypto and 53.1 for participants using Cryptography.io. This mirrors the nearly identical mean SUS scores (56.4 and 48.9) with PyCrypto receiving a slightly higher score found in the write condition. The distribution of SUS scores for both libraries can be found in Figure 5.10.

Next, we explore the how the library used effects the score on the usability scale developed in [28]. We again start by testing for normality using the Shapiro-Wilk test and are able to reject the null hypothesis ($p = 0.022$), so we cannot assume normality for the data. Because of this rejection of normality, we use a Mann-Whitney U test instead of a t-test. We find no significant difference between the usability scores for Cryptography.io and PyCrypto ($p = 0.827$). The average score for participants using PyCrypto and Cryptography.io was 66.4 and 67.5 respectively, again mirroring the nearly identical means (57.9 and 58.4) and slightly higher usability score for Cryptography.io found in the write condition. The distribution of usability scores from [28] for both libraries can be found in Figure 5.11. Libraries likely received higher usability scores in the read condition than the write condition as participants felt it was easier to read code and evaluate than write code from scratch.

Comparing Fix to Write We first explore how the library used effects the SUS score. We again start by testing for normality using the Shapiro-Wilk test and are able to reject the null hypothesis ($p = 0.041$), so we cannot assume normality for the data. Because of this rejection of normality,

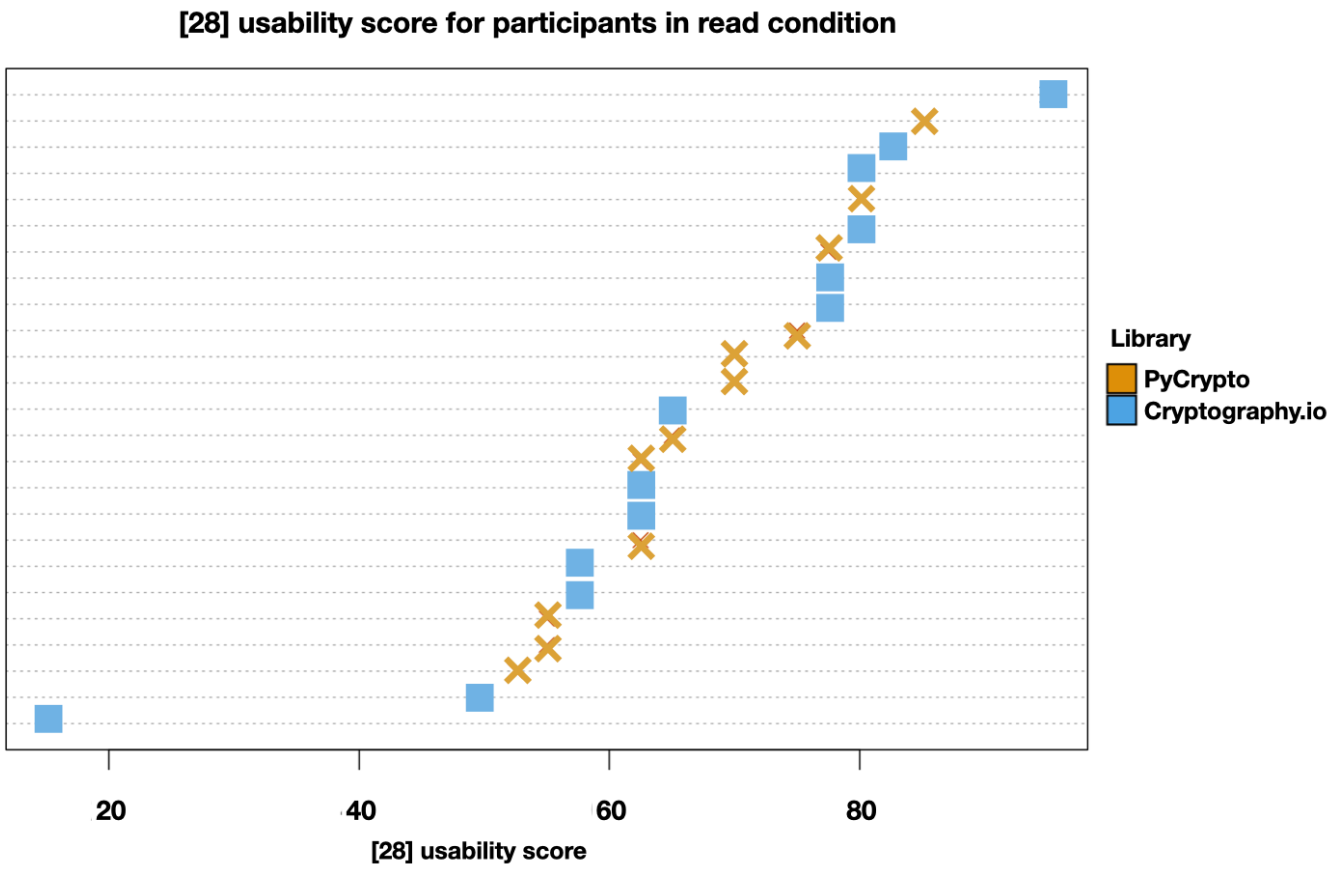

Figure 5.11: Distribution of usability scores from [28] in read condition

we use a Mann-Whitney U test instead of a t-test. We find no significant difference between Cryptography.io and PyCrypto ($p = 0.610$). The average SUS score was 65.4 for participants using PyCrypto and 68.1 for participants using Cryptography.io. This mirrors the nearly identical mean SUS scores (56.4 and 48.9) found in the write condition. The distribution of SUS scores for both libraries can be found in Figure 5.12.

Next, we explore the how the library used effects the score on the usability scale developed in [28]. We start by testing for normality using the Shapiro-Wilk test and are unable to reject the null hypothesis ($p = 0.204$), so we assume normality for the data. We find no significant difference between the usability scores for Cryptography.io and PyCrypto ($p = 0.633$, $t = -0.48$). The average score for participants using PyCrypto and Cryptography.io was 75.4 and

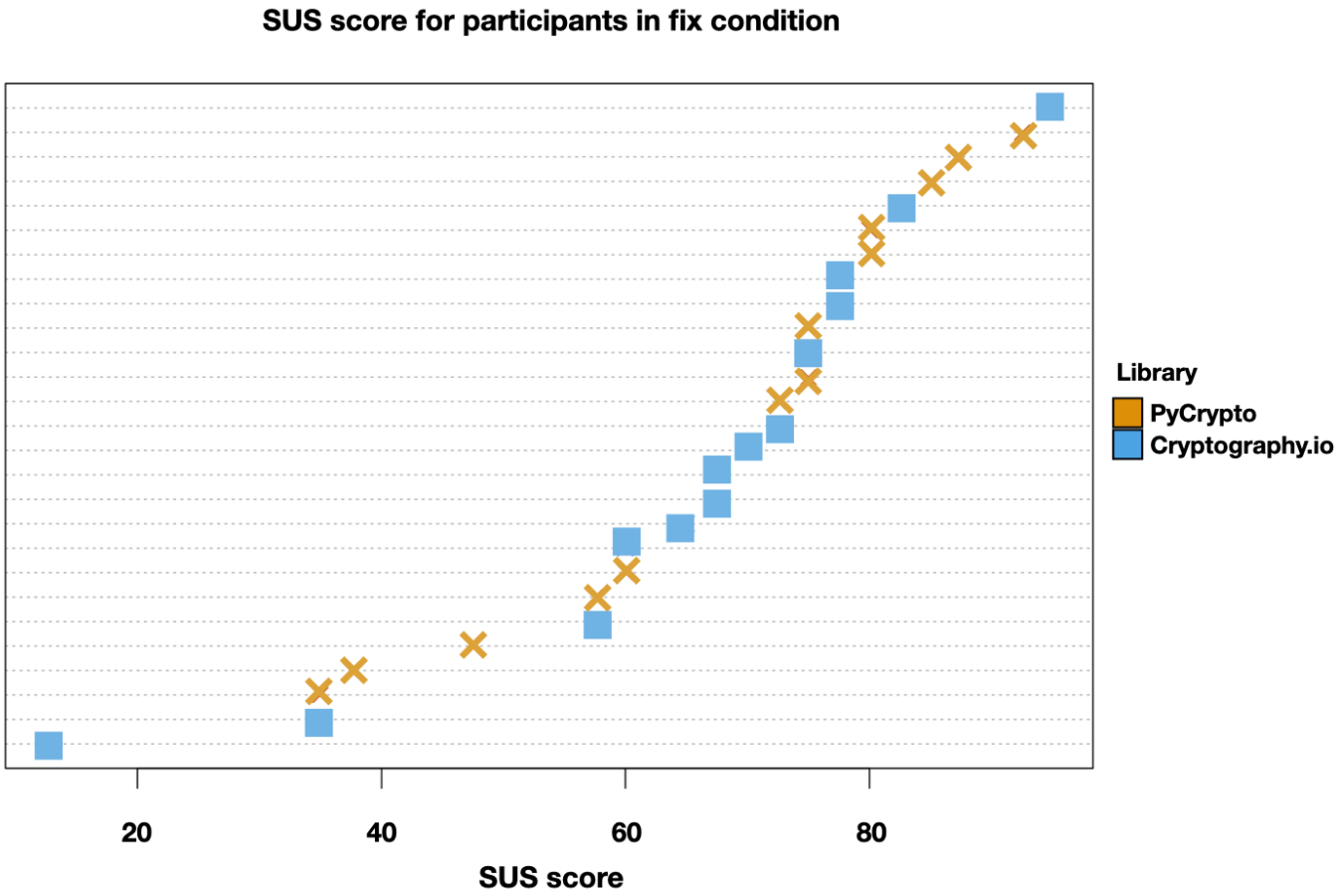

Figure 5.12: Distribution of SUS scores in fix condition

78.5 respectively, again mirroring the nearly identical means (57.9 and 58.4) and slightly higher usability score for Cryptography.io found in the write condition. The distribution of usability scores from [28] for both libraries can be found in Figure 5.13. Libraries likely received higher usability scores in the fix condition than the write and read condition as participants felt it was easier to identify bugs and vulnerabilities in already written code that they could run than writing code from scratch and just reading code to find issues.

While the difference between the means of the usability scores for both libraries are similar in the read, fix, and write conditions, it is worth highlighting that participants in the fix condition rated libraries as more usable than those in the write condition, and participants in the read

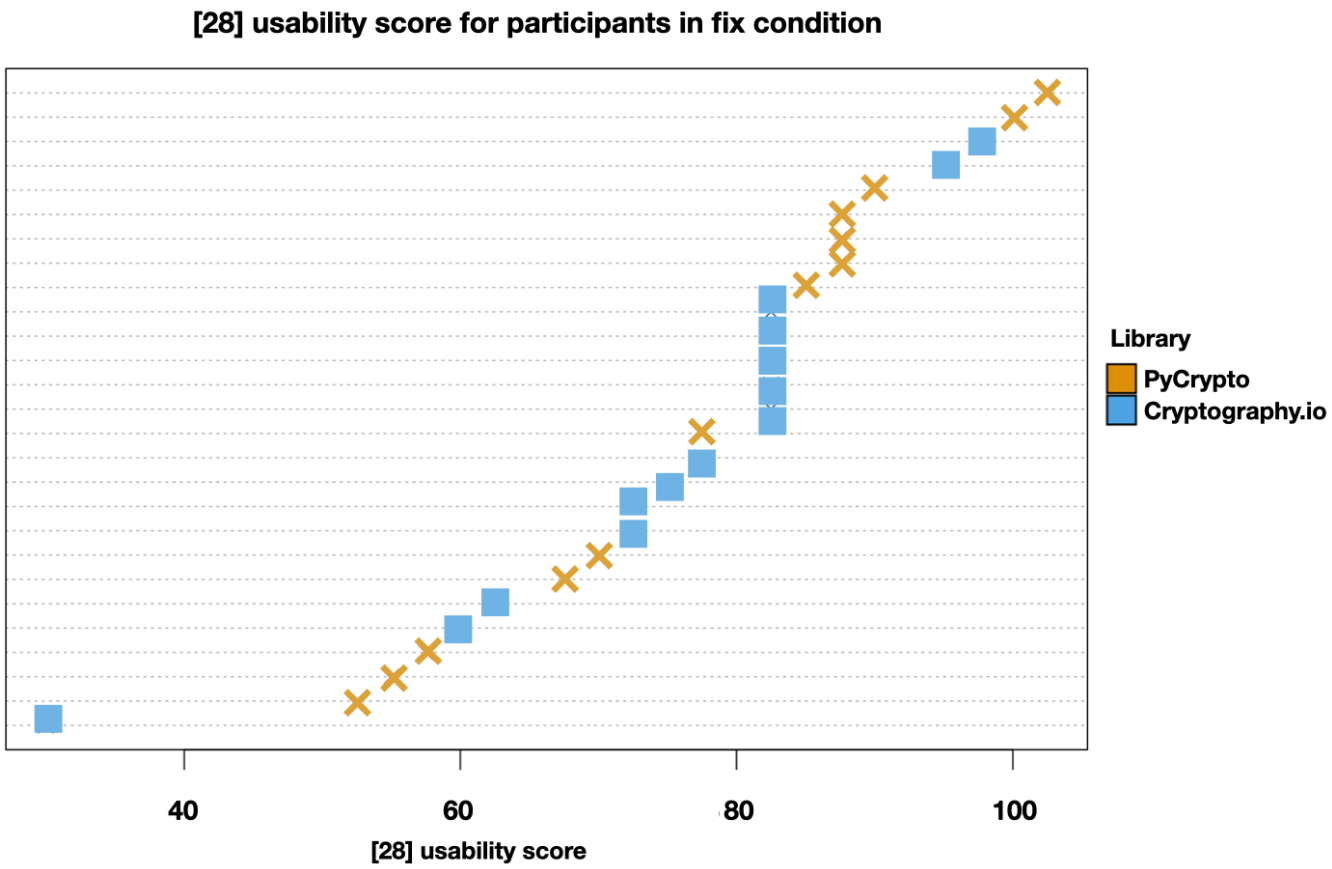

Figure 5.13: Distribution of usability scores from [28] in fix condition

condition rated libraries around the same level of usability as those in the write condition. This is likely due to the fact that participants in the fix condition had less to do to "complete" the assigned task than those in the write or read conditions. This likely increased their perceived usability of the libraries they were tasked with.

5.5 Effects on participants

In this section, we discuss the affect of the different experimental conditions on participants and response quality, such as, dropout rate, time to complete, reported frustration, and the number of participants excluded for quality reasons. Throughout this section we use N_r to represent the

number of participants in the read condition, N_w to represent the number of participants in the write condition, and N_f to represent the number of participants in the fix condition.

5.5.1 Data exclusion

We excluded 17 participants total: 14 from the read condition and 3 from the fix condition. Participants were excluded from the read condition primarily for running or editing code while completing the study (N_r = 9). This became such a prevalent issue they we modified the study infrastructure to make running code in the read condition impossible. Other reasons for exclusion included not participating/skipping through the questions just for compensation (N_r = 1, N_f = 3) and not understanding what they were being asked to complete (N_r = 1).

To explore the effect of experimental condition on whether data got excluded, we ran a Fisher's exact test. We find a significant difference between the conditions ($p = 0.0003$). To understand this effect, we performed a pairwise comparison between the read and fix conditions. We did not compare to the write condition because nobody was excluded from the write condition. When comparing read and fix conditions, we find that participants in the read condition were 4.57 times more likely to be exlucded than those in the fix condition ($p = 0.024$, $CI = [1.10, 27.63]$).

5.5.2 Dropout

We only had 5 dropouts total (N_r = 1, N_f = 4). This is drastically different than the experiences of the authors in [28], who had a dropout rate of nearly 84%. This is likely due to the fact that we paid our participants, but only if they completed the study, thus incentivizing them to finish.

To explore whether the library had any effect on whether a participants dropped out, we ran Fisher's exact test. To explore how the experimental condition affected whether participants completed the data, a crude estimation of how frustrating the condition is, we ran Fisher's exact test. We did not find a significant difference between conditions ($p = 0.121$). However this is likely due to the incredibly low dropout rate in our study.

5.5.3 Time to complete

On average, participants spent 2.9 hours completing the study ($\eta = 1.3$ hours, $\sigma = 4.7$ hours). However, this includes time spent taking a break from the study. Participants spent 3 hours on average in the read condition ($\eta = 1.2$ hours, $\sigma = 5.2$ hours), 4 hours on average in the write condition ($\eta = 2.0$ hours, $\sigma = 5.8$ hours), and 1.8 hours on average in the fix condition ($\eta = 59$ minutes, $\sigma = 2.6$ hours). As another measure of study stress on participants, we evaluate the effect of experimental condition on the amount of time spent participating in the study. We started by testing for normality using the Shapiro-Wilk test. We can reject the null hypothesis, so we cannot assume normality for our data ($p = 5.494x10^{-}14$). We tested for this effect using the Kruskal-Wallis test. We find no significant difference between conditions ($p = 0.935$, $\chi^2 = 0.007$). The distribution of time spent for participants in each condition can be seen in Figure 5.14.

On average, participants spent 1.6 hours working on the encrypt/decrypt task ($\eta = 22$ minutes, $\sigma = 4.06$ hours). Broken down by condition, participants in the read condition spent an average of 1.3 hours ($\eta = 0.35$ hours, $\sigma = 3.3$ hours), participants in the write condition spent 2.7 hours ($\eta = 42$ minutes, $\sigma = 6.3$ hours, and participants in the fix condition spent 51 minutes

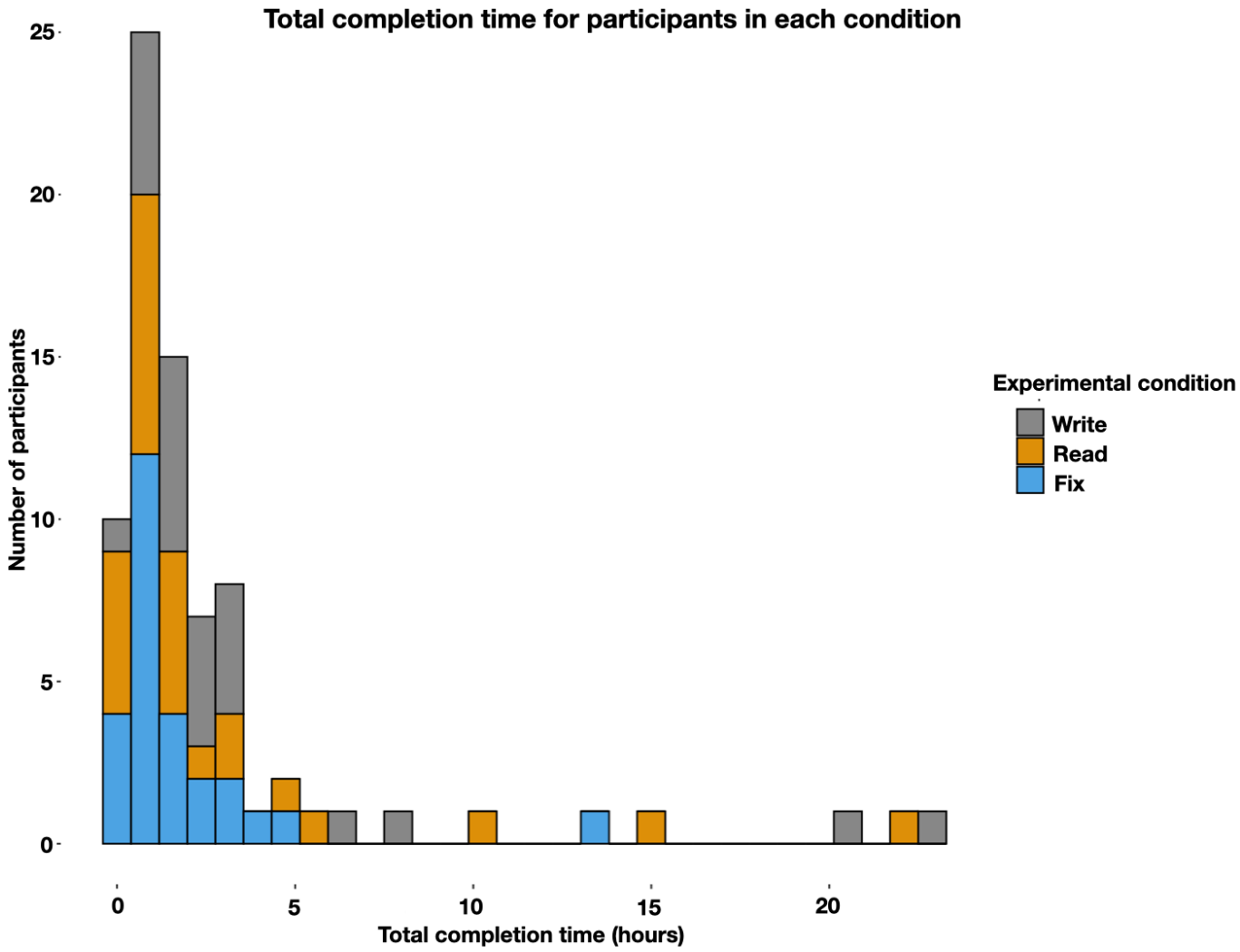

Figure 5.14: Distribution of total time spent by participants in the study

($\eta = 31$ minutes, $\sigma = 57$ minutes).

On average, participants spent 1.8 hours completing the key generation and storage task ($\eta = 26$ minutes, $\sigma = 4.1$ hours). In the the read condition, the write condition, and the fix condition, participants spent an average of 2.6 hours ($\eta = 31$ minutes, $\sigma = 5.2$ hours), 1.9 hours ($\eta = 40$ minutes, $\sigma = 4.1$ hours), and 50 minutes ($\eta = 15$ minutes, $\sigma = 2.4$ hours) on the key generation and storage task respectively. The distribution of time spent for participants in each condition can be seen in Figure 5.16.

While the differences are not significant, we can clearly see that participants in the the fix condition spent less than half the time working on the tasks than those in the the read condition and the write condition.

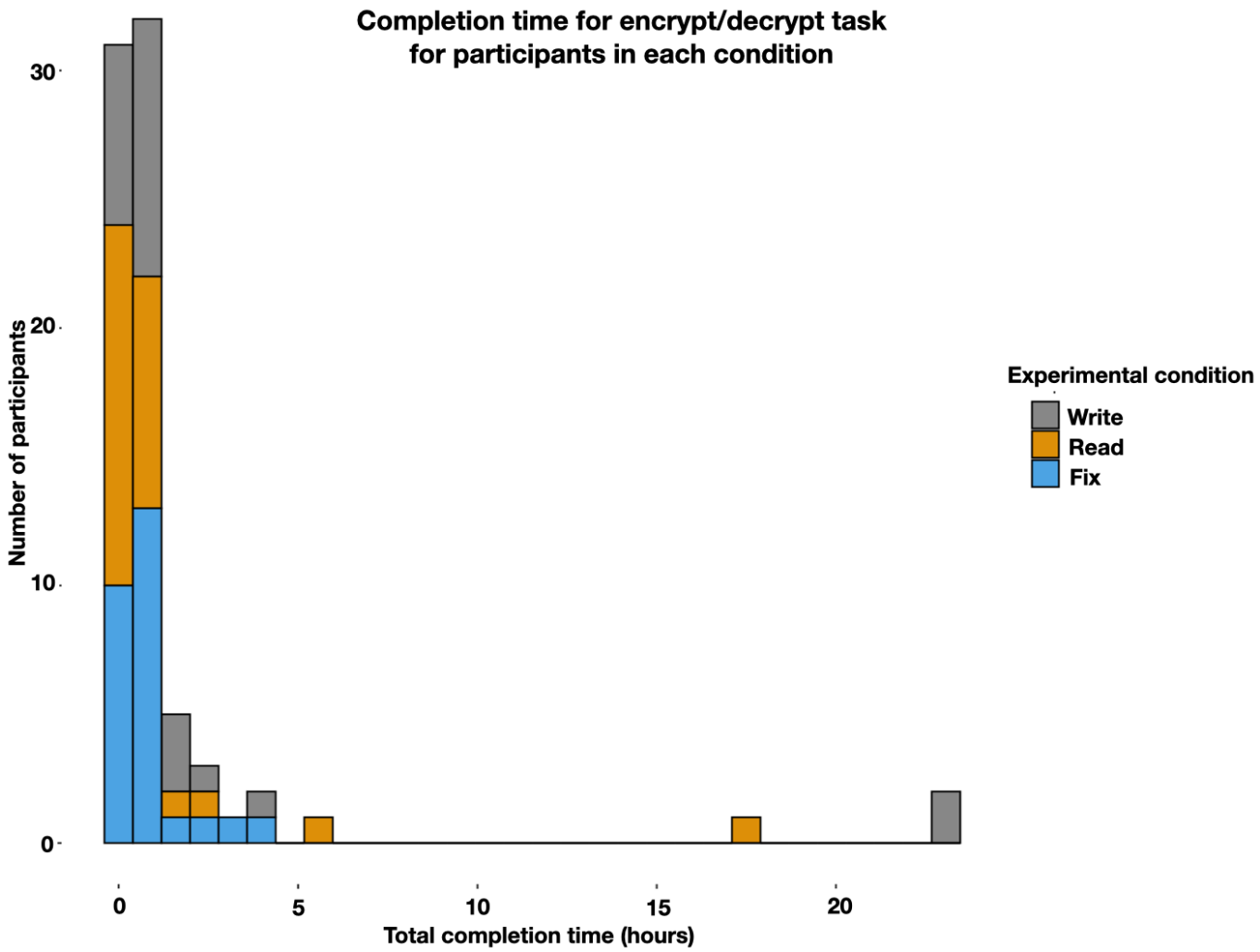

Figure 5.15: Distribution of time spent completing encrypt/decrypt task by participants in the study

5.5.4 Reported frustration

As a final measure of study stress on participants, we asked participants to self-report their frustration with the required tasks and whether they found the tasks fun. To understand the effect

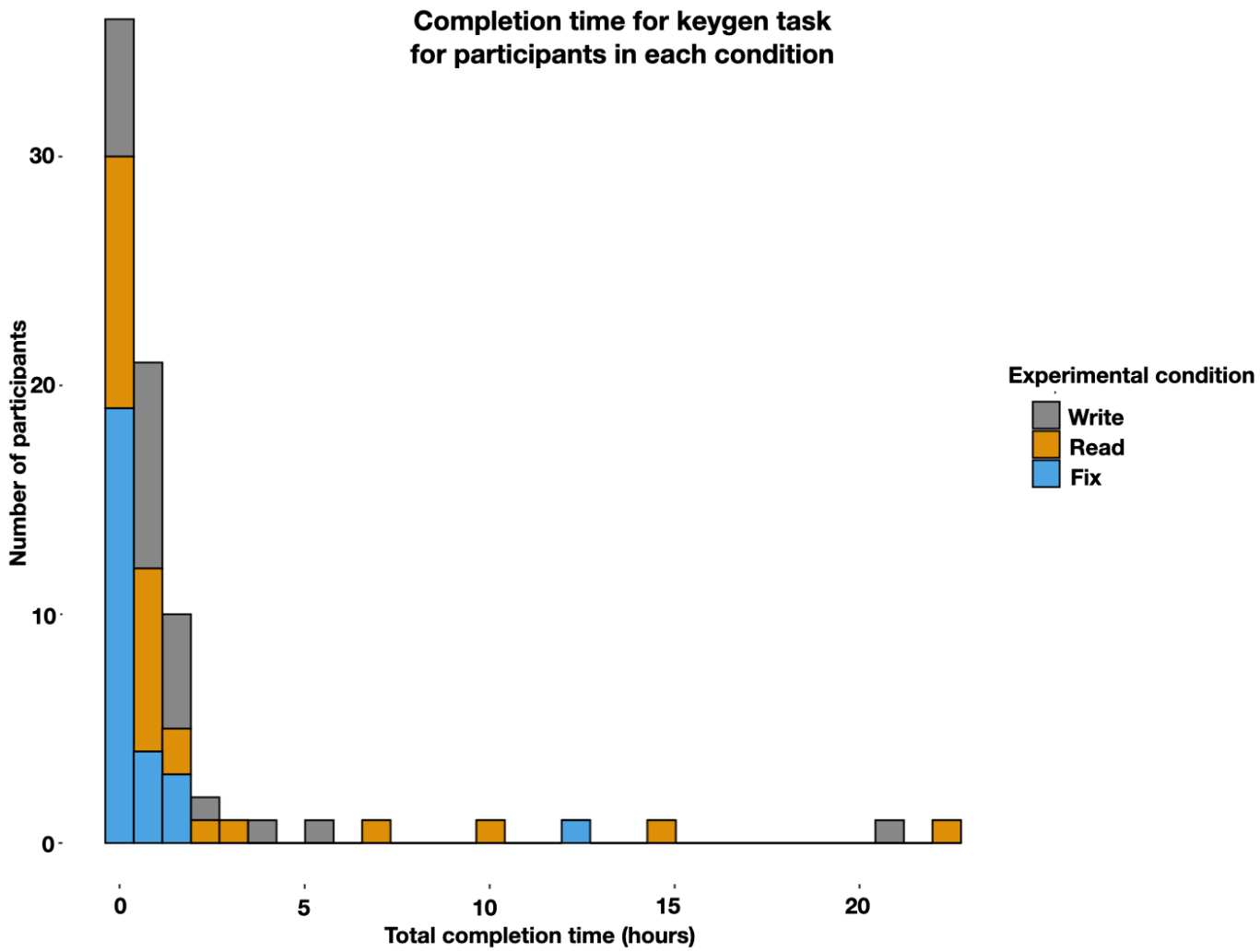

Figure 5.16: Distribution of time spent completing key generation and storage task by participants in the study

of the task (encrypt/decrypt or key generation and storage) and condition on frustration, we ran ordinal logistic regression for both self-reported frustration and fun. For the encrypt/decrypt task, 8 participants (33%) in the write condition, 7 participants (27%) in the read condition, and 4 participants (15%) in the fix condition reported being frustrated (agree and strongly agree). For the key generation and storage task, 15 participants (63%) in the write condition, 3 (12%) in the read condition, and 3 participants (11%) in the fix condition reported being frustrated (agree and strongly agree). The reported frustration in each condition for both tasks can be seen in

Figure 5.17. Our model for overall frustration indicated that participants in the write condition were 3.06 times and 6 times more likely to report higher frustration than those in the the read condition and the fix condition respectively. We see no significance of the task on the reported frustration. We report the model and p-values in Table 5.6.

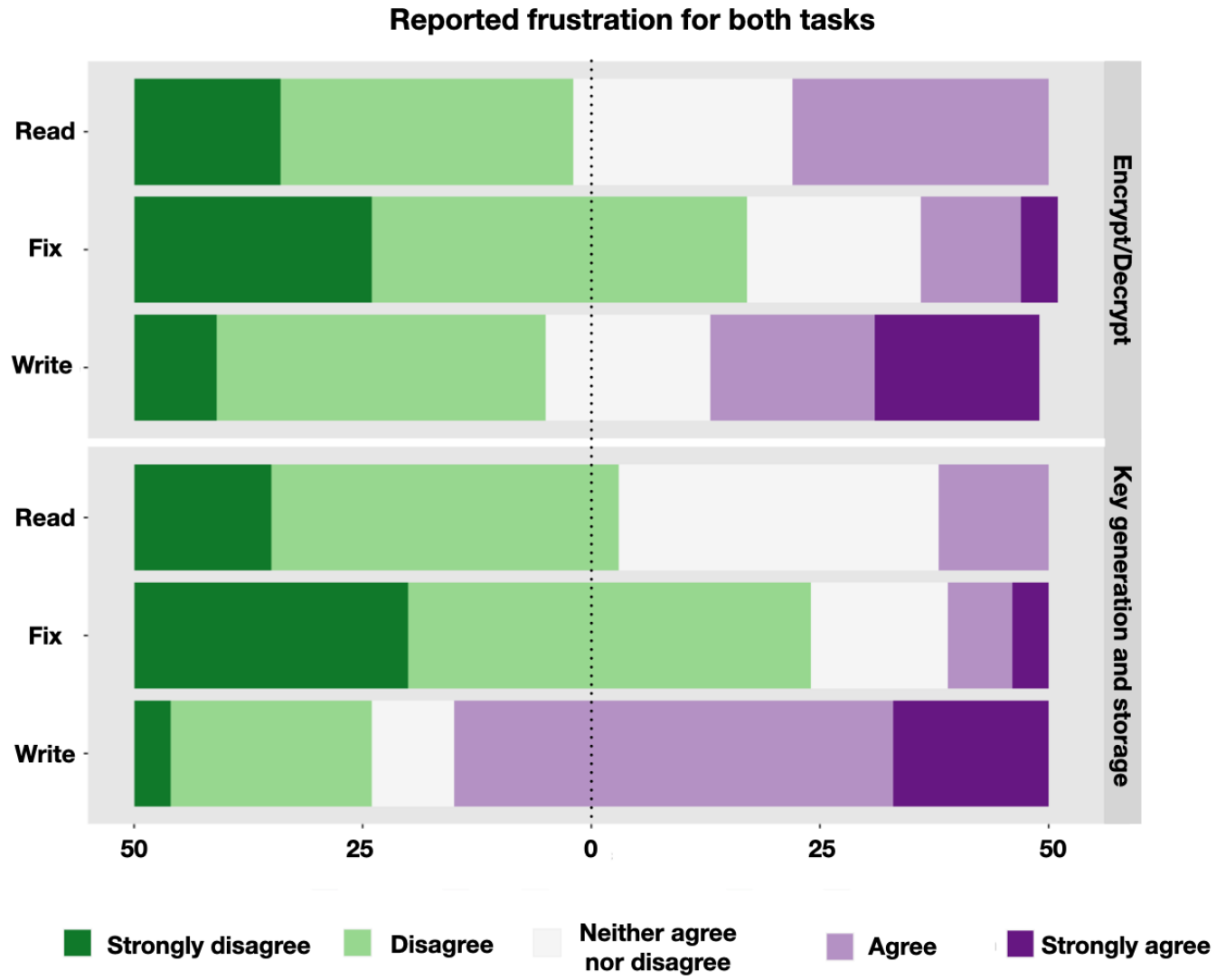

Figure 5.17: Reported frustration for both tasks in each condition

For the encrypt/decrypt task, 15 participants (63%) in the write condition, 20 participants (77%) in the read condition, and 23 participants (85%) in the fix condition reported being frustrated (agree and strongly agree). For the key generation and storage task, 11 participants (46%) in the

write condition, 18 participants (69%) in the read condition, and 17 participants (63%) in the fix condition reported being frustrated (agree and strongly agree). The reported fun in each condition for both tasks can be seen in Figure 5.18. Our model for overall fun indicated that participants in the fix condition were 2.55 times more likely to report higher fun than those in the write condition. We report the model and p-values in Table 5.7.

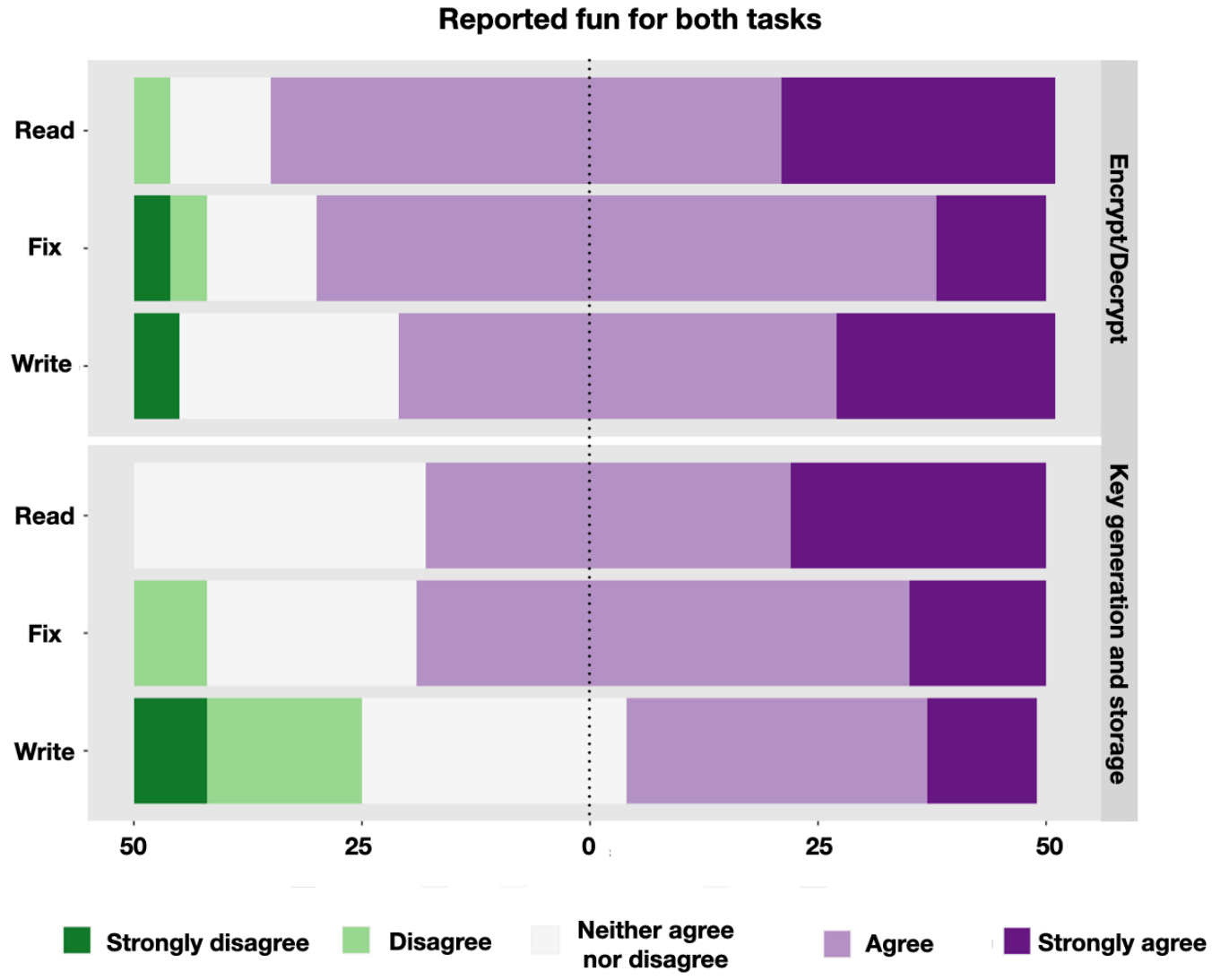

Figure 5.18: Reported fun for both tasks in each condition

This suggests that participants overall feel less frustrated in the read and fix conditions than in the write condition and actually enjoyed working in the read and fix condition. This means

that retention may be higher in the read/fix condition, since participants may experience less frustration or actually enjoy the tasks they're working on.

Factor	**O.R.**	**C.I.**	$p-value$
Baseline: Write			
Read	0.33	[-1.88, -0.38]	0.003*
Fix	0.17	[-2.58, -1.02]	<0.001*
Baseline: encrypt/decrypt			
Key generation and storage	1.06	[-0.53, 0.64]	0.847

Table 5.6: Final ordinal logistic regression for participant frustration

Factor	**O.R.**	**C.I.**	$p-value$
Baseline: Write			
Read	1.57	[-0.31, 1.21]	0.247
Fix	2.55	[0.17, 1.72]	0.018*
Baseline: encrypt/decrypt			
Key generation and storage	0.56	[-1.20, 0.03]	0.063

Table 5.7: Final ordinal logistic regression for participant self-reported fun

5.6 Vulnerabilities in Read, Write, and Fix

In this section we discuss the vulnerabilities introduced, identified, and fixed by our participants. Throughout this section, we use V_w to represent the number of vulnerabilities from the write condition, V_r to represent the number of vulnerabilities from the read condition, and V_f to represent the number of vulnerabilities from the fix condition. For vulnerabilities that were unique to the write condition, we do not include counts for V_r and V_f. The list of included vulnerabilities for the read and fix conditions can be found in Table 5.1.

5.6.1 Vulnerabilities introduced and not identified

In total, participants introduced 49 vulnerabilities in the write condition, left 259 vulnerabilities unidentified in the fix condition, and left 155 vulnerabilities unidentified in the read condition. The large disparity in vulnerabilities between the write and read and fix conditions is likely due to the fact that participants in the read condition and the fix condition started with vulnerabilities in their codebase due to the study setup, while participants in the the write condition started with a blank slate. Participants assigned to use PyCrypto (V_w = 36, V_r = 85, V_f = 135) introduced more vulnerabilities than those assigned to use Cryptography.io (V_w = 13, V_r = 70, V_f = 124) in all three conditions.

The most common type of vulnerability introduced in the write condition was one where participants attempted to implement cryptography protocols but made a weak cryptography choice (V_w = 25). This mirrors the read and fix conditions, where the least commonly identified vulnerabilities were also caused by a weak cryptography choice (V_r = 102, V_f = 157). In the write condition, vulnerabilities caused by a weak cryptography choice occurred overwhelmingly in the PyCrypto library (V_w = 22) rather than the Cryptography.io library (V_w = 3), mirroring the results from [28]. This is likely due to the structure of the documentation; the documentation makes identifying and using the most secure options difficult. For participants in the read condition and the fix condition, vulnerabilities caused by a weak cryptography choice occurred nearly equally in the Cryptography.io (V_r = 47, V_f = 74) and PyCrypto (V_r = 55, V_f = 83) library. This is likely due to the fact that finding vulnerabilities in code that was written by another person is difficult and these libraries were not designed to aid in this task. This included vulnerabilities such as failing to use a key derivation function (V_w = 8), using an insecure encryption algorithm (V_w =

5) or mode (V_w = 6, V_r = 39, V_f = 55), building a custom key derivation function (V_w = 4), using a weak key derivation function (V_r = 14, V_f = 28), having insufficient iterations for the key derivation function (V_w = 2, V_r = 37, V_f = 52), and using a weak hash algorithm in the key derivation function (V_r = 12, V_f = 24).

The second most common type of issue in the write condition, the read condition, and the fix condition was using a fixed or static value where the value needed to be random (V_w = 16, V_r = 53, V_f = 102). In the write condition, vulnerabilities caused by using a fixed or static value again occurred overwhelmingly in the PyCrypto library (V_w = 11) rather than the Cryptography.io library (V_w = 5). This is likely cased by a difficulty in navigating the provided documentation. Participants often turned to other resources to develop their code and these resources encouraged the use of static values. For participants in the read condition and the fix condition, these vulnerabilities occurred nearly equally in the Cryptography.io (V_r = 23, V_f = 50) and PyCrypto (V_r = 30, V_f = 52) library. This again highlights the fact that these libraries did not help with identifying vulnerabilities in other people's code, despite one of them being designed to help promote usability. This included vulnerabilities such as using a static salt in the key derivation function (V_w = 4, V_r = 27, V_f = 54), a static initialization vector for encryption (V_w = 7, V_r = 26, V_f = 48), and a static value for the encryption key (V_w = 2).

The prevalence of the issues and types of vulnerabilities were similar amongst the three conditions. However, while we notice a large difference in the types and number of vulnerabilities introduced by participants using the different libraries in the write condition, we don't see the same difference in the read and fix conditions. This is likely due to the overall difficulty in identifying vulnerabilities, which is independent of the library used. Given this, we can likely measure the types of vulnerabilities introduced by participants using the varying methods, but

distinguishing between the causes may be more difficult.

5.6.2 Vulnerabilities identified and fixed in Read and Fix

Participants correctly identified 97 out of 252 vulnerabilities in the read condition and 6 out of 265 vulnerabilities in the fix condition. In the read condition, vulnerabilities caused by the use of fixed or static values (V_r = 49) and weak cryptography choices (V_r = 48) were nearly equally identified. Vulnerabilities caused by these two issues occurred nearly equally in PyCrypto (V_{rfixed} = 24, V_{rweak} = 23) and Cryptography.io (V_{rfixed} = 25, V_{rweak} = 25). In the fix condition, vulnerabilities caused by a weak cryptography choice (V_f = 4) were identified slightly more than those caused by the use of a fixed or static value (V_f = 2). Vulnerabilities occurred slightly more in Cryptography.io (V_f = 4) than in PyCrypto (V_f = 2). Vulnerabilities caused by the use of a fixed or static value that were identified included the use of static initialization vector (V_r = 26, V_f = 2) and static salt (V_r = 23). Vulnerabilities caused by the use of weak cryptography choices that were identified included insufficient iterations in the key derivation function (V_r = 13, V_f = 2) and the use of a weak hash algorithm for the key derivation function (V_r = 12, V_f = 2).

As seen in Section 5.6, vulnerability identification varied widely between the read condition and the fix condition. To explore the impact of the condition (read and fix condition) and the library used on the number of security vulnerabilities identified, we ran a poisson regression. Our final model indicates that participants in the read condition identified 2.8 times more vulnerabilities than those in the fix condition ($p = 3.71x10^{-}11$). This is likely due to the fact that those in the fix condition were able to run their code whereas those in the read condition were not. Participants in the fix condition prioritized passing the tests. Every single participant in the fix condition started

the study by running their code (N_f = 27). For many participants in the fix condition, once their code ran they assumed their code was correct and moved on (N_f] = 18). This mirrors prior work where developers often prioritize functionality over security, often assuming that if their code runs and passes all the provided tests then it is correct and secure [173].

In fact, participants falsely identified 8 vulnerabilities in the read condition and 1 vulnerability in the fix condition. Some of the "vulnerabilities" identified by participants were security measures that are important to consider but outside of the scope of the assigned tasks such as including integrity checks for the encrypted data (V_r = 1, V_f = 0) or storing the salt in a safe location (V_r = 1, V_f = 0). Other "vulnerabilities" identified by participants were conceptual misunderstandings of what was needed for the code to actually be secure. For example, 2 participants in the read condition identified that the encryption of the key in the key generation and storage function was missing an initialization vector. They flagged this as a security issue. However, they failed to notice that the mode used for encryption, ECB, did not require an initialization vector and was an insecure mode of operation. Participants likely flagged more of these issues in read than fix as they paid extra close attention to the code because they could not run the code and get feedback.

Further, the extent to which vulnerabilities got fixed after identification varied between the read condition and the fix condition. While participants in the fix condition finished the study with substantially more vulnerabilities than those in the read condition, they were able to address every vulnerability they identified (V_f = 6). Participants in the read condition were able to address most of the vulnerabilities they identified (V_r = 87/97). However, they struggled to correct 10 vulnerabilities caused by weak cryptography choices. They struggled the most with addressing insufficient iterations in the key derivation function (V_r = 7). Best practices recommend at least 10,000 iterations in a key derivation function, as was true during the original

study [28]. Our participants were often able to identify that we has insufficient iterations in the code, but they often only increased the iterations to 1000, as recommended to them by the documentation. While participants in the read condition were able to better recognize this detail over those in the fix condition (V_r = 13, V_f = 2), they were unable to address it sufficiently. Of the 12 participants who used a KDF in the write condition, only 2 participants in failed to use sufficient iterations.

5.7 Bugs in Read, Write, and Fix

In this section we discuss the functionality bugs introduced, identified, and fixed by our participants. Throughout this section, we use B_w to represent the number of bugs from the write condition, B_r to represent the number of bugs from the read condition, and B_f to represent the number of bugs from the fix condition.

5.7.1 Bugs introduced and not identified

In total participants introduced 33 functionality issues in the write condition, left 49 bugs unidentified in the read condition, and 52 bugs unidentified in the fix condition. In the write and read condition, participants introduced only slightly more bugs when using Cryptography.io (B_w = 21, B_r = 26) than when using PyCrypto (B_w = 12, B_r = 23). In the fix condition, participants using PyCrypto (B_f = 36) introduced substantially more bugs than those who used Cryptography.io (B_f = 16).

Participants focus on passing tests in the write condition Functionality bugs varied widely across the write condition. However, with most of the functionality bugs in the write condition,

the participants' code would still run and pass the provided tests. The code would just fail to complete the required task. For example, the most common functionality issue in the write condition was caused by failing to store the encryption information correctly (B_w = 6), such as failing to store the key in the provided directory, as per the instructions (B_w = 4) or not storing the key in a file (B_w = 2). While the participants' code would still run and pass the provided tests, they failed to complete the task as instructed. These vulnerabilities occurred most commonly in Cryptography.io (B_w = 4).

Participants focus on passing tests in the fix condition Similarly, most of the bugs unidentified in the fix condition were caused by failing to complete the task rather than the code failing to run or pass the tests. The most common functionality issue left unidentified in the fix condition was caused by inconsistent checking for password length in the key derivation function (B_f = 22). This issue was equally unidentified for participants using PyCrypto (B_r = 12) and Cryptography.io (B_r = 10). The second most common functionality issue left unidentified in the fix condition was failing to identify that unencrypted data was being returned from the encrypt function and encrypted data was being returned from the decrypt function (B_f = 18). This issue was overwhelmingly unidentified in the PyCrypto library (B_f = 14). The code would run and pass our provided tests but does not correctly implement encryption and decryption or key derivation. This again mirrors prior work and supports our earlier findings where developers often prioritize functionality over security, often assuming that if their code runs and passes all the provided tests then it is correct and secure [45, 173].

Participants identify a variety of bugs in the read condition Conversely, in the read condition, bugs that caused the code to not run or not pass the provided tests went unidentified as often as those that caused the code to fail to complete the task. The issues least identified in the read condition were cases where the wrong variable name was passed to or used in a function, causing the code to crash if run (B_r = 26). This vulnerability was equally unidentified for participants using PyCrypto (B_r = 13) and Cryptography.io (B_r = 13). For example, about half the participants failed to identify a bug where a Python keyword was passed to the open function in Python (B_r = 10). This causes the code to crash when run. Participants were equally unable to identify the inconsistent check for the password length (B_r = 11) and returning (un)encrypted data in the encrypt and decrypt functions (B_r = 12). This is likely caused by the fact that participants in the read condition are unable to run code, causing them to review the code closely, resulting in them identifying a variety of functionality issues.

5.8 Discussion and recommendations

In this section, we discuss our results compared to prior work and make recommendations for future experiments based on our results.

5.8.1 Comparing our results to [28]

While we were not able to directly replicate the statistical tests found in the prior study [28], we were able to draw parallels between the results of that study and our write condition. For example, we see parallels in the functionality results for the two different tasks, the security results for the two different tasks, and the SUS and other usability scale scores. We see some

qualitative trends in the write condition showing that participants using PyCrypto produce less secure code than those using Cryptography.io and participants produced more secure solutions for the encrypt/decrypt task than the key generation and storage task, mirroring the results from the original paper. Additionally, we see the same types of vulnerabilities for the encrypt/decrypt and key generation and storage tasks in both our study and the original. Further research with a bigger sample size needs to be done to be able to directly compare the statistical results of our paper and the original paper, possibly expanding the number of libraries used to compare.

5.8.2 Comparing results among conditions

While much of our statistical results proved to not be significant, likely due to our small sample size, we see parallels in the results of the different conditions. For example, participants in the read and write conditions were both able to produce more functional solutions using PyCrypto. Participants in the read and write conditions were able to produce more secure solutions for the encrypt/decrypt task than the key generation and storage task. The results of the the read condition also highlighted the impact of insecure examples in documentation for the PyCrypto library, mirroring the results from [28]. However, it seems that the read and fix conditions have less power in determining the security differences of the libraries and thus a bigger sample size is needed to measure this than in the write condition. Further research with a bigger sample should be done to confirm that the read and fix condition can be used to compare the security properties of conditions.

Finally, we see similarities between the vulnerabilities introduced by participants in the write condition and not identified by participants in the read condition and the fix condition

conditions with the first and second most common vulnerabilities being shared among conditions. This points to some ability to use these conditions to measure various secure development phenomena such as the types of vulnerabilities that developers introduce and how libraries affect their ability to produce secure and functional code.

5.8.3 Using the fix condition and the read condition to measure secure development

Overwhelmingly, participants in the fix condition caught more functionality bugs than security vulnerabilities. Further, the functionality bugs caught by participants in the fix condition were mostly bugs that caused the code to fail to run rather than bugs that prevented the code from completing the task. Conversely, participants in the read condition were equally proficient at identifying functionality bugs that caused the code to not run and bugs that caused the code to not complete the assigned task. Further participants in the read condition identified 16 times the vulnerabilities than those in the fix condition despite being unable to run the code. Tasking participants with a manual code review resulted in code with fewer vulnerabilities than providing participants with tests and allowing them to run and test their code as frequently as they'd like. This echoes our and other prior work, in which participants struggled to identify areas of further testing when provided with tests [45, 173]. This suggests that fix may be useful for identifying what security vulnerabilities developers are able to quickly identify and address, testing their overall awareness for security concerns and allowing us to understand a lower bound for developers' secure development abilities. The read condition may be useful for understanding how well developers can do with security when they are required to pay close attention to details, allowing us to understand an upper threshold for developers' secure development abilities.

5.8.4 Read and fix are less frustrating, more fun, and shorter

Participants spent, on average, about the same time in the read condition and the write condition but spent nearly half the time in the fix condition. Unsurprisingly, participants reported feeling less frustrated in the fix condition than the write condition. However, despite spending nearly equal time working on tasks in the read condition and the write condition, participants also reported feeling significantly less frustrated in the read condition. Further, participants in the fix condition reported finding the tasks significantly more fun than those in the write condition. This points to some enjoyment in participating in this study in these conditions, which is a substantial improvement over the original study in which they had an 84% dropout rate. Retention for studies would likely be better if participants had fun while performing tasks in the study.

5.8.5 Building a study for reading or fixing code

In order to build and conduct a study that uses reading or fixing code, the study has to start with vulnerabilities or security issues. We built our vulnerabilities around a prior study in which participants were tasked with writing code. However, if researchers want to use read or fix to measure new phenomena an existing write study may not exist. Researchers could likely base the vulnerabilities in the study on existing 'real-world' vulnerabilities from the CWE of CVE lists or popular programming sites like StackOverflow. Additionally, researchers could base their vulnerabilities from interview studies or surveys that uncover the mental models that developers operate with as they build secure code.

Chapter 6: Contributions and Impact

The work presented in this thesis sheds light onto why and how developers introduce, find, and fix different types of vulnerabilities, the benefits and drawbacks to adopting a secure programming language, and possible alternatives to code writing studies to measure secure development.

In Chapter 3, we uncover that there is not a "one size fits all" approach to secure development owing to the fact that security vulnerabilities differ in more than just content. For example, our work highlights the importance of best practices, such as detailed design, to help prevent *Mistakes* and *No Implementations*, but shows that these best practices do not help with *Misunderstandings*. For *Misunderstandings*, security experts should be included at design time and throughout the development cycle to catch *Misunderstandings* before they propagate to the code. This work was one of the first empirical confirmations of folk advice often given in the community.

In Chapter 4, we identified benefits and drawbacks of adopting a secure programming language, uncovering that steep learning curves of secure tools and languages can drastically inhibit their adoption as adoptees must pay a steep upfront cost with no guarantee of benefitting from the language or tool. However, we also uncovered that good documentation for tooling, helpful feedback from tools, and a supportive and strong community for the tool can help reduce the initial buy-in for adoptees. The results of this work can serve as a guide for the creation of future tools to ease their adoption.

Chapter 5 took a first step toward building alternative methodologies for performing human-centered secure development studies through reading or fixing code rather than writing code. The results from this chapter support the possibility of using reading code for and/or fixing vulnerabilities as an alternative to writing code from scratch, although possibly not for every study. Participants tasked with fixing the issues in the code were test driven, often moving on the next task as soon as they passed the provided tests. Unsurprisingly, these participants were able to find nearly all of the functionality issues and very few of the security vulnerabilities. Conversely, participants in the read condition spent time combing through the code, as they were unable to run the code or tests. They found a wider breadth of both functionality issues and security vulnerabilities, finding more security vulnerabilities but fewer functionality issues than those in the fix condition. This suggests that the fix methodology may be most useful in understanding what vulnerabilities developers are able to identify quickly when security is not a primary concern or task, whereas the read methodology may be most useful in understanding what vulnerabilities developers can identify when they are more focused on security and are thoroughly searching for issues.

6.1 Importance of Security Experts

A consistent theme through Chapter 3 and Chapter 4 is the importance of security experts to promoting secure development. In Chapter 3, the main takeaway was the fact that there is not a one-size-fits-all solution for secure development. *Misunderstandings* require the help of a security expert to identify and understand, *Mistakes* benefit from the use of security tools and comprehensive testing to uncover, and *No Implementations* require detailed design and thorough

testing to prevent and uncover. The main finding of Chapter 4 is the difficulty of adopting a new secure tool or language at a company. While secure languages provide many benefits like improving secure development mental models and improving overall productivity, they face substantial push back from companies and management as they require large buy-in to get developers up to speed. However, this push back and buy-in can be reduced through the use of security-minded advocates and built in support for onboarding new developers from more experienced developers.

Taken together, these results highlight the importance of security-minded developers and security professionals at companies and during the software development process. Security experts can help developers address their security misunderstandings at design time, before they ever make it into code. They can also help identify any gaps in security considerations and promote the usage of security best practices such as the use of security tools and comprehensive testing. However, in most companies today, security experts and professionals work separately from the development team, often only performing security evaluations after the code is built [36, 45]. Future work should explore what this relationship between developers and the security team looks like and how they communicate with one another. More importantly, future work should consider how to better integrate security experts into the development process early on and integrate developers into the security testing process to promote a more holistic approach to secure software development.

6.2 Improving current resources and processes

While this thesis presents important foundational research that explores *why* secure development is hard and *why* current secure tooling and resources do not meet the needs of developers, it fails to address *how* to actually improve secure development. In Chapter 4, we make recommendations for improving the adoption of secure tools such as investing in infrastructure, training developers early (in college) on complex security and ownership models, and possibly altering secure tools to make them easier to learn. Future work should explore the efficacy of these suggestions to understand if they actually help reduce the investment in these tools and further promote security during development.

Further, this thesis focused entirely on exploring how secure programming languages and security best practices help developers with secure software development. The recommendations in this thesis center on the expectation that developers have security experts working in their proximity. However, many developers do not work closely with security experts and often have to turn to online Q&A platforms to help supplement their security knowledge, which can often lead to vulnerabilities in their code [24]. Future work must consider how to improve these online resources, often used as proxies for professionals, rather than just measuring their impact by exploring ways to signal to developers the security implications of the suggestions they receive from these sites.

6.3 Continued need for studying humans in secure software development

Finally, the work in this thesis highlights the continued need for the study of humans in secure software development. This thesis covers just a small set of human-centered concerns in secure software development, and, as discussed above, only produces foundational answers to *why* secure development is hard and *how* we can possibly make it easier. The continued study of secure development from a human-centered perspective is necessary to understand more about *why* secure development is difficult from organizational perspectives and *how* we can make it easier by exploring actionable steps for improving resources and tooling. It is especially important given the every increasing occurrence of security incidents and the recent advent of systems that can automatically generate code for both malicious attackers and developers [174, 175].

Additionally, this thesis took first steps to explore how to improve the generalizability and validity of human-centered secure software development studies. However, much work remains to help build best practices for conducting these studies to help ease the burden on participants and make studies reflect real-world conditions as much as possible while maintaining the ability to make strong statements about causes of security issues.

Chapter 7: Conclusion

Despite a number of resources such as security tooling, secure programming languages, secure development processes, and security educational materials, secure software development remains a difficult task exemplified by the continuously growing number of vulnerabilities found in production code. The success of these interventions relies heavily on both technical and socio-technical factors. In this thesis, I explore the socio-technical factors that influence secure software development, as well as approaches to studying these socio-technical factors.

7.1 Contributions

The first key conribution of this work is understanding how and why developers introduce, find, and fix different types of vulnerabilities. Chapter 3 emphasizes that there is no one-size-fits-all solution to secure development and that a number of different approaches are required to identify and prevent vulnerabilities. This study was one of the first empirical confirmations of security best practices and folk advice often distilled by professionals.

The second key contribution of this work is understanding how current tooling meets the needs of developers by understanding the benefits and drawbacks of adopting a secure programming language. In Chapter 4, the results highlight the benefits of adopting a secure programming language. Our results show that a key drawback of secure languages is the steep learning curves

associated with understanding the new security paradigms that come with the tooling. However, key benefits, such as good official documentation, good compiler error messages, and a helpful community can help reduce the investment into secure languages.

A final key contribution of this work is an initial exploration into methodological alternatives to writing code for secure development studies. Chapter 5 presents two alternatives to traditional code writing studies for studying human-centered secure software development, reading code and identifying security vulnerabilities and reading code and fixing security vulnerabilities. These alternatives offer relatively equivalent results while minimizing the stress on participants.

This work serves as a foundational exploration of difficulties faced by developers when building secure software and foundational exploration of best practices when studying this phenomenon.

Appendix A: Study Instruments

A.1 Chapter 4: Interview protocol

Most interviews were conducted by one interviewer; some interviews were assisted by a second interviewer. The second interviewer took notes and asked some additional and follow-up questions. Each interview was audio recorded, with permission.

Rust Background

- How did you initially learn about Rust?
- Why did you/your team/your company decide to adopt Rust?
- Was it hard to convince the necessary people/groups (bosses, team members, others?) at your company to use Rust?
  - What concerns did they have?
  - What were they excited about?
- Have any attitudes/policies of (team members, management) changed since you attempted this project in Rust?
  - How so?

- Can you please describe at a high level the project you/your team/your company are/is working on in Rust?
  - Is the project currently ongoing?
    - If yes, would you (briefly) characterize it as going well? Why (not)?
    - If no, did you finish it?
      - If no, why do you think you weren't able to complete the project?
      - If yes, do you consider the outcome a success? Why (not)?
  - Why did you pick this project to write in Rust?
- Can you tell me more about what happened when you tried to adopt Rust?
  - How long did it take you/did you spend trying/do you think you'll need to complete this project in Rust?
  - What went particularly well when adopting Rust at your company/on your team?
  - What went particularly poorly when adopting Rust at your company/on your team?
  - Did you receive positive feedback from adopting Rust?
    - From whom?
    - What were they happy about?
  - Did you receive negative feedback from adopting Rust?
    - From whom?
    - What were they happy about?
  - What would you do differently if you were to attempt another project in Rust?

        * Would you even try again at all?

- What would you tell someone in your position at a different company that is also thinking about adopting Rust?

Experiences with Rust (Only ask if they are programmer/familiar with coding)

- Have you ever felt like there was a programming task or something you wanted to program in Rust but could not get it to work?
  - What was it?
  - What did you try in order to debug/fix this problem?
- Can you tell me how Rust specific things affect your ability to fit a problem specification into a solution in Rust?
  - Ownership?
  - Lifetimes?
- Can you tell me more about your process of going from a problem specification to a solution in Rust?
- Do you find it difficult to find a solution to a programming problem in Rust?
  - Why?
- How easy is it for you to find solutions to any problems or errors you encounter while programming in Rust?

- What features do you like most about the Rust programming language?
  - Libraries/APIs?
  - Online community?
- What features do you like least about the Rust programming language?
  - Libraries/APIs?
  - Online community?
- In your opinion, what are the biggest strengths of Rust?
  - Libraries/APIs?
  - Online community?
- In your opinion, what are the biggest weaknesses of Rust?
  - Libraries/APIs?
  - Online community?
- Have you ever used unsafe blocks in this project?
  - Why did you use them?
  - What solutions did you try before using the unsafe blocks?
- Does this project code interoperate with any other code from another language?
  - What was hard about getting the code to interoperate?
  - What was easy about getting the code to interoperate?

- Did you/the team port code from another programming language to Rust for this project?
  - What language did you port from?
  - What was hard about porting your code to Rust?
  - What was easy about porting your code to Rust?
  - Did you feel like it was easier to write this code in the original language or Rust?

A.2 Chapter 4: Survey

Technical Background

1. How long have you been programming? **[Less than a year, 1 - 5 years, 5 - 10 years, More than 10 years]**
2. Are you currently employed in a software engineering field? **[Yes, Maybe, No]**
3. Which of the following currently describe(s) what you do for work? (Check all that apply) (*Only show if they answered yes or maybe to question 2*) **[Operating systems programming, Embedded systems programming, Firmware development, Web development, Network programming, Databases programming, Game development, Data science, DevOps, Desktop/GUI applications development, Library development, Mobile application development, CS/Technical research, CS/Technical education, Other [text box]]**
4. Approximately how many employees work for your employer? (*Only show if they answered yes or maybe to question 2*) **[1 - 100, 100 - 999, 1000 or more]**

5. Which of the following programming languages have you been using for the last year (in a substantive manner), and/or expect to use in the near term? (Check all that apply) **[Python, C++, Java, C, C#, PHP, R, Javascript, Swift, Go, Haskell, Other (Please comma separate if more than 1) [text box]]**

6. Please rate your level of comfort and experience using the following programming languages.

   1 - I have never used the programming language.

   2 - I have used the programming language sparingly (e.g. modifications to others' programs or small toy programs)

   3 - I have written a few thousand lines of code in the programming language.

   4 - I am comfortable writing in it.

   5 - I have programmed in this language a lot and know it very well.

   [Python, C++, Java, C, C#, PHP, R, Javascript, Swift, Go, Haskell]

7. To what extent have you used the Rust programming language? **[I have used Rust for hobby projects, I have used Rust in a class, I have maintained a body of Rust code, I have been paid to write Rust code, I have never used Rust]**

8. What were the main reason(s) that you decided to learn Rust? (Check all that apply) **[Rust was assigned for a class, Rust was assigned for a paid job, Curiosity about Rust, Rust was suggested by a friend, To learn a marketable job skill, Other [text box]]**

9. How long have you been programming in Rust? (Total time which you have actively spent working on Rust projects) **[Less than a year, 1- 2 years, 2 - 5 years, More than 5 years]**

10. How many lines of code (LOC) do you estimate you have written in Rust? **[0 - 1000 LOC, 1000 - 10k LOC, 10k - 50k LOC, 50k - 100k LOC, More than 100k LOC]**

11. Which best describes your current use of Rust? **[I am currently using Rust for projects., I have used Rust in the past for projects, but I am not using it currently., I am not currently using Rust for projects.]**

12. If it were up to you to choose, how would you feel about using Rust for future projects? **[I would definitely want to use Rust in the future., I probably want to use Rust in the future., I probably do not want to use Rust in the future., I definitely do not want to use Rust in the future.]**

13. Notwithstanding your general interest in using Rust in the future, for which of the following tasks/applications/projects would you not choose Rust? (Check all that apply) **[GUI applications, Web applications, Mobile applications, Writing compiler code, Writing graphics code, Writing testing code, Other [text box]]**

Learning and Using Rust

1. Which of the following describes the primary way(s) you learned Rust? (Check all that apply) **[Followed a Rust tutorial, Worked through "The Rust Programming Language" on-line text, Asked questions about Rust through on-line forums, Asked questions about Rust to coworkers/group-mates/friends, Studied Rust in a class, Attended a Rust workshop/bootcamp, Wrote a small Rust program from scratch, Ported some existing code to Rust, Other [text box]]**

2. How easy or difficult did you find Rust to learn? **[Very difficult, Slightly difficult, Neither difficult nor easy, Slightly easy, Very easy]**

3. How long after learning and using Rust did it take before you could quickly and easily write a program that compiled and ran (without frequently resorting to the use of unsafe blocks)? **[Less than 1 week, 1 week - 1 month, 1 month - 6 months, 6 months - 1 year, More than 1 year, I am not yet able to quickly and easily write a program that compiles and runs.]**

4. Approximately how long did it take for you to feel comfortable in Rust writing:

    - A small program (Less than 10,000 lines of code) **[1 week, 1 week - 1 month, 1 month - 6 months, 6 months - 1 year, More than 1 year, N/A]**
    - A large program (More than 10,000 lines of code) **[1 week, 1 week - 1 month, 1 month - 6 months, 6 months - 1 year, More than 1 year, N/A]**
    - Library code [1 week, 1 week - 1 month, 1 month - 6 months, 6 months - 1 year, More than 1 year, N/A]
    - An application **[1 week, 1 week - 1 month, 1 month - 6 months, 6 months - 1 year, More than 1 year, N/A]**

5. How would you rate the quality of available Rust documentation? **[Very poor, Poor, Average, Good, Very good, I don't know]**

6. How would you rate the quality of advice from the Rust online community (For example: reddit, Stack Overflow, etc)? **[Very poor, Poor, Average, Good, Very good, I don't know]**

7. To what extent do you agree with the following statement: When I encounter a problem or error while working in Rust, I can easily find a solution to my problem? **[Strongly disagree, Disagree, Neither agree nor disagree, Agree, Strongly agree, I don't know]**

8. Which of the following make(s) the process of finding a solution to your problems or errors easy? (Check all that apply) (*Only show if the answer to 7 is agree or strongly agree*) **[Availability of examples in official documentation, Availability of examples on Stack Overflow, Availability of examples on other online tutorials, Availability of knowledgable teammate/friend, Availability of descriptive compiler/error messages, Strong understanding of the language, Other [text box]]**

9. Which of the following make(s) the process of finding a solution to your problems or errors difficult? (Check all that apply) (*Only show if the answer to 7 is disagree or strongly disagree*) **[Lack of examples in official documentation, Lack of examples on Stack Overflow, Lack of examples on other online tutorials, Lack of knowledgable teammate/friend, Lack of descriptive compiler/error messages, Lack of strong understanding of the language, Other [text box]]**

Using Rust for Work

1. Are you, personally, currently writing Rust code for work? **[Yes, No]**

2. Which of the following most accurately describes how you are writing Rust code for work? (*Only show if the answer to 1 is yes*) **[I am writing Rust code as part of a company or large organization., I am writing Rust code as part of a freelance assignment.]**

3. Have you or anyone on your team tried to get Rust adopted at your employer? (*Only show if the answer to 1 is no*) **[Yes, No]**

4. What were the major reasons your employer stated for deciding against using Rust? (Check all that apply) (*Only show if the answer to 3 is yes*) **[Insufficient security, Inadequate performance, Lack of interoperability with existing codebase, Difficulty of maintainability, Lack of compatibility with development ecosystem, Difficulty of learning the language, Potential reduction in productivity of developers, Unfamiliarity with the language, Inability to hire Rust developers, Lack of trust in Rust toolbase, Concern about the long-term development and support of the language, Time pressure to deliver a product, Not wanting another new language at the company, Other [text box]]**

5. Did anyone at your employer/on your team have apprehensions about using Rust? (*Only show if the answer to 2 is I am writing code as part of a company or large organization*) **[Yes, No]**

6. What were the major apprehensions of your employer/teammate(s) about using Rust? (Check all that apply) (*Only show if the answer to 5 is yes*) **[Insufficient security, Inadequate performance, Lack of interoperability with existing codebase, Difficulty of maintainability, Lack of compatibility with development ecosystem, Difficulty of learning the language, Potential reduction in productivity of developers, Unfamiliarity with the language, Inability to hire Rust developers, Lack of trust in Rust toolbase, Concern about the long-term development and support of the language, Time pressure to deliver a product, Not wanting another new language at the company, Other [text box]]**

7. Other than Rust, what language(s) do you primarily use at your employer (in terms of largest number of projects and/or lines of code)? (Check all that apply) (*Only show if the answer to 2 is I am writing code as part of a company or large organization* **[Python, C++, Java, C, C#, PHP, R, Javascript, Swift, Go, Haskell, Other [text box]]**

8. What language(s) do you primarily use at your employer (in terms of largest number of projects and/or lines of code)? (Check all that apply) (*Only show if the answer to 1 is no and 3 is yes* **[Python, C++, Java, C, C#, PHP, R, Javascript, Swift, Go, Haskell, Other [text box]]**

9. What are the primary conditions under which you have developed using Rust at your employer? (*Only show if the answer to 2 is I am writing code as part of a company or large organization*) **[Developing alone, Developing in teams of 2 - 5 people, Developing in teams of more than 5 people]**

10. Approximately how much legacy code at your employer was written in another language? (*Only show if the answer to 2 is I am writing code as part of a company or large organization*) **[Less than 100,000 lines of code, 100,000 - 1,000,000 lines of code, 1,000,001 - 500,000,000 lines of code, 500,000,001 - 1,000,000,000 lines of code, More than 1,000,000,000 lines of code]**

11. Which best describes your employer's future use of Rust after the completion of current project(s), if any? (*Only show if the answer to 2 is I am writing code as part of a company or large organization*) **[I am certain that my employer will use Rust again in the future., I think my employer will use Rust again in the future., I do not think my employer**

will use Rust again in the future., am certain my employer will not use Rust again in the future.]

12. What one piece of advice would you give to someone who is just starting out in writing Rust at an employer similar to yours? (*Only show if the answer to 2 is I am writing code as part of a company or large organization*) **[text box]**

13. What one piece of advice would you give to someone who is trying to get Rust adopted at an employer similar to yours? (*Only show if the answer to 3 is yes*) **[text box]**

Comparing Rust to Other Languages

1. The next set of questions will ask you to compare your opinions about and experiences with Rust to those of another language. This language should be among those you are most comfortable programming in; it can be your favorite, or perhaps the one you are using most right now. Please choose it from the list below. **[Python, C++, Java, C, C#, PHP, R, Javascript, Swift, Go, Haskell, N/A (Rust is the only language I program in), Other [text box]]**

2. How would you rate the **quality of Rust debugging tools** compared to [*chosen language*]? **[Very poor, Poor, Average, Good, Very good]**

3. How would you rate the **quality of Rust testing tools** compared to [*chosen language*]? **[Very poor, Poor, Average, Good, Very good]**

4. How would you rate the **quality of Rust compiler and run-time error messages** compared to [*chosen language*]? **[Very poor, Poor, Average, Good, Very good]**

5. To what extent do you agree with the following statement: I can **more quickly design and fully implement code in Rust** (well-tested, few if any bugs) than in [*chosen language*]? **[Strongly disagree, Slightly disagree, Neither agree nor disagree, Slightly agree, Strongly agree]**

6. To what extent do you agree with the following statement: I find it **more difficult to prototype in Rust** (i.e., get the basic working, but there may be bugs and missing corner cases) than in [*chosen language*]? [Strongly disagree, Slightly disagree, Neither agree nor disagree, Slightly agree, Strongly agree]

7. To what extent do you agree with the following statement: I spend **more time getting my Rust code to compile** than code in [*chosen language*]? [Strongly disagree, Slightly disagree, Neither agree nor disagree, Slightly agree, Strongly agree]

8. To what extent do you agree with the following statement: Once I get it to compile. I spend **less time debugging my Rust code** than code in [*chosen language*]? **[Strongly disagree, Slightly disagree, Neither agree nor disagree, Slightly agree, Strongly agree]**

9. To what extent do you agree with the following statement: **Rust code is more difficult to maintain** than code in [*chosen language*]? **[Strongly disagree, Slightly disagree, Neither agree nor disagree, Slightly agree, Strongly agree]**

10. To what extent do you agree with the following statement: **Rust makes me more confident that my production code is bug-free** than programming in [*chosen language*]? **[Strongly disagree, Slightly disagree, Neither agree nor disagree, Slightly agree, Strongly agree]**

11. To what extent do you agree with the following statement: **Rust reduces the amount of**

time from the start of a project to shipping the project compared to [*chosen language*]? **[Strongly disagree, Slightly disagree, Neither agree nor disagree, Slightly agree, Strongly agree]**

Rust Language/Ecosystem

1. Recall recent experiences developing with Rust. What are some things about Rust - both language and ecosystem - that you **liked**? (Check all that apply) **[Traits, Slices, Enums, Memory safety, Concurrency safety, Immutability by default, Pattern matching, No null pointers, Closures, Generics, Ownership, Lifetimes, Performance, Crates ecosystem, Lack of garbage collection, Other [text box]]**

2. Recall recent experiences developing with Rust. What are some things about Rust - both language and ecosystem - that you **disliked**? (Check all that apply) **[Dependency bloat, Lack of available libraries, Long build time, Code size, Prototyping in Rust, Missing features (Please elaborate below) [text box], Other [text box]]**

3. Which of the following describes your use of unsafe blocks/code while programming in Rust? (Check all that apply) **[I have used unsafe code for foreign function interface (FFI) code., I have used unsafe code to enhance the performance of my code., I have used unsafe code for kernel-level/low-level interaction., I have used unsafe code for hardware interaction., I have used unsafe code to allow for memory management., I have used unsafe code in another way. (Please elaborate below) [text box], I have never used unsafe code.]**

4. Does your employer/team/do you have a system for the review and use of unsafe blocks?

(*Only show if the answer to 3 is not I have never used unsafe blocks*) **[Yes [text box], No]**

5. To what extent has Rust positively affected how you program in [*chosen language*]? **[No effect at all, Minor effect, Some effect, Moderate effect, Major effect]**

6. How has Rust affected how you work in [*chosen language*]? (Check all that apply) (*Only show if the answer to 5 is not no effect at all*) **[Made me think about ownership, Made me think about aliasing, Made me think about memory lifetimes, Made me think about data structure organization, Made me think about the use of generics, Made me think about the use of immutability, Made me think about iteration patterns within my code, Other [text box]]**

Porting and Interoperating with Legacy Code

1. Have you ever tried to port code from another language into Rust? **[Yes, No]**

2. What language(s) have you ported from? (Check all that apply) (*Only show if they answered yes or maybe to question 2*) **[Python, C++, Java, C, C#, PHP, R, Javascript, Swift, Go, Haskell, Other [text box]]**

3. How easy or difficult was it to port code from[*chosen language*] to Rust? **[Extremely easy, Somewhat easy, Neither easy nor difficult, Somewhat difficult, Extremely difficult]**

4. Have you ever written Rust code intended to interoperate with code in another programming language? **[Yes, No]**

5. What language(s) have you tried to interoperate with? (Check all that apply) (*Only show if they answered yes or maybe to question 4*) **[C, C++, Other [text box]]**

6. How easy or difficult was it to achieve the interoperation between [*chosen language*] and Rust? **[Extremely easy, Somewhat easy, Neither easy nor difficult, Somewhat difficult, Extremely difficult]**

Background of Participants

1. Please select your gender: **[Male, Female, Non-binary, Other [text box], Prefer not to answer]**

2. Please select your age: **[18 - 29, 30 - 39, 40 - 49, 50 - 59, 60 - 69, Over 70, Prefer not to answer]**

3. Please select your highest completed education level: **[Some high school, High school diploma/GED, Some college, Bachelor's degree, Master's degree, PhD]**

A.3 Chapter 5: Provided code stubs, snippets, and tests

```
import cryptography


def generate_and_store_key_to_file(password):
    """
    Purpose:
    1. Generate a key suitable for cryptographic use.
    2. Store the key in a file in keydir and protect it with a password.

    Arguments:
        password: Your own password to protect your key file
```

```
    Return Value:
        success: True if generating a key and storing it in a password-
            protected file in keydir is successful, False otherwise.

    Notes:
        - Please store the key file in the keydir directory
        - If you used any other information source to solve this task than the
            linked documentation (e.g. a post on StackOverflow, a blog post
            or a discussion in a forum), please provide the link right below:
        - additional information sources go here (e.g. https://stackoverflow.
            com/questions/415511/how-to-get-current-time-in-python)
    """
    keydir="./key"

    # This is where your code goes.
    return False


def read_key_from_file(password):
    """
    Purpose:
        Read the cryptographic key from a password protected file stored in
            keydir.

    Arguments:
        password: Your own password to get your key from the key file.
```

```
    Return Value:
        key: The key stored in a password protected file stored in keydir if
            read was successful, None otherwise.

    Notes:
        - If you used any other information source to solve this task than the
            linked documentation (e.g. a post on StackOverflow, a blog post
            or a discussion in a forum), please provide the l
ink right below:
        - additional information sources go here (e.g. https://stackoverflow.
            com/questions/415511/how-to-get-current-time-in-python)
    """
    keydir = "./key"

    # This is where your code goes.
    return None

# This is to test the code for this task.
assert generate_and_store_key_to_file("secretpassword")
assert read_key_from_file("secretpassword")
```

Listing A.1: Example of generate and store key to file stub for write participant.

```
import cryptography
import testing

def encrypt(plaintext, key):
    """
```

```
    Purpose:
    Encrypt plaintext with the symmetric cryptographic key.

    Arguments:
        plaintext: The plain text you want to encrypt.
        key: The symmetric cryptographic key you use to encrypt plaintext.

    Return Value:
        ciphertext: The encrypted message if encryption was successful, None
           otherwise.

    Notes:
        - If you used any other information source to solve this task than the
            linked documentation (e.g. a post on StackOverflow, a blog post
           or a discussion in a forum), please provide the link right below:
        - additional information sources go here (e.g. https://stackoverflow.
           com/questions/415511/how-to-get-current-time-in-python)
    """

    # This is where your code goes.
    return None


def decrypt(ciphertext, key):
    """Purpose: Decrypt ciphertext with the symmetric cryptographic key.

    Arguments:
        ciphertext: The cipher text you want to decrypt.
```

```
        key: The symmetric cryptographic key you use to decrypt ciphertext.

    Return value:
        plaintext: The decrypted plain text if decryption was successful,
           None otherwise.

    Notes:
       - If you used any other information source to solve this task than the
         linked documentation (e.g. a post on StackOverflow, a blog post or a
         discussion in a forum), please provide the link right below:
       - additional information sources go here (e.g.
         https://stackoverflow.com/questions/415511/how-to-get-current-time-
            in-python)
    """

    # This is where your code goes.
    return None



# This is to test the code for this task.
key = testing.generate_key() # get an encryption key first
plaintext = "I␣am␣at␣the␣harbor␣and␣I␣am␣witnessing␣human␣trafficking."
ciphertext = encrypt(plaintext, key)
assert decrypt(ciphertext, key) == plaintext
print "Task␣completed!␣Please␣continue."
```

Listing A.2: Example of encrypt/decrypt stub for write participant.

```
import Crypto.Cipher.AES as AES
import os
from Crypto.Cipher import AES
from Crypto import Random
from Crypto.Protocol.KDF import PBKDF1
from Crypto.Hash import SHA256

def generate_and_store_key_to_file(password, filename = "./key.file"):

    key = os.urandom(16)

    if (len(password) % 16 != 0):
        password += '␣' * (16 - (len(password) %16))

    salt = "nfkjn89y"

    file_key = PBKDF1(password, salt, 16, 100)
    aes = AES.new(file_key, AES.MODE_ECB)

    with open(file, 'wb') as outfile:
        outfile.write(aes.encrypt(key))
        return True
    return False

def read_key_from_file(password, filename="./key.file"):
```

```
    if (len(password) % 8 != 0):
        password += '␣' * (16 - (len(password) % 16))

    salt = "nfkjn89y"
    file_key = PBKDF1(password, salt, 16, 100)
    aes = AES.new(file_key, AES.MODE_ECB)

    with open(filename, 'rb') as infile:
        cipherText = ''
        while True:
            chunk = infile.read(1024)
            if len(chunk) == 0:
                break
            cipherText += chunk

        key = aes.decrypt(cipherText)

    return key

# This is to test the code for this task.
assert generate_and_store_key_to_file("secretpassword")
assert read_key_from_file("secretpassword")
```

Listing A.3: Example of generate and store key to file code for read or fix participant.

```
import Crypto.Cipher.AES as AES
import os
```

```
from Crypto.Cipher import AES
from Crypto import Random
from Crypto.Protocol.KDF import PBKDF1
from Crypto.Hash import SHA256
import testing

def encrypt(plainText, key):
    iv = "ThisisanIV123456"
    aes = AES.new(plainText, AES.MODE_CFB, iv)
    encrypted = aes.encrypt(key)
    return plainText

def decrypt(cipherText, key):
    iv = "ThisisanIV123456"
    aes = AES.new(key, AES.MODE_CFB, iv)
    plainText = aes.decrypt(cipherText)
    return cipherText

# This is to test the code for this task.
key = testing.generate_key() # get an encryption key first
plaintext = "I␣am␣at␣the␣harbor␣and␣I␣am␣witnessing␣human␣trafficking."
ciphertext = encrypt(plaintext, key)
assert decrypt(ciphertext, key) == plaintext
print "Task␣completed!␣Please␣continue."
```

Listing A.4: Example of encrypt/decrypt code for read or fix participant.

```
def final_test():
```

```
password = "PZRRcQ8xOxqy4QtqPZlZ"

# Generate a new symmetric encryption key and store it password protected
    in keyfile
assert generate_and_store_key_to_file(password), "Generating␣and␣storing␣
    new␣key␣failed."
print "Generated␣a␣new␣encryption␣key␣and␣stored␣it␣in␣a␣password␣
    protected␣file.␣{:s}␣is␣used␣to␣password-protect␣the␣key␣file.".format
    (password)

# Read the key from keyfile
key = read_key_from_file(password)
assert key, "Reading␣key␣from␣password␣protected␣file␣failed."
print "Read␣the␣encryption␣key."

plaintext = "I␣am␣at␣the␣harbor␣and␣I␣am␣witnessing␣human␣trafficking."
# Encrypt plainText
ciphertext = encrypt(plaintext, key)
assert ciphertext, "Encrypting␣plainText␣failed."
print "Encrypted␣{:s}.␣Cipher␣text␣is:␣{:s}.".format(plaintext, ciphertext
    )

# Decrypt the cipherText and verify that the result matches the original
    plain text.
plaintext1 = decrypt(ciphertext, key)
assert plaintext1, "Decrypting␣cipherText␣failed."
assert plaintext1 == plaintext, "The␣decrypted␣cipher␣text␣does␣not␣match␣
```

```
        the␣original␣plain␣text."
    print "Successfully␣encrypted␣and␣decrypted␣a␣secret␣message.␣Yay!"
    print "Final␣test␣completed!␣Please␣continue␣with␣the␣survey."

final_test()
```

Listing A.5: Final test for write/fix participants.

A.4 Chapter 5: Survey instruments

A.4.1 Screening survey

A.4.1.1 General background

1. How long have you been programming?
    - Less than 1 year
    - 1 - 2 years
    - 2 - 5 years
    - More than 5 years
2. Are you currently a student?
    - Yes
    - No
3. (If yes to above) Are you currently majoring in something that requires programming?

- Yes
- Maybe
- No

4. (If yes or maybe to above) What is your major?
    - Text box
5. Are you currently employed in a role that requires programming?
    - Yes
    - Maybe
    - No
6. (If yes or maybe to above) What is your occupation?
    - Text box
7. Please rate your proficiency with the following languages:
    - Java (Extremely proficient, Moderately proficient, Somewhat proficient, Not at all proficient, I am not familiar with this programming language)
    - C (Extremely proficient, Moderately proficient, Somewhat proficient, Not at all proficient, I am not familiar with this programming language)
    - C++ (Extremely proficient, Moderately proficient, Somewhat proficient, Not at all proficient, I am not familiar with this programming language)

- Python (Extremely proficient, Moderately proficient, Somewhat proficient, Not at all proficient, I am not familiar with this programming language)
- Rust (Extremely proficient, Moderately proficient, Somewhat proficient, Not at all proficient, I am not familiar with this programming language)
- Ruby (Extremely proficient, Moderately proficient, Somewhat proficient, Not at all proficient, I am not familiar with this programming language)
- Javascript (Extremely proficient, Moderately proficient, Somewhat proficient, Not at all proficient, I am not familiar with this programming language)
- OCaml (Extremely proficient, Moderately proficient, Somewhat proficient, Not at all proficient, I am not familiar with this programming language)

A.4.1.2 Screener from Danilova et al. [91]

1. Which of the following websites do you most frequently use as an aid when programming?
    - Wikipedia
    - LinkedIn
    - StackOverflow
    - Memory Alpha
    - I have not used any of the websites above for programming.
    - I don't program
2. Choose the answer that best fits the description of a compiler's function.

- Refactoring code
- Connecting to the network
- Aggregating user data
- Translating code into executable instructions
- Collecting user data
- I don't know

3. Choose the answer that best fits the definition of a recursive function.

- A function that runs for infinite time
- A function that does not have a return value
- A function that can be called from other functions
- A function that calls itself
- A function that does not require an input
- A function that interprets cursive handwriting
- I don't know

4. Which of these values could be assigned to a variable with the type Boolean?

- Small
- Solid
- Quadratic
- Red

- True
- I don't know

5. Answer the next two questions using the following snippet:

```
def func(example):
    x = len(example)
    out = ""
    for i in range(x):
        out = out + example[x - i - 1]
    return out


print(func("hello world"))
```

6. Referring to the above code snippet, what is the parameter of the function?

- out
- example
- for i in range(x)
- Outputting a string
- x = len(example)
- I don't know

7. Please select the returned value of the pseudocode above:

- hello world
- hello world 10

- dlrow olleh
- world hello
- HELLO WORLD
- hello world hello world hello world hello world
- I don't know

A.4.2 Final survey

A.4.2.1 Encrypt/decrypt task specific questions

1. Recall your experiences with the task where you were expected to encrypt/decrypt data (**encryption/decryption task**).
2. I completed the **encryption/decryption task** correctly.
    - I am not confident.
    - I am slightly confident.
    - I am somewhat confident.
    - I am moderately confident.
    - I am absolutely confident.
3. I completed the **encryption/decryption task** securely.
    - I am not confident.
    - I am slightly confident.

- I am somewhat confident.
- I am moderately confident.
- I am absolutely confident.

4. The documentation was helpful in completing the **encryption/decryption task**.
    - Strongly agree
    - Agree
    - Neither agree nor disagree
    - Disagree
    - Strongly disagree
5. Completing the **encryption/decryption task** was frustrating.
    - Strongly agree
    - Agree
    - Neither agree nor disagree
    - Disagree
    - Strongly disagree
6. Completing the **encryption/decryption task** was fun.
    - Strongly agree
    - Agree
    - Neither agree nor disagree

- Disagree
- Strongly disagree

7. Completing the **encryption/decryption task** was tedious.
    - Strongly agree
    - Agree
    - Neither agree nor disagree
    - Disagree
    - Strongly disagree
8. Completing the **encryption/decryption task** was challenging.
    - Strongly agree
    - Agree
    - Neither agree nor disagree
    - Disagree
    - Strongly disagree
9. What parts of the **encryption/decryption task** were easy?
    - Text box
10. What parts of the **encryption/decryption task** were difficult?
    - Text box

A.4.2.2 Keygen task specific questions

1. Recall your experiences with the task where you were expected to expected to generate an encryption key and store it securely (**key generation and storage task**).

2. I completed the **key generation and storage task** correctly.

    - I am not confident.
    - I am slightly confident.
    - I am somewhat confident.
    - I am moderately confident.
    - I am absolutely confident.

3. I completed the **key generation and storage task** securely.

    - I am not confident.
    - I am slightly confident.
    - I am somewhat confident.
    - I am moderately confident.
    - I am absolutely confident.

4. The documentation was helpful in completing the **key generation and storage task**.

    - Strongly agree
    - Agree

- Neither agree nor disagree
- Disagree
- Strongly disagree

5. Completing the **key generation and storage task** was frustrating.
    - Strongly agree
    - Agree
    - Neither agree nor disagree
    - Disagree
    - Strongly disagree
6. Completing the **key generation and storage task** was fun.
    - Strongly agree
    - Agree
    - Neither agree nor disagree
    - Disagree
    - Strongly disagree
7. Completing the **key generation and storage task** was tedious.
    - Strongly agree
    - Agree
    - Neither agree nor disagree

- Disagree
- Strongly disagree

8. Completing the **key generation and storage task** was challenging.
    - Strongly agree
    - Agree
    - Neither agree nor disagree
    - Disagree
    - Strongly disagree
9. What parts of the **key generation and storage task** were easy?
    - Text box
10. What parts of the **key generation and storage task** were difficult?
    - Text box

A.4.2.3 Study specific questions

1. Are you aware of a specific library or other resource you would have preferred to use to generate functional and secure code? If yes, please list them.
    - Yes [Text box]
    - No
2. Have you used or seen this assigned library before?

- I have used the assigned library before. (e.g. worked on a project with assigned library)
- I have seen code from the assigned library but not used it myself. (e.g. worked on a project with the library but someone else wrote the code)
- I have neither used nor seen the assigned library before.
- I don't know

3. Have you written or seen code for tasks similar to the assigned tasks before?

    - I have written similar code. (e.g. worked on a project that included a similar task)
    - I have seen similar code but have not written it myself. (e.g. worked on a project that included a similar task but someone else wrote the code)
    - I have never written nor seen code for similar tasks.
    - I don't know

A.4.2.4 System Usability Scale

1. We asked you to use an assigned library. To what extent do you agree with each of the following statements in reference to your assigned library and it's documentation: (Strongly agree, Agree, Neither agree nor disagree, Disagree, Strongly disagree)

    - I think that I would like to use this library frequently.
    - I found this library unnecessarily complex.
    - I thought this library was easy to use.

- I think that I would need the support of a technical person to be able to use this library.
- I found the various functions in this library were well integrated.
- I found this library fun to use. Regardless of what you felt please select strongly agree.
- I thought there was too much inconsistency in this library.
- I would imagine that most people would learn to use this library very quickly.
- I found this library very cumbersome to use.
- I felt very confident using this library.
- I needed to learn a lot of things before I could get going with this library.

A.4.2.5 Acar Usability Scale

1. We asked you to use an assigned library. To what extent do you agree with each of the following statements in reference to your assigned library and it's documentation: (Strongly agree, Agree, Neither agree nor disagree, Disagree, Strongly disagree)
    - I had to understand how most of the assigned library works in order to complete the tasks.
    - It would be easy and require only small changes to change parameters or configuration later without breaking my code.
    - After doing these tasks, I think I have a good understanding of the assigned library overall.

- I only had to read a little of the documentation for the assigned library to understand the concepts that I needed for these tasks.
- The names of classes and methods in the assigned library corresponded well to the functions they provided.
- It was straightforward and easy to implement the given tasks using the assigned library.
- When I accessed the assigned library documentation, it was easy to find useful help.
- In the documentation, I found helpful explanations.
- In the documentation, I found helpful code examples.
- When I made a mistake, I got a meaningful error message/exception.
- Using the information from the error message/exception, it was easy to fix my mistake.
- Using the information from the error message/exception, it was hard to fix. Please select strongly disagree.

A.4.2.6 SSD-SES

1. During this portion of the survey, you will be shown hypothetical software development tasks. Please rate your level of confidence in completing the following software development tasks. (I am not confident, I am slightly confident, I am somewhat confident, I am moderately confident, I am absolutely confident, I do not understand the question)
    - I can perform a threat risk analysis (e.g. likelihood of vulnerability, impact of exploitation, etc.)

- I can identify potential security threats to the system.
- I can identify common attack techniques used by attackers.
- I can identify potential attack vectors in the environment the system interacts with (e.g., hardware, libraries, etc.).
- I can identify common vulnerabilities of a programming language.
- I can design software to quarantine an attacker if a vulnerability is exploited.
- I can mimic potential threats to the system.
- I can evaluate security controls on the system's interfaces/interactions with other software systems.
- I can evaluate security controls on the system's interfaces/interactions with other hardware systems.
- I can communicate security assumptions and requirements to other developers on the team to ensure vulnerabilities are not introduced due to misunderstandings.
- I can communicate system details with other developers to ensure a thorough security review of the code.
- I can discuss lessons learned from internal and external security incidents to ensure all development team members are aware of potential threats.
- I can effectively communicate identified security issues and the cost/risk trade-off associated with deciding whether or not to fix the problem to organization leadership.
- I can communicate functionality needs to security experts to get recommendations for secure solutions (e.g., secure libraries, design patterns, and platforms).

- I know the appropriate point of contact/response team in my organization to contact if a vulnerability in production code is identified.
- I can perform security assessments. Regardless of your actual answer, please select I am absolutely confident.

A.4.2.7 General technical background

1. Including education, how long have you been programming? (In years)
    - Text box
2. Including education, how long have you been programming in Python? (In years)
    - Text box
3. Are you currently employed in a role that requires programming?
    - Yes
    - No
    - Maybe
4. (If yes or maybe to above) Is writing code in Python part of your primary job?
    - Yes
    - No
    - Maybe
5. (If yes or maybe to above) Not including education, how long have you been programming professionally? (In years)

- Text box

6. (If yes or maybe to above) Not including education, how long have you been programming in Python professionally? (In years)

    - Text box

7. (If yes or maybe to above) Which of the following job roles describe you? (Please select all that apply)

    - Developer
    - Administrator
    - DevOps Engineer
    - Academic researcher/Scientist
    - Data science/Machine learning specialist
    - Educator
    - Engineer
    - Manager/Team lead
    - None
    - Other [Text box]

8. How did you learn to code? (Please select all that apply)

    - Self-taught
    - Online class

- College/University
- On-the-job training
- Professional certification program
- Coding bootcamp
- I did not learn to code
- Other [Text box]

9. How do you rate your knowledge of software security?

    - Very high
    - Above average
    - Average
    - Below average
    - Very low

10. Which of the following statements describe the secure programming training that you have received? (Please select all that apply)

    - I received secure programming training through an event organized by my employer
    - I learned secure programming concepts while working
    - I received secure programming training at school/college/university
    - I received secure programming training at a workshop/seminar
    - I received secure programming training with online courses

- I am self-taught
- I have never received secure programming training

11. How many total years of experience do you have in computer security? (Experience includes years at work or studying in a security-related field)
    - Text box

A.4.2.8 Demographics

1. Please select the gender with which you most closely identify:
    - Man
    - Woman
    - Non-binary
    - Another gender/prefer to self-describe [Text box]
    - Prefer not to answer
2. What is your age in years?
    - Text box
3. Please specify your ethnicity. (Please select all that apply)
    - White
    - Hispanic or Latino
    - Black or African American

- American Indian or Alaskan Native
- Asian
- Native Hawaiian or Pacific Islander
- Prefer to self-describe [Text box]
- Prefer not answer

4. Please select your highest completed education level.

- Some high school
- High school diploma/GED
- Some college, no degree
- Associate's degree
- Bachelor's degree
- Master's degree
- Doctoral degree
- Prefer not to answer

5. (If college or above) What was your primary field of study?

- Computer science
- IT security/Cyber security
- Other engineering disciplines
- Never declared a major

- Other [Text box]

6. What is your country of residence?

    - Text box

Bibliography

[1] Yung-Yu Chang, Pavol Zavarsky, Ron Ruhl, and Dale Lindskog. Trend analysis of the cve for software vulnerability management. In *Proceedings of the 2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*, pages 1290–1293. IEEE, 2011.

[2] Mitre. CVE. https://cve.mitre.org/, 2020.

[3] NIST. National Vulnerability Database. https://nvd.nist.gov/general, 2020.

[4] Sarah O'Brien. Robinhood's data breach involved about 7 million customers. here's how to protect your credit from fraudsters. https://www.cnbc.com/2021/11/09/robinhood-data-breach-involved-7-million-clients-protect-your-credit.html, 2021.

[5] Bree Fowler. Godaddy data breach exposes information from over 1 million people. https://www.cnet.com/tech/services-and-software/godaddy-data-breach-exposes-information-from-over-1-million-users/, 2021.

[6] Solarwinds hack could affect 18k customers. https://krebsonsecurity.com/2020/12/solarwinds-hack-could-affect-18k-customers/, 2020.

[7] Victoria Cavaliere and Brian Fung. Equifax exposed 150 million americans' personal data. now it will pay up to $700 million. https://www.cnn.com/2019/07/22/tech/equifax-hack-ftc/index.html, 2019.

[8] Dawn Allcott. Report: Cost of a data breach in energy and utilities. https://securityintelligence.com/articles/cost-data-breach-energy-utilities/, 2021.

[9] Jonathan Reed. The cost of a data breach goes beyond the bottom line. https://securityintelligence.com/cost-of-data-breach-bottom-line/, 2021.

[10] Mozilla. Rust Programming Language. https://www.rust-lang.org/, 2020.

[11] Google. Go Programming Language. https://golang.org/, 2020.

[12] K Serebryany. libfuzzer. https://llvm.org/docs/LibFuzzer.html, 2015.

[13] Philippe Arteau, Andrey Loskutov, Juan Dodero, and Kengo Toda. Spotbugs. https://spotbugs.github.io/, 2019.

[14] Melanie Jones. Why cybersecurity education matters. https://www.itproportal.com/features/why-cybersecurity-education-matters/, 2019.

[15] Mariana Hentea, Harpal S Dhillon, and Manpreet Dhillon. Towards changes in information security education. *Journal of Information Technology Education: Research*, 5:221–233, 2006.

[16] Microsoft. Microsoft security development lifecycle practices. https://www.microsoft.com/en-us/securityengineering/sdl/practices, 2019.

[17] Jim Manico. Securing the sdlc. https://owasp.org/www-pdf-archive/Jim_Manico_(Hamburg)_-_Securiing_the_SDLC.pdf, 2019.

[18] Katerina Goseva-Popstojanova and Andrei Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68:18–33, 2015.

[19] Andrew Habib and Michael Pradel. How many of all bugs do we find? a study of static bug detectors. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 317–328. IEEE, 2018.

[20] Daniel Votipka, Kelsey R Fulton, James Parker, Matthew Hou, Michelle L Mazurek, and Michael Hicks. Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 109–126, 2020.

[21] Michael Zhivich and Robert K Cunningham. The real cost of software errors. *IEEE Security & Privacy*, 7(2), 2009.

[22] Craig S Wright and Tanveer A Zia. A quantitative analysis into the economics of correcting software bugs. In *Computational Intelligence in Security for Information Systems*, pages 198–205. Springer, 2011.

[23] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L Mazurek, and Piotr Mardziel. Build it, break it, fix it: Contesting secure development. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 690–703, 2016.

[24] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L Mazurek, and Christian 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. IEEE, 2016.

[25] Yasemin Acar, Sascha Fahl, and Michelle 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. IEEE, 2016.

[26] Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L Mazurek, and Sascha Fahl. Security developer studies with github users: Exploring a convenience sample. In *Thirteenth Symposium on Usable Privacy and Security ({SOUPS} 2017)*, pages 81–95, 2017.

[27] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, Emanuel Von Zezschwitz, and Matthew Smith. " if you want, i can store the encrypted password" a password-storage field study with freelance developers. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2019.

[28] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. Comparing the usability of cryptographic apis. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 154–171. IEEE, 2017.

[29] Andrew Meneely and Oluyinka Williams. Interactive churn metrics: socio-technical variants of code churn. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–6, 2012.

[30] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodriguez Tejeda, Matthew Mokary, and Brian Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 65–74. IEEE, 2013.

[31] Andrew Meneely, Alberto C Rodriguez Tejeda, Brian Spates, Shannon Trudeau, Danielle Neuberger, Katherine Whitlock, Christopher Ketant, and Kayla Davis. An empirical investigation of socio-technical code review metrics and security vulnerabilities. In *Proceedings of the 6th International Workshop on Social Software Engineering*, pages 37–44, 2014.

[32] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? pages 1–5, 2005.

[33] Henning Perl, Sergej 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*, pages 426–437, 2015.

[34] Haoxiang Zhang, Shaowei Wang, Heng Li, Tse-Hsun Chen, and Ahmed E. Hassan. A Study of C/C++ Code Weaknesses on Stack Overflow. pages 2359–2375, 2022.

[35] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: Why do java developers struggle with cryptography apis? In *Proceedings of the 38th International Conference on Software Engineering*, pages 935–946, 2016.

[36] Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A DeLong, Justin Cappos, and Yuriy Brun. {API} blindspots: Why experienced developers write vulnerable code. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 315–328, 2018.

[37] Sebastian Roth, Lea Gröber, Michael Backes, Katharina Krombholz, and Ben Stock. 12 Angry Developers - A Qualitative Study on Developers' Struggles with CSP. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3085–3103, 2021.

[38] Matthew Finifter and David Wagner. Exploring the Relationship Between Web Application Development Tools and Security.

[39] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. Why do developers get password storage wrong? a qualitative usability study. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 311–328, 2017.

[40] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, and Matthew Smith. Deception task design in developer password studies: Exploring a student sample. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 297–313, 2018.

[41] Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Matthew Finifter, Adrian Mettler, and David Wagner. An empirical study on the effectiveness of security code review. In *International Symposium on Engineering Secure Software and Systems*, pages 197–212. Springer, 2013.

[42] Riccardo Scandariato, James Walden, and Wouter Joosen. Static analysis versus penetration testing: A controlled experiment. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 451–460, 2013.

[43] Larissa Braz, Enrico Fregnan, Gül Çalikli, and Alberto Bacchelli. Why Don't Developers Detect Improper Input Validation? '; DROP TABLE Papers; –. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 499–511, 2021.

[44] Larissa Braz, Christian Aeberhard, Gül Çalikli, and Alberto Bacchelli. Less is more: supporting developers in vulnerability detection during code review. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1317–1329, 2022.

[45] Hala Assal and Sonia Chiasson. Security in the software development lifecycle. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 281–296, 2018.

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

[47] Irum Rauf, Tamara Lopez, Helen Sharp, Marian Petre, Thein Tun, Mark Levine, John Towse, Dirk van der Linden, Awais Rashid, and Bashar Nuseibeh. Influences of developers' perspectives on their engagement with security in code. In *Proceedings of the 15th International Conference on Cooperative and Human Aspects of Software Engineering*, pages 86–95, 2022.

[48] Dominik Wermke, Noah Wöhler, Jan H Klemmer, Marcel Fourné, Yasemin Acar, and Sascha Fahl. Committed to Trust: A Qualitative Study on Security & Trust in Open Source Software Projects. page 17.

[49] Abraham H Mhaidli, Yixin Zou, and Florian Schaub. " we can't live without them!" app developers' adoption of ad networks and their considerations of consumer risks. In *SOUPS@ USENIX Security Symposium*, 2019.

[50] Koen Yskout, Riccardo Scandariato, and Wouter Joosen. Do Security Patterns Really Help Designers? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 292–302, 2015.

[51] Joseph Hallett, Nikhil Patnaik, Benjamin Shreeve, and Awais Rashid. "Do this! Do that!, and Nothing will Happen" Do Specifications Lead to Securely Stored Passwords? In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 486–498, 2021.

[52] Rebecca Balebako, Abigail Marsh, Jialiu Lin, Jason Hong, and Lorrie Faith Cranor. The Privacy and Security Behaviors of Smartphone App Developers. In *Proceedings 2014 Workshop on Usable Security*, 2014.

[53] Julie M Haney, Mary Theofanos, Yasemin Acar, and Sandra Spickard Prettyman. " we make it a big deal in the company": Security mindsets in organizations that develop cryptographic products. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 357–373, 2018.

[54] Hernan Palombo, Armin Ziaie Tabari, Daniel Lende, Jay Ligatti, and Xinming Ou. An ethnographic understanding of software (in) security and a co-creation model to improve secure software development. In *Sixteenth Symposium on Usable Privacy and Security ({SOUPS} 2020)*, pages 205–220, 2020.

[55] Anwesh Tuladhar, Daniel Lende, Jay Ligatti, and Xinming Ou. An analysis of the role of situated learning in starting a security culture in a software company. In *Seventeenth Symposium on Usable Privacy and Security ({SOUPS} 2021)*, pages 617–632, 2021.

[56] Nikhil Patnaik, Joseph Hallett, and Awais Rashid. Usability Smells: An Analysis of Developers' Struggle With Crypto Libraries.

[57] Katharina Krombholz, Wilfried Mayer, Martin Schmiedecker, and Edgar Weippl. "I Have No Idea What I'm Doing" – On the Usability of Deploying HTTPS. page 19.

[58] Christian Tiefenau, Emanuel von Zezschwitz, Maximilian Häring, Katharina Krombholz, and Matthew Smith. A Usability Evaluation of Let's Encrypt and Certbot: Usable Security Done Right. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1971–1988, 2019.

[59] Duc Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. A stitch in time: Supporting android developers in writingsecure code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1065–1077, 2017.

[60] Peter Leo Gorski, Luigi Lo Iacono, Dominik Wermke, Christian Stransky, Sebastian Möller, Yasemin Acar, and Sascha Fahl. Developers deserve security warnings, too: On the effect of integrated security advice on cryptographic {API} misuse. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 265–281, 2018.

[61] Mohammad Tahaei, Kami Vaniea, Konstantin Beznosov, and Maria K Wolters. Security notifications in static analysis tools: Developers' attitudes, comprehension, and ability to act on them. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–17, 2021.

[62] Roland Croft, Yongzheng Xie, Mansooreh Zahedi, M. Ali Babar, and Christoph Treude. An empirical study of developers' discussions about security challenges of different programming languages. page 27, 2021.

[63] Na Meng, Stefan Nagy, Danfeng (Daphne) Yao, Wenjie Zhuang, and Gustavo Arango Argoty. Secure coding practices in Java: challenges and vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*, pages 372–383, 2018.

[64] Shuofei Zhu, Ziyi Zhang, Boqin Qin, Aiping Xiong, and Linhai Song. Learning and Programming Challenges of Rust: A Mixed-Methods Study. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1269–1281, 2022.

[65] Kai Mindermann, Philipp Keck, and Stefan Wagner. How usable are Rust cryptography APIs? In *Proceedings of the 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 143–154. IEEE, 2018.

[66] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 332–343, 2016.

[67] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter 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*, pages 248–259, 2015.

[68] Peter Leo Gorski, Yasemin Acar, Luigi Lo Iacono, and Sascha Fahl. Listen to developers! a participatory design study on security warnings for cryptographic apis. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020.

[69] Anastasia Danilova, Alena Naiakshina, and Matthew Smith. One size does not fit all: a grounded theory and online survey study of developer preferences for security warning types. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 136–148. IEEE, 2020.

[70] Marcelo Almeida, Grant Cole, Ke Du, Gongming Luo, Shulin Pan, Yu Pan, Kai Qiu, Vishnu Reddy, Haochen Zhang, Yingying Zhu, and Cyrus Omar. RustViz: Interactively Visualizing Ownership and Borrowing. pages 1–10, 2022.

[71] Shundan Xiao, Jim Witschey, and Emerson 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*, pages 1095–1106, 2014.

[72] Jim Witschey, Olga Zielinska, Allaire Welk, Emerson Murphy-Hill, Chris Mayhorn, and Thomas Zimmermann. Quantifying developers' adoption of security tools. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 260–271, 2015.

[73] Julie M Haney and Wayne G Lutters. " it's scary... it's confusing... it's dull": How cybersecurity advocates overcome negative perceptions of security. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 411–425, 2018.

[74] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 598–608. IEEE, 2015.

[75] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[76] Yasemin Acar, Christian Stransky, Dominik Wermke, Charles Weir, Michelle L Mazurek, and Sascha Fahl. Developers need support, too: A survey of security advice for software developers. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 22–26. IEEE, 2017.

[77] Tamara Lopez, Thein Tun, Arosha Bandara, Levine Mark, Bashar Nuseibeh, and Helen Sharp. An Anatomy of Security Conversations in Stack Overflow. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, pages 31–40, 2019.

[78] Mengsu Chen, Felix Fischer, Na Meng, Xiaoyin Wang, and Jens Grossklags. How Reliable is the Crowdsourced Knowledge of Security Implementation? In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 536–547, 2019.

[79] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha 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. IEEE, 2017.

[80] Wei Bai, Omer Akgul, and Michelle L Mazurek. A qualitative investigation of insecure code propagation from online forums. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 34–48. IEEE, 2019.

[81] Peter Leo Gorski, Sebastian Möller, Stephan Wiefling, and Luigi Lo Iacono. "I just looked for the solution!"On Integrating Security-Relevant Information in Non-Security API Documentation to Support Secure Coding Practices. pages 3467–3484, 2022.

[82] Felix Fischer, Yannick Stachelscheid, and Jens Grossklags. The Effect of Google Search on Software Security: Unobtrusive Security Interventions via Content Re-ranking. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3070–3084, 2021.

[83] Aiko Yamashita and Leon Moonen. Surveying developer knowledge and interest in code smells through online freelance marketplaces. In *2013 2nd International Workshop on User Evaluations for Software Engineering Researchers (USER)*, pages 5–8. IEEE, 2013.

[84] Sebastian Baltes and Stephan Diehl. Worse than spam: Issues in sampling software developers. In *Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement*, pages 1–6, 2016.

[85] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, and Matthew Smith. On conducting security developer studies with cs students: Examining a password-storage study with cs students, freelancers, and company developers. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020.

[86] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. Are Students Representatives of Professionals in Software Engineering Experiments? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 666–676, 2015.

[87] Harjot Kaur, Sabrina Amft, Daniel Votipka, Yasemin Acar, and Sascha Fahl. Where to recruit for security development studies: Comparing six software developer samples. 2022.

[88] Mohammad Tahaei and Kami Vaniea. Recruiting participants with programming skills: A comparison of four crowdsourcing platforms and a cs student mailing list. In *CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2022.

[89] Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. Measuring programming experience. In *2012 20th IEEE international conference on program comprehension (ICPC)*, pages 73–82. IEEE, 2012.

[90] Gunnar R Bergersen, Dag IK Sjøberg, and Tore Dybå. Construction and validation of an instrument for measuring programming skill. *IEEE Transactions on Software Engineering*, 40(12):1163–1184, 2014.

[91] Anastasia Danilova, Alena Naiakshina, Stefan Horstmann, and Matthew Smith. Do you really code? designing and evaluating screening questions for online surveys with programmers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 537–548. IEEE, 2021.

[92] Anastasia Danilova, Stefan Horstmann, Matthew Smith, and Alena Naiakshina. Testing time limits in screener questions for online surveys with programmers. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2080–2090, 2022.

[93] Daniel Votipka, Desiree Abrokwa, and Michelle L Mazurek. Building and validating a scale for secure software development self-efficacy. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–20, 2020.

[94] Anastasia Danilova, Alena Naiakshina, Johanna Deuter, and Matthew Smith. Replication: On the Ecological Validity of Online Security Developer Studies: Exploring Deception in a Password-Storage Study with Freelancers.

[95] Nicolas Huaman, Alexander Krause, Dominik Wermke, Jan H. Klemmer, Christian Stransky, Yasemin Acar, and Sascha Fahl. If You {Can't} Get Them to the Lab: Evaluating a Virtual Study Environment with Security Information Workers. pages 313–330, 2022.

[96] Anastasia Danilova, Alena Naiakshina, Anna Rasgauski, and Matthew Smith. Code reviewing as methodology for online security studies with developers-a case study with freelancers on password storage. In *Seventeenth Symposium on Usable Privacy and Security ({SOUPS} 2021)*, pages 397–416, 2021.

[97] Arthur C Graesser, Katja Wiemer-Hastings, Peter Wiemer-Hastings, Roger Kreuz, Tutoring Research Group, et al. Autotutor: A simulation of a human tutor. *Cognitive Systems Research*, 1(1):35–51, 1999.

[98] Sally A Fincher and Anthony V Robins. *The Cambridge handbook of computing education research*. Cambridge University Press, 2019.

[99] Maura Borrego, Elliot P Douglas, and Catherine T Amelink. Quantitative, qualitative, and mixed research methods in engineering education. *Journal of Engineering education*, 98(1):53–66, 2009.

[100] Lecia Jane Barker, Kathy Garvin-Doxas, and Michele Jackson. Defensive climate in the computer science classroom. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 43–47, 2002.

[101] Jeffrey Bonar and Elliot Soloway. Uncovering principles of novice programming. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 10–13, 1983.

[102] Jonas Boustedt. Students' different understandings of class diagrams. *Computer Science Education*, 22(1):29–62, 2012.

[103] Sebastian Dziallas and Sally Fincher. Aspects of graduateness in computing students' narratives. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, pages 181–190, 2016.

[104] Carsten Schulte and Maria Knobelsdorf. Attitudes towards computer science-computing experiences as a starting point and barrier to computer science. In *Proceedings of the third international workshop on Computing education research*, pages 27–38, 2007.

[105] David Socha and Josh Tenenberg. Sketching software in the wild. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1237–1240. IEEE, 2013.

[106] Josh Tenenberg and Sally Fincher. Students designing software: a multi-national, multi-institutional study. *Informatics in Education*, 4(1):143–162, 2005.

[107] James C Spohrer and Elliot Soloway. Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM*, 29(7):624–632, 1986.

[108] Arthur W Combs, Daniel W Soper, and Clifford C Courson. The measurement of self concept and self report. *Educational and Psychological Measurement*, 23(3):493–500, 1963.

[109] Bridget M Reynolds, Theodore F Robles, and Rena L Repetti. Measurement reactivity and fatigue effects in daily diary research with families. *Developmental Psychology*, 52(3):442, 2016.

[110] Kathy Charmaz. *Constructing grounded theory: A practical guide through qualitative analysis*. sage, 2006.

[111] Johnny Salda na. *The coding manual for qualitative researchers*. Sage, 2 edition, 2014.

[112] Andrew F Hayes and Klaus Krippendorff. Answering the call for a standard reliability measure for coding data. *Communication methods and measures*, 1(1):77–89, 2007.

[113] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. Reliability and inter-rater reliability in qualitative research: Norms and guidelines for cscw and hci practice. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):1–23, 2019.

[114] Matthew Smith. Usable Security—The source awakens. San Francisco, CA, January 2016. USENIX Association.

[115] Veroniek Binkhorst, Tobias Fiebig, Katharina Krombholz, Wolter Pieters, et al. Security at the end of the tunnel: The anatomy of vpn mental models among experts and non-experts in a corporate context. In *USENIX Security Symposium*, number 31, 2022.

[116] Noura Alomar, Primal Wijesekera, Edward Qiu, and Serge Egelman. " you've got your nice list of bugs, now what?" vulnerability discovery and management processes in the wild. In *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*, pages 319–339, 2020.

[117] Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016.

[118] Mohammadreza Hazhirpasand, Oscar Nierstrasz, Mohammadhossein Shabani, and Mohammad Ghafari. Hurdles for developers in cryptography. *arXiv preprint arXiv:2108.07141*, 2021.

[119] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L. Mazurek, and Christian 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)*, 2016.

[120] GitLab. What are git version control best practices? https://about.gitlab.com/topics/version-control/version-control-best-practices/, 2022.

[121] Thomas D LaToza, Maryam Arab, Dastyni Loksa, and Amy J Ko. Explicit programming strategies. *Empirical Software Engineering*, 25(4):2416–2449, 2020.

[122] Marlene Scardamalia, Carl Bereiter, and Rosanne Steinbach. Teachability of reflective processes in written composition. *Cognitive science*, 8(2):173–190, 1984.

[123] John H Flavell. Metacognitive aspects of problem solving. *The nature of intelligence*, 1976.

[124] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. *Research methods in human-computer interaction*. Morgan Kaufmann, 2017.

[125] Rock Stevens, Daniel Votipka, Elissa M Redmiles, Colin Ahern, Patrick Sweeney, and Michelle L Mazurek. The battle for new york: a case study of applied digital threat modeling at the enterprise level. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 621–637, 2018.

[126] Elmer Lastdrager, Inés Carvajal Gallardo, Pieter Hartel, and Marianne Junger. How effective is {Anti-Phishing} training for children? In *Thirteenth symposium on usable privacy and security (soups 2017)*, pages 229–239, 2017.

[127] Nick Rutar, Christian B Almazan, and Jeffrey S Foster. A comparison of bug finding tools for java. In *15th International symposium on software reliability engineering*, pages 245–256. IEEE, 2004.

[128] Dejan Baca, Bengt Carlsson, Kai Petersen, and Lars Lundberg. Improving software security with static automated code analysis in an industry setting. *Software: Practice and Experience*, 43(3):259–279, 2013.

[129] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 111–131. Springer, 2010.

[130] Andrew Austin and Laurie Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 97–106. IEEE, 2011.

[131] Nuno Antunes and Marco Vieira. Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services. In *2009 15th IEEE pacific rim international symposium on dependable computing*, pages 301–306. IEEE, 2009.

[132] Larry Suto. Analyzing the effectiveness and coverage of web application security scanners. *San Francisco, October*, 2007.

[133] Larry Suto. Analyzing the accuracy and time costs of web application security scanners. *San Francisco, February*, 2010.

[134] G McGraw and J Steven. Software [in] security: Comparing apples, oranges, and aardvarks (or, all static analysis tools are not created equal, 2011.

[135] Diana Burley, Matt Bishop, Scott Buck, Joseph J. Ekstrom, Lynn Futcher, David Gibson, Elizabeth K. Hawthorne, Siddharth Kaza, Yair Levy, Herbert Mattord, and Allen Parrish. Curriculum guidelines for post-secondary degree programs in cybersecurity. Technical report, ACM, IEEE, AIS, and IFIP, 12 2017.

[136] Mitre. CWE-416: Use After Free. https://cwe.mitre.org/data/definitions/416.html, 2020.

[137] Mitre. CWE-476: NULL Pointer Dereference. https://cwe.mitre.org/data/definitions/476.html, 2020.

[138] Mitre. CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer. https://cwe.mitre.org/data/definitions/119.html, 2020.

[139] Mitre. CWE-125: Out-of-bounds Read. https://cwe.mitre.org/data/definitions/125.html, 2020.

[140] Mitre. CWE-787: Out-of-bounds Write. https://cwe.mitre.org/data/definitions/787.html, 2020.

[141] NIST. CWE Over Time. https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cwe-over-time, 2020.

[142] Google. Chrome: 70% of all security bugs are memory safety issues. https://www.chromium.org/Home/chromium-security/memory-safety, 2020.

[143] Catalin Cimpanu. Microsoft: 70 percent of all security bugs are memory safety issues. https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/, 2019.

[144] Alex Gaynor. What science can tell us about C and C++'s security. https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/, 2020. Presentation at Enigma.

[145] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[146] Mozilla. The Rust programming language. https://developer.mozilla.org/en-US/docs/Mozilla/Rust, 2020.

[147] Rob Pike. Go at Google: Language design in the service of software engineering. https://talks.golang.org/2012/splash.article, 2020.

[148] StackOverflow. Developer Survey Results. https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved, 2020.

[149] Ryan Donovan. Why the developers who use Rust love it so much. https://stackoverflow.blog/2020/06/05/why-the-developers-who-use-rust-love-it-so-much/, 2020.

[150] Jake Goulding. What is Rust and why is it so popular? https://stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular/, 2020.

[151] Jeffrey M Perkel. Why scientists are turning to Rust? *Nature*, 588:186–186, 2020.

[152] Steve Klabnik and Carol Nichols. The Rust Programming Language Book. https://doc.rust-lang.org/1.9.0/book/README.html, 2020.

[153] C. A. R. (Tony) Hoare. Null References: The Billion Dollar Mistake. https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/, 2009. Presentation at QCon.

[154] Mozilla. Rust Programming Language Production. https://www.rust-lang.org/production, 2020.

[155] Greg Guest, Arwen Bunce, and Laura Johnson. How many interviews are enough? an experiment with data saturation and variability. *Field Methods*, 18(1):59–82, 2006.

[156] Juliet Corbin and Anselm Strauss. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications, 2014.

[157] Klaus Krippendorff. Reliability in Content Analysis : Some Common Misconceptions and Recommendations. 2015.

[158] Deen G Freelon. ReCal: Intercoder reliability calculation as a web service. *International Journal of Internet Science*, 5(1):20–33, 2010.

[159] Matthew Lombard, Jennifer Snyder-Duch, and Cheryl Campanella Bracken. Content analysis in mass communication: Assessment and reporting of intercoder reliability. *Human Communication Research*, 28(4):587–604, 2002.

[160] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, Emanuel von Zezschwitz, and Matthew Smith. "If You Want, I Can Store the Encrypted Password": A Password-Storage Field Study with Freelance Developers. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 140:1–140:12, 2019.

[161] IEEE. The Top Programming Languages. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019, 2020.

[162] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J Summers. How do programmers use unsafe rust? *PACMPL*, 4(OOPSLA):1–27, 2020.

[163] Julie M. Haney and Wayne G. Lutters. "It's Scary... It's Confusing... It's Dull": How Cybersecurity Advocates Overcome Negative Perceptions of Security. In *Proceedings of the Symposium on Usable Privacy and Security*, 2018.

[164] Leo A Meyerovich and Ariel S Rabkin. Empirical analysis of programming language adoption. In *Proceedings of the Conference on Object oriented programming systems languages & applications*, pages 1–18, 2013.

[165] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L Mazurek, and Christian Stransky. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 289–305, May 2016.

[166] Nischal Shrestha, Colton Botta, Titus Barik, and Chris Parnin. Here We Go Again: Why Is It Difficult for Developers to Learn Another Programming Language? In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, 2020.

[167] Jim Witschey, Shundan Xiao, and Emerson Murphy-Hill. Technical and personal factors influencing developers' adoption of security tools. In *Proceedings of the ACM Workshop on Security Information Workers*, pages 23–26, 2014.

[168] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. Is rust used safely by software developers? In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 246–257, 2020.

[169] Kelsey R Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L Mazurek. Benefits and drawbacks of adopting a secure programming language: rust as a case study. In *Symposium on Usable Privacy and Security*, 2021.

[170] Stephan Plöger, Mischa Meier, and Matthew Smith. A qualitative usability evaluation of the clang static analyzer and libfuzzer with {CS} students and {CTF} players. In *Seventeenth Symposium on Usable Privacy and Security ({SOUPS} 2021)*, pages 553–572, 2021.

[171] Joe Lewis. Developer observatory. https://github.com/SP2-MC2/Developer-Observatory, 2021.

[172] John Brooke et al. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[173] Kelsey R Fulton, Daniel Votipka, Desiree Abrokwa, Michelle L Mazurek, Michael Hicks, and James Parker. Understanding the how and the why: Exploring secure development practices through a course competition. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1141–1155, 2022.

[174] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2022.

[175] Dan Goodin. Chatgpt is enabling script kiddies to write functional malware. *Ars Technica*, Jan 2023.