ABSTRACT

Title of dissertation: A HUMAN-CENTRIC APPROACH TO SOFTWARE VULNERABILITY DISCOVERY
Daniel J. Votipka
Doctor of Philosophy, 2020

Dissertation directed by: Professor Michelle L. Mazurek
Department of Computer Science

Software security bugs — referred to as vulnerabilities — persist as an important and costly challenge. Significant effort has been exerted toward automatic vulnerability discovery, but human intelligence generally remains required and will remain necessary for the foreseeable future. Therefore, many companies have turned to internal and external (e.g., penetration testing, bug bounties) security experts to manually analyze their code for vulnerabilities. Unfortunately, there are a limited number of qualified experts. Therefore, to improve software security, we must understand how experts search for vulnerabilities and how their processes could be made more efficient, by improving tool usability and targeting the most common vulnerabilities. Additionally, we seek to understand how to improve training to increase the number of experts.

To answer these questions, I begin with an in-depth qualitative analysis of secure development competition submissions to identify common vulnerabilities developers introduce. I found developers struggle to understand and implement complex security concepts, not recognizing how nuanced development decisions could

lead to vulnerabilities. Next, using a cognitive task analysis to investigate experts' and non-experts' vulnerability discovery processes, I observed they use the same process, but differ in the variety of security experiences which inform their searches. Together, these results suggest exposure to and in-depth understanding of potential vulnerabilities as essential for vulnerability discovery.

As a *first* step to leverage both experts and non-experts, I pursued two lines of work: education to support experience development and vulnerability discovery automation interaction improvements. To improve vulnerability discovery tool interaction, I conducted observational interviews of experts' reverse engineering process, an essential and time-consuming component of vulnerability discovery. From this, I provide guidelines for more usable interaction design. For security education, I began with a pedagogical review of security exercises to identify their current strengths and weaknesses. I also developed a psychometric measure for secure software development self-efficacy to support comparisons between educational interventions.

A HUMAN-CENTRIC APPROACH TO SOFTWARE VULNERABILITY DISCOVERY

by

Daniel J. Votipka

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
2020

Advisory Committee:
Professor Michelle L. Mazurek, Chair/Advisor
Professor Michael Hicks
Professor Jeffrey S. Foster
Professor Michael K. Reiter
Professor Katie Shilton

Acknowledgments

Many people have influenced me personally and academically over the years culminating in this thesis. This thesis is as much an attribute to them and their selfless support as it is an example of my academic achievement.

First, and foremost, I owe a lifelong debt of gratitude to my wife, Rhea. Without her tireless support, this would not be possible. She has always been there for me through the good times and the bad, encouraging me at every turn. I also want to recognize my daughter, Luca. She is persistent source of joy in my life and has helped remind me what's most important through this process.

I would also like to thank my advisor, Dr. Michelle Mazurek. She was an invaluable resource academically, but more importantly a mentor and friend I knew I could always count on. I will always contend that selecting Dr. Mazurek as my advisor was the smartest thing I ever did in my academic career. It has been a pleasure to work with and learn from such an extraordinary individual.

I would like to thank Dr. Jeffrey Foster and Dr. Michael Hicks for their support and mentorship throughout my career. Thanks are also due to Dr. Katie Shilton and Dr. Michael Reiter for agreeing to serve on my dissertation committee and for sparing their invaluable time reviewing the manuscript.

My colleagues in the security, privacy, and people lab have enriched my graduate life in many ways and deserve special thanks. Specifically, I want to thank Rock Stevens, Kelsey Fulton, and Omer Akgul for all the time spent helping me hash out ideas and run this research. None of my success would be possible without them.

I would also like to acknowledge help and support from the UMIACS, MC2, CS Department, and IRB staff. I would like to thank Dana Purcell, Tom Hurst, Joseph Smith, and Andrea Dragan in particular. They made every administrative issue simple and were always quick to help and answer my questions. My graduate program was a mostly painless process specifically because of their tireless effort.

Finally, I would like to acknowledge financial support from Accenture, ATT, Galois, Leidos, Patriot Technologies, NCC Group, Trail of Bits, Synposis, ASTech Consulting, Cigital, SuprTek, Cyberpoint, and Lockheed Martin; by a grant from the NSF under awards EDU-1319147 and CNS-1801545; by DARPA under contract FA8750-15-2-0104; by a UMIACS contract under the partnership between the University of Maryland and DoD; by a Google Research Award; and by the U.S. Department of Commerce, National Institute for Standards and Technology, under Cooperative Agreement 70NANB15H330 for all the projects discussed herein.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Many security bugs are discovered every year in production code [75, 82, 277]. These issues, known as vulnerabilities, can be exploited by malicious actors to catastrophic effect [111, 126, 319, 391, 437]. Further, even if companies are able to identify them before an attacker, they still cost significant time and effort to remediate [38, 243, 303, 374, 390, 441].

One approach to this problem is automated vulnerability discovery. Significant advances have been made toward automatically identifying — and in some cases remediating — vulnerabilities prior to code release [16, 25, 26, 106, 264, 349, 368, 382, 383]. However, these results are currently fairly limited: Human intelligence remains necessary at least for the foreseeable future.

In practice, companies rely on multiple rounds of manual code review by the development team themselves, internal security experts, and in some cases external experts (e.g., contracted penetration tests and bug bounties). This approach has the benefit of potentially drawing many eyes to the problem, but each party is inherently limited. Development team members have a deep understanding of their specific component's functionality, but may lack the security expertise necessary to identify vulnerabilities [113, 322]. Conversely, security experts are tasked with

considering the full codebase, limiting their ability to review and understand the nuances of all components completely [322, 340].

In this thesis, I focus on developing an understanding of the humans—at all levels of expertise—at the center of vulnerability discovery. Specifically, I investigate:

1. *What types of vulnerabilities do developers introduce?* (Chapter 3)

2. *How do experts' and non-experts' search for vulnerabilities?* (Chapter 4 and 5)

3. *What are the educational strengths and weaknesses of current online security exercises?* (Chapter 6)

In this thesis, we answer these questions through a series of research studies. First, in Chapter 3, I describe my review (with colleagues) of 94 project submissions to the *Build it, Break it, Fix it* (BIBIFI) secure-coding competition series [347]. In these competitions, teams develop programs to meet a given specification with multiple security and functionality requirements. Then, teams attempt to find and exploit vulnerabilities in other teams' submissions. Scores are determined based functionality, performance, and security.

We reviewed project's code and the 182 unique security vulnerabilities associated with those projects. This manual analysis of this dataset provides implications for improving secure-development training, security-relevant APIs [6, 155, 283], and vulnerability discovery tools.

Simple mistakes, in which the developer attempts a valid security practice but makes a minor programming error, were least common. Mitigations to these

types of mistakes are plentiful. For example, in our data, minimizing the trusted code base (e.g., by avoiding duplication of security-critical code) led to significantly fewer mistakes. Moreover, we believe that modern analysis tools and testing techniques [20, 22, 61, 62, 105, 117, 177, 196, 218, 358, 367, 435] should uncover many of them. All but one of the mistakes in our dataset were found and exploited by opposing teams. In short, this type of bug appears to be both relatively uncommon and amenable to existing tools and best practices, suggesting it can be effectively managed.

On the other hand, vulnerabilities arising from misunderstanding of security concepts were significantly more common. In examining these errors, we identify an important distinction between intuitive and unintuitive security requirements; for example, several teams used encryption to protect confidentiality but failed to also protect integrity. In 45% of projects, teams missed unintuitive requirements altogether, failing to even attempt to implement them. When teams implemented security requirements, most were able to select the correct security primitives to use, but made conceptual errors in attempting to apply a security mechanism. For example, several projects failed to provide randomness when an API expects it. Although common, these vulnerabilities proved harder to exploit, and our qualitative labeling identified 35% as difficult to exploit (compared to none of the simple mistakes). These more complex errors expose a need for APIs less subject to misuse, better documentation, and better security training that focuses on less-intuitive concepts like integrity.

Overall, our findings suggest rethinking strategies to prevent and detect vul-

nerabilities, with more emphasis on conceptual difficulties rather than mistakes. To understand how these conceptual difficulties arise between experts and non-experts in security, in Chapter 4, my colleagues and I performed 25 semi-structured interviews with security experts and non-experts, focusing on the process of finding vulnerabilities in software.

We found that both experts and non-experts describe a similar set of steps for discovering vulnerabilities. Their success in each step, however, depends on their vulnerability discovery experience, their knowledge of underlying systems, available access to the development process, and what motivates them to search for vulnerabilities.

Of these variables, interviewees reported that experience — which differed greatly between experts and non-experts – is most significant to success in vulnerability finding. Differences in experience stem primarily from the fact that experts were typically exposed to a wider variety of vulnerabilities through a broad array of sources including employment, online security exercises, communication with peers, and prior vulnerability reports. On the other hand, we found non-experts were typically exposed to only a narrow set of vulnerabilities through fewer sources, as they primarily search for vulnerabilities in only a single code base, only read bug reports associated with their program, and only participate in small, internal hacking exercises, if any.

Building on the finding that experience is critical for vulnerability discovery, the remainder of this thesis takes *first* steps toward better leveraging the existing experience of experts (Chapter 5) and increasing non-expert exposure to vulnerabili-

ties (Chapters 6 and 7). To improve human-automation interaction for vulnerability discovery, I sought to develop a develop a theoretical model of the RE process—the central step in the vulnerability discovery process identified in Chapter 4. In Chapter 5, along with my colleagues, I conducted a 16-participant, semi-structured observational study. In each session, we asked participants to recreate a recent RE experience while we observed their actions and probed their thought process. Throughout, we tracked the decisions made, mental simulation methods used, questions asked, hypotheses formulated, and beacons (recognizable patterns) identified.

We found that in general, the RE process can be modeled in three phases: overview, sub-component scanning, and focused experimentation. REs begin by establishing a broad view of the program's functionality (*overview*). They use their overview's results to prioritize sub-components—e.g., functions—for further analysis, only performing detailed review of specific sub-components deemed most likely to yield useful results (*sub-component scanning*). As REs review these sub-components, they identify hypotheses and questions that are tested and answered, respectively, through execution or in-depth, typically manual static analysis (*focused experimentation*). The last two phases form a loop. REs develop hypotheses and questions, address them, and use the results to inform their understanding of the program. This produces new questions and hypotheses, and the RE continues to iterate until the overall goal is achieved.

Further, we identified several trends in REs' processes spanning multiple phases. We found that REs use more static analysis in the first two phases and switch to dynamic simulation methods during focused experimentation. We also observed

that experience plays an important role throughout REs' decision-making processes, helping REs prioritize where to search (overview and sub-component scanning), recognize implemented functionality and potential vulnerabilities (sub-component scanning), and select which mental simulation method to employ (all phases). Finally, we found REs choose to use tools to support their analysis when a tool's input and output can be closely associated with the code and when the tools improve code readability. Based on these results, we established five guidelines for designing RE tools.

Switching my focus to consider educational improvements for non-experts, in Chapter 6, my colleagues and I performed an in-depth qualitative review of 30 popular online hacking exercises (68% of all online exercises we identified). As part of our analysis, we completed a sample of 306 unique challenges from these 30 exercises. We evaluated each exercise against a set of recommended pedagogical principles grounded in learning theory [13,47]. We based our approach on previous curriculum evaluation efforts [220], tailoring the specific pedagogical principles we examine for applicability to hacking exercises. Further, we interviewed the organizers of 14 exercises to understand the challenges they face.

We found that no exercise implemented every pedagogical principle, but most were implemented by at least some exercises, some in thoughtful and creative ways. Notable shortcomings include that many exercises lack sufficient structure to help students organize knowledge, and enough feedback to guide their progress through learning objectives. Few organizers had considered *metacognition*, or helping students understand what and how much they have learned. We find that some learning

principles are in tension with each other — such as balancing difficulty with realism — while others are in tension with the competitive origin of many exercises. Finally, we find that community participation brings many benefits to an exercise, but must be carefully managed to ensure educational structures are maintained.

While this work presents likely strengths and weaknesses of current educational interventions, further research measuring actual student outcomes is necessary to confirm our findings and determine optimal ways to balance conflicting pedagogical decisions. Unfortunately, current methods for measuring student educational improvement are severely limited or cumbersome. In Chapter 7, I present mine and my colleagues work developing a light-weight psychometric measure targeted at closing this gap. Specifically, we developed a scale to measure developers' secure-development *self-efficacy*—belief in one's ability to successfully perform a task—which correlates with actual skill in other contexts [32]. This scale measures developers' belief in their ability to complete tasks, such as identifying security problems during software design or implementation. To develop the scale, we followed Netemeyer et al.'s 4-step scale creation process [285]. Throughout this process, our scale was evaluated with 311 software developers and 22 security experts. The final scale consists of 15 items measuring two underlying factors: vulnerability discovery and mitigation, and security communication. We show that our scale performs reliably over multiple samples and behaves as expected with respect to relevant measures from prior work.

Chapter 2: Background and Related Work

We discuss related work across four key areas: the characteristics of and reasons for vulnerabilities developers introduce; the way security experts and non-experts find vulnerabilities; vulnerability discovery tool interaction; security education; and secure development efficacy measures.

2.1 Vulnerabilities Developers Introduce

Several researchers have investigated the types of vulnerabilities developers introduce and the factors associated with their introduction.

2.1.1 Measuring metadata in production code

Several researchers have measured the association of different metadata properties from revision-control systems with vulnerability introduction. In two papers, Meneely et al. investigated metadata from PHP and the Apache HTTP server [270, 272]. They found that vulnerabilities are associated with higher-than-average code churn, committing authors who are new to the codebase, and interactive churn (editing others' code rather than one's own). Follow-up work investigating Chromium found that source code reviewed by more developers was more likely to contain a vul-

nerability, unless reviewed by someone who had participated in a prior vulnerability-fixing review [271]. Significantly earlier, Sliwerski et al. explored mechanisms for identifying bug-fix commits in the CVS archives for Eclipse, finding, e.g., that fix-inducing changes typically span more files than other commits [370]. Perl et al. used metadata from Github and CVEs to train a classifier to identify commits that might contain vulnerabilities [318].

Other researchers have investigated trends in CVEs and the National Vulnerability Database (NVD). Christey et al. examining CVEs from 2001–2006, found noticeable differences in the types of vulnerabilities reported for open- and closed-source operating-system advisories [82]. As a continuation, Chang et al. explored CVEs and the NVD from 2007–2010, showing that the percentage of high-severity vulnerabilities decreased over time, but that more than 80% of all examined vulnerabilities were exploitable via network access without authentication [75]. Our research complements this work by examining a smaller set of vulnerabilities in more depth. While these works focus on metadata about code commits and vulnerability reports, I instead examine the code itself.

2.1.2 Measuring cryptography problems in production code

Lazar et al. discovered that only 17% of cryptography vulnerabilities present in the CVE database were caused by bugs in the libraries themselves, while 83% were caused by developer misuse of the libraries [240]. Egele et al. and Kruger et al. developed analyzers to recognize specific cryptographic problems and found that

nearly 88% [114] and 95% [232] — respectively — of the applications on the Google Play Store make at least one of these mistakes when using cryptographic APIs.

Other researchers used fuzzing and static analysis to identify problems with SSL/TLS implementations in libraries and in Android apps [123, 145]. Focusing on one particular application of cryptography, Reaves et al. uncovered serious vulnerabilities in mobile banking applications related to homemade cryptography, certificate validation, and information leakage [335]. These works examine specific categories of vulnerabilities across many real-world programs; my use of contest data allows me to similarly investigate patterns of errors made when addressing similar tasks, but explore more classes of vulnerabilities.

The most similar work to my investigation of the mistakes developers make and the reasons behind these mistakes is that of Nadi et al. [280]. Nadi et al. investigated the ways developers used Java cryptographic APIs by reviewing the types of questions they ask on StackOverflow; a sample of API uses in public GitHub repositories; and two surveys of developers regarding their experiences with these APIs. They reported that developers found cryptographic libraries hard to understand and use because they were too low-level and did not provide easy-to-follow documentation. My results produce similar conclusions, but expand the problems and languages studied. I also provide deeper insights into the problem since I identified specific errors made by developers and associate these with correlated features of the developer and their environment to identify reasons for these errors instead of relying on self-reporting of issues.

2.1.3 Experiments with developers

In contrast to production-code measurements, other researchers have explored security phenomena through controlled experiments with small, security-focused programming tasks. Oliveira et al. studied developer misuse of cryptographic APIs via Java "puzzles" involving APIs with known misuse cases and found that neither cognitive function nor expertise correlated with ability to avoid security problems [302]. Other researchers have found, in the contexts of cryptography and secure password storage, that while simple APIs do provide security benefits, simplicity is not enough to solve the problems of poor documentation, missing examples, missing features, and insufficient abstractions [6, 283]. Naiakshina et al. found that security is not likely to be considered unless participants were primed to think about it. Further, when it was considered, they regularly misunderstood cryptographic properties — this was true both for students [284], professionals employed at software development companies [282], and freelancers [281].

Finifter et al. compared different teams' attempts to build a secure web application using different tools and frameworks [128]. They found no relationship between programming language and application security, but that automated security mechanisms were effective in preventing vulnerabilities.

I further substantiate many of these findings in a different experimental context: larger programming tasks in which functionality and performance were prioritized along with security, allowing increased ecological validity while still maintaining some quasi-experimental controls.

2.2 How Humans Find Vulnerabilities

Previous work has studied how security experts and non-experts find vulnerabilities. This work has focused on how both populations perform the task of bug identification and the characteristics and motivations of security experts.

2.2.1 Bug identification processes

Aranda et al. studied how developers and testers found 10 performance, security, and functionality bugs in a production environment [17]. They reviewed all reporting artifacts associated with the bugs and interviewed the developers and testers who found and fixed them. They found that bugs were most commonly discovered through manual testing; close cooperation and verbal communication were key to helping developers fix bugs.

Assala and Chiasson interviewed 13 developers to understand whether and how they performed common security tasks during the software development lifecycle, including design and implementation reviews and code analysis [23]. They found that developers could be classified as security adopters — meaning they performed most of the security practices — or security inattentive — they performed very few.

Palombo et al. conducted an ethnographic study, embedding in a software development company, working alongside developers while observing secure and insecure behaviors [310]. They studied the effects of incentive structures, organizational relationships, work flow, and other contextual factors on developer security considerations, finding that vulnerabilities are sometimes knowingly introduced or

overlooked due to organizational or economic pressures.

Other studies have experimentally investigated how effective developers are at looking for vulnerabilities and reverse engineering programs. Edmundson et al. conducted an experiment in manual code review: no participant found all three previously confirmed vulnerabilities, and more experience was not necessarily correlated with more accuracy in code review [113]. Other work suggested that users found more vulnerabilities faster with static analysis than with black-box penetration testing [353].

Fang et al. surveyed security experts who disclosed vulnerabilities in the SecurityFocus repository, asking how participants choose software to investigate, what tools they use, and how they report their findings [125, 168]. They found that hackers typically targeted software they were familiar with as users, predominantly preferred fuzzing tools to static analysis, and preferred full disclosure. Summers et al. studied problem-solving mental models through semi-structured interviews of 18 hackers [380]. They find that hackers require a high tolerance for ambiguity, because they seek to find problems that may or may not exist in a system they did not design. Additionally, Summers et al. observed that hackers rely on discourse with others or visualization techniques (i.e., mapping system semantics on a white-board) to deal with ambiguity and identify the most probable issues.

Becker et al. investigated the technical and cognitive processes of hardware reverse engineering [39]. In their study, Becker et al. asked eight participants to reverse engineer the function of a sample set of Integrated Circuits. Through their observations, they produced a hardware reverse engineering process model and found

participants' with higher working memory capacity performed tasks more efficiently.

Ceccato et al. reviewed detailed reports by three penetration testing teams searching for vulnerabilities in a suite of security-specific programs [68]. The participating teams were asked to record their process for searching the programs, finding vulnerabilities, and exploiting them. My work delves deeper into the specific steps followed to understand a program and identify vulnerabilities. Further, through my interviews, I was able to probe the vulnerability discovery process to elicit more detailed responses.

Bryant investigated reverse engineering using a mixed methods approach, including three semi-structured interviews with reverse engineers and an observational study where four participants completed a predesigned reverse engineering task [52]. Based on his observations, Bryant developed a sense-making model for reverse engineering where reverse engineers generate hypotheses from prior experience and cyclically attempt to (in)validate these hypotheses, generating new hypotheses in the process. My results in Chapter 5 align with these findings; I expand on them, producing a more detailed model describing the specific approaches used and how reverse engineering behaviors change throughout the process. My more detailed model is achieved through a larger sample size and observation of reverse engineering processes on different, real-world programs, demonstrating reverse engineering behaviors to ensure saturation of themes [78, pg. 113-115].

Finally, Thomas et al. interviewed 32 internal security team members [394], asking participants to discuss the tools they used, how they worked with developers, and the challenges they face. They found that their participants used both static

and dynamic analysis tools as a first pass, performing some manual analysis in limited cases. Additionally, these experts indicated that the biggest issue with the tools they used were false positives and difficulty in making configuration changes. I build on this work by studying the steps of expert processes in more depth to determine how to provide experts with the requested ability to better interact with their tools.

I expand on all these prior studies by comparing security experts and non-experts and including participants from multiple companies and bug bounty programs. Also, I thoroughly investigate participants' processes, communication about vulnerabilities and reporting strategies, skill-development, and reasons for using specific tools.

2.2.2 Security expert characteristics

Al-Banna et al. focus on external security professionals, asking both professionals and those who hire them which indicators they believed were the most important to discern security expertise [11]. Similarly, Cowley interviewed 10 malware reverse engineering professionals to understand the necessary skills and define levels of professional development [91]. I borrow the concept of task analysis from this work to guide my interviews while expanding the scope of questions and comparing security experts to non-experts.

Criminology research has also examined why some individuals who find vulnerabilities become cyber criminals, finding that although most hackers work alone,

they improve knowledge and skills in part through mentoring by peers [192,193,431]. While I explicitly do not consider black-hat hackers, I build on these findings with further analysis of how security experts learn skills and create communities.

2.2.3 Empirical measurement of motivations through bug bounty programs

Several researchers have investigated what factors (e.g., money, probability of success) most influence participation and productivity in vulnerability discovery through bug bounty programs. Finifter et al. studied the Firefox and Chrome bug bounty programs [127]. They found that a variable payment structure based on the criticality of the vulnerability led to higher participation rates and a greater diversity of vulnerabilities discovered as more researchers participated in the program. Maillart et al. studied 35 public HackerOne bounty programs, finding that hackers tend to focus on new bounty programs and that a significant portion of vulnerabilities are found shortly after the program starts [258]. The authors suggest that hackers are motivated to find "low-hanging fruit" (i.e., easy to discover vulnerabilities) as quickly as possible, because the expected value of many small payouts is perceived to be greater than for complex, high-reward vulnerabilities that might be "scooped" by a competitor.

While these studies suggest potential motivations for hacker behavior based on observed trends, I directly interview hackers about their motivations. Additionally, these studies do not explore the full decision process of bug bounty participants. This

exploration is important because any effective change to the market needs to consider all the nuances of participant decisions if it hopes to be successful. Additionally, prior work does not compare experts with non-experts. This comparison is necessary, as it suggests ways to best train and allocate resources to all stakeholders in the software development lifecycle.

2.3 Vulnerability Discovery Tool Interaction

Finally, previous work has studied users interactions with vulnerability discovery tools, observing the ways existing tools are used or proposing new interactions.

2.3.1 Usability of current tools

Several researchers have studied the usability of static analysis tools and the ways users respond to provided alert messages. Baca et al. studied this problem in the industry setting, finding low adoption rates of static analysis tools among software developers and attributed this to the high number false-positives generated by the tool, the high initial cost for integrating these tools into their workflow, and a lack of understanding how these tools could be valuable [26]. Johnson et al. produced similar results [209] with users of FindBugs, a popular static analysis tool [197]. Smith et al. built on this work by identifying the questions developers ask when using FindBugs, observing that they generally want to know additional details about the vulnerability, possible attacks, and the ecosystem the program is deployed into [372]. Layman et al. focused on which factors developers use to decide

whether or not to heed an error notice. Layman et al. found that developers were more likely to fix a fault when it was presented while the developer was performing a relevant programming task, if the developer had been conditioned to believing the notifications were accurate, and if the notice provided precise descriptions (e.g. included steps to fix the problem) [239].

Yakdan et al. studied usability issues facing hackers with respect to decompiled code readability [432]. Yakdan et al. found that a simple set of code transformations could significantly increase both students' and professionals' ability to reverse engineer malware.

I also investigate the tools used by security experts and non-experts and issues they face, but perform further in-depth investigations into how they use these tools within the full process of vulnerability discovery, how they learned to use these tools, and what differences exist between experts and non-experts. Differentiating between these two groups helps tailor systems to meet the needs of the different populations and understand if and why adoption differs.

2.3.2 Improved usability for vulnerability discovery

The most similar approach to mine is that by Do et al. who created a Just-in-time static analysis framework called CHEETAH, based on the result user studies investigating the way developers interact with static analysis tools [209,372]. CHEETAH allows developers to run static analyses incrementally as they write new code. This approach allows developers to put the analyses results in context and reduces

the overwhelming feeling of a "wall of alerts". While I follow a similar approach — qualitative user studies to inform tool development —, I focus on manual vulnerability discovery processes by a wider range of users to identify new, richer interaction models.

Shoshitaishvili et al. propose a tool-centered human-assisted paradigm for vulnerability discovery [368]. They suggest a new interaction pattern where users provide on-demand feedback to a automated agent by performing well-defined subtasks to support the agent's analysis. This model leverages the human insights to overcome the agent's deficiencies, while allowing the system to scale significantly beyond limited human resources. To demonstrate this model, Shoshitaishvili et al. present HaCRS, an automated vulnerability discovery agent which allows users to provide seed input to a fuzzing-based system. The system uses this input as a seed for a mutation based fuzzer and returns the amount of path coverage increase from the given seed—therefore, incentivizing seeds that increase coverage. However, the demonstrated interaction model specifically targets non-expert users who cannot understand program internals (e.g., code, control flow diagrams, etc.) and therefore treat the program as a black box.

Focusing on expert users, Kruger et al. propose a specification language to allow cryptography experts to state the requirements for secure usage of cryptographic APIs [232]. Unfortunately, this approach still requires the expert to learn a new language that can be very complicated — hundreds of lines of code for each API.

Nosco et al. consider the problem at the team level, proposing an assembly-

line process to leverage team members of differing skill in conjunction with varying automation support [291]. They find that this coordinated approach provides significant improvements in vulnerability discovery efficiency as opposed to all team members operating independently.

Finally, Sadowski et al. created the TRICORDER program analysis platform, which includes a set of guidelines for usable program analyses, a developer-workflow-integrated environment, and developer feedback support. By integrating in the development environment and incentivizing analyses that are responsive to user needs (through direct user feedback), TRICORDER provides over 93,000 analysis results across the entire Google codebase daily while producing significant positive feedback from developers.

While I agree that these are useful models, my work takes a step back to answers a question necessary to unlocking their potential; what is the best way for security experts to communicate with a tool?

2.4 Computer Security Education

A significant amount of effort has gone into determining what topics should be covered by computer security education, with several organizations (e.g., NIST, AIS SIGSEC, NCSC) setting up multi-year task forces and holding conferences of security experts and educators to answer this question [58, 71, 151, 204, 287, 311, 332]. One common thread from the guidelines produced by these groups is that secure software development is an essential component for education [171]. However, they

do not prescribe any particular pedagogy. While many educators rely on a lecture-based approach, several researchers have proposed novel educational interventions to supplement or replace lectures.

2.4.1 Hands-on exercises

Educators and practitioners have long recognized the benefit of hands-on practice for computer security education, suggesting the inclusion of hacking competitions into the academic pipeline [48, 90, 346]. This has led several researchers to propose exercise development guidelines to teach educators how to build these types of exercises and improve educational outcomes [219, 314, 325, 326]. While many of these guidelines provide limited recommendations for specific pedagogy, Pusey et al. provide suggestions for tailoring challenges to student prior experience [325] and establish a supportive environment for underrepresented populations [326]. In this thesis, I not only consider a broader range of pedagogical dimensions, but also evaluate whether—and why not—these are applied in practice.

New exercises to address specific pedagogy. Several researchers have also proposed novel educational interventions to implement and evaluate particular pedagogical principles. Several researchers have proposed attack oriented exercises, generally in a competition setting to engage participants [44, 83, 110, 112, 135, 208, 346, 423]. Similarly, others have created exercises where students perform attacks against robots [242] and virtual reality headsets [76], presenting security in a visual and easily accessible context. Others have incorporated elements of peer-based instruc-

tion, having students participate in teams of varying experience levels, encouraging collaboration and discussion among and between teams [27, 275, 276, 306]. Other exercises have been developed to go beyond the traditional program exploitation challenges. This includes challenging students to design and implement secure systems [110, 136, 327, 347] and perform penetration testing, navigating through sophisticated network topologies and pivoting from machine-to-machine [44]. Reed et al. evaluate the value of presenting challenges in the context of an overarching narrative, finding students were more likely to complete challenges associated with a storyline [336]. Chung and Cohen outline challenges related to extraneous load (e.g., difficulty getting technology set up to be able to participate) and propose technical solutions to limit initial student burden [83].

Significant effort in this space has focused on evaluating and improving challenge difficulty, to provide feedback and ensure challenges are appropriate for learner experience. Chung and Cohen reflect on years of experiences running the CSAW CTF [234], highlighting the importance of quality assurance in challenge development to ensure appropriate difficulty and feedback within challenges [83]. Owens et al. introduced more easy and medium difficulty challenges into picoCTF [77], to provide a more gradual difficulty slope for beginning students [306]. They found this increased student engagement and reduced participant dropout over previous years of the exercise. Several researchers have added time-on-task tracking for each challenge to compare student behavior (time spent working on a challenge), student-reported feedback, and original assigned challenge difficulty [257,336,416], using this data to tailor future exercise progressions and challenge difficulty ratings. Maennel

et al. further suggested this information could be used by organizers in real-time to identify unintentionally difficult challenges and support struggling students [257].

Each of these studies have shown the benefit of particular pedagogical principles within a given context. My work takes these principles, but asks whether they are being applied broadly in the larger exercise ecosystem and therefore impacting student education.

Broad exercise educational reviews. While much of the literature in security education has focused on individual exercises, there has been some research focused on the ecosystem more broadly. Tobey et al. studied engagement among beginning students in three exercises in the National Cyber League, finding that experienced students are more likely to be engaged and continue participation [395]. However, the authors do not indicate reasons for this lack of engagement. By reviewing the way exercises are organized, we hope to provide some indication insights into how the exercises themselves might impact student participation trends.

Karagiannis et al. reviewed 4 open source platforms for exercise deployment to evaluate their usability with respect to setup and administration for academic purposes [215]. In Chapter 6, we ask an orthogonal question, focusing on students' experiences with specific exercises, not educators' experiences setting up exercises generally.

Burns et al. provide an extensive review of 3600 challenges, outlining the concepts covered and developing a framework to assess the difficulty of each [59]. This work offers a useful complement to our own, as it investigates *what content* exercises

teach and we evaluate *how* they teach.

Finally, Taylor et al. review 36 CTFs, highlighting their organization and structure (e.g., whether content is dynamic or static, whether the exercise is open source). Our work in Chapter 6 provides additional depth to this survey as we consider how specific implementation details impact exercises' educational characteristics.

2.4.2 Other studies of security education

In addition to these hands-on hacking exercises, other work has proposed a variety of security-related educational interventions. These interventions employ some of the pedagogical principles we discuss. Multiple researchers have proposed and evaluated adding secure development education into the developer's daily workflow [324, 420, 424]. Whitney et al. incorporate security nudges into the IDE, providing security context as developers write code [424]. This tool was shown to provide more insights into security flaws than dedicated teaching assistants [389]. Similarly, Nguyen et al. created FixDroid, an Android Studio IDE plugin, which using an Android Linter to flag potentially vulnerable patterns [288]. They also showed that this approach resulted in the introduction of less security errors. While this approach provides significant benefits, providing the right information at the right time, it relies on being able to quickly identify vulnerable code patterns. Therefore it can only be used for a subset of vulnerabilities at this time.

Weir et al. take a Participatory Action Research approach, embedding a security researcher in the development team to support security decision-making and

evaluate this approach's effect over time [420]. Similarly, Poller et al. evaluate the impact of third-party security reviews on security behaviors over time [324]. Other researchers have suggested narrative-based education for computer security. Sherman et al. and Rowe et al. present case studies around exploited systems and have students discuss the cause and potential mitigations [346, 360]. Blasco and Quaglia had students discuss attacks and defenses portrayed in fictional scenarios from popular culture (e.g., *Star Wars: Rogue One*) [43]. Other researchers have had students share stories of relevant experiences [101, 210]. Denning et al. developed a tabletop card game designed to expose participants to general security problems and adversarial thinking through its overarching storyline [99]. Relatedly, Frey et al. developed a tabletop game in which players defend cyber-physical infrastructure, to help players reflect on security-relevant decision processes and strategies [139].

While I do not consider these alternative methods for security education directly in this thesis, the creation of a light-weight measure of secure development self-efficacy makes direct comparison of these interventions possible in future work.

2.5 Secure Development Efficacy Measurement

While human-centered secure development is a growing subfield, work related scale development in this area is limited. Woon and Kankanhalli's secure-development intentions scale focuses on development and includes self-efficacy questions [430]. However, it is a much broader measure, intended to assess several factors influencing secure-development practice adoption. Therefore, their self-efficacy

questions are limited and less concrete than mine (e.g., "I would feel comfortable carrying out secure development of applications on my own"). By focusing specifically on self-efficacy, I provide a more precise measure and identify underlying factors. Because this scale has received only preliminary validation, I did not use it to establish discriminant validity. Rajivan et al. developed a measure assessing security expertise by asking participants if they have performed several network defense and system administration tasks (e.g., configuring a firewall) along with two open-ended questions asking participants to describe security concepts (i.e., certificates and phishing). While this scale targets expert users, it again measures an orthogonal domain (e.g., network defense and system administration). Finally, Campbell et al. propose a metric for cybersecurity aptitude—potential to develop skills necessary for cybersecurity tasks—as opposed to my measure targeted at current skill level [64].

There have also been several efforts to develop scales for efficiently measuring end-user security. Egelman and Peer created the Security Behavior Intentions Scale (SeBIS) [115] and Faklaris et al. established a measure for Security Attitudes (SA-6) [124]. Together these scales cover participants' security thoughts and behaviors; however, due to the difference in domains, my scale measures an orthogonal construct.

Chapter 3: Exploring the Vulnerabilities Developers Introduce

This chapter [1] presents a systematic, in-depth examination (using best practices developed for qualitative assessments) of vulnerabilities present in software projects. In particular, we looked at 94 project submissions to the *Build it, Break it, Fix it* (BIBIFI) secure-coding competition series [347]. In each competition, participating teams (many of which were enrolled in a series of online security courses [147]) first developed programs for either a secure event-logging system, a secure communication system simulating an ATM and a bank, or a scriptable key-value store with role-based access control policies. Teams then attempted to exploit the project submissions of other teams. Scoring aimed to match real-world development constraints: teams were scored based on their project's performance, its feature set (above a minimum baseline), and its ultimate resilience to attack. Our six-month examination considered each project's code and 866 total exploit submissions, corresponding to 182 unique security vulnerabilities associated with those projects.

The BIBIFI competition provides a unique and valuable vantage point for examining the vulnerability landscape, complementing existing field measures and

[1] Published as [411]

lab studies. When looking for trends in open-source projects (field measures), there are confounding factors: Different projects do different things, and were developed under different circumstances, e.g., with different resources and levels of attention. By contrast, in BIBIFI we have many implementations of the same problem carried out by different teams but under similar circumstances. As such, we can postulate the reasons for observed differences with more confidence. At the other end of the spectrum, BIBIFI is less controlled than a lab study, but offers more ecological validity—teams had weeks to build their project submissions, not days, using any languages, tools, or processes they preferred.

3.1 Data

This section presents the Build It, Break It, Fix It (BIBIFI) secure-programming competition [347], the data we gathered from it which forms the basis of our analysis, and reasons why the data may (or may not) represent real-world situations.

3.1.1 Build it, Break it, Fix it

A BIBIFI competition comprises three phases: *building*, *breaking*, and *fixing*. Participating teams can win prizes in both *build-it* and *break-it* categories.

In the first (*build it*) phase, teams are given just under two weeks to build a project that (securely) meets a given specification. During this phase, a team's *build-it score* is determined by the correctness and efficiency of their project, assessed by test cases provided by the contest organizers. All projects must meet a core set of

functionality requirements, but they may optionally implement additional features for more points. Submitted projects may be written in any programming language and are free to use open-source libraries, so long as they can be built on a standard Ubuntu Linux VM.

In the second (*break it*) phase, teams are given access to the source code of their fellow competitors' projects in order to look for vulnerabilities.[2] Once a team identifies a vulnerability, they create a test case (a *break*) that provides evidence of exploitation. Depending on the contest problem, breaks are validated in different ways. One is to compare the output of the break on the target project against that of a "known correct" reference implementation (RI) written by the competition organizers. Another way is by confirming knowledge (or corruption) of sensitive data (produced by the contest organizers) that should have been protected by the target project's implementation. Successful breaks add to a team's *break-it score*, and reduce the target project's team's build-it score.

The final (*fix it*) phase of the contest affords teams the opportunity to fix bugs in their implementation related to submitted breaks. Doing so has the potential benefit that breaks which are superficially different may be unified by a fix, preventing them from being double counted when scoring.

[2]Source code obfuscation was against the rules. Complaints of violations were judged by contest organizers.

3.1.2 Data gathered

We analyzed projects developed by teams participating in four BIBIFI competitions, covering three different programming problems: *secure log*, *secure communication*, and *multiuser database*. (Appendix B.1 provides additional details about the makeup of each competition.) Each problem specification required the teams to consider different security challenges and attacker models. Here we describe each problem, the size/makeup of the reference implementation (for context), and the manner in which breaks were submitted.

Secure log (SL, Fall 2014[3]and Spring 2015, RI size: 1,013 lines of OCaml). This problem asks teams to implement two programs: one to securely append records to a log, and one to query the log's contents. The build-it score is measured by log query/append latency and space utilization, and teams may implement several optional features.

Teams should protect against a malicious adversary with access to the log and the ability to modify it. The adversary does not have access to the keys used to create the log. Teams are expected (but not told explicitly) to utilize cryptographic functions to both encrypt the log and protect its integrity. During the break-it phase, the organizers generate sample logs for each project. Break-it teams demonstrate

[3]The Fall'14 contest data was not included in the original BIBIFI data analysis [347]. It had only 12 teams and was organizationally unusual; notably, build-it teams were originally only allocated 3 days to complete the project, but then were given an extension (with the total time on par with that of later contests). Including Fall'14 in the original data analysis would have required adding a variable (the contest date) to all models, but the small number of submissions would have required sacrificing a more interesting variable to preserve the models' power. In this paper, including Fall'14 is not a problem because we are performing a qualitative rather than quantitative analysis.

compromises to either integrity or confidentiality by manipulating a sample log file to return a differing output or by revealing secret content of a log file.

Secure communication (SC, Fall 2015, RI size: 1,124 lines of Haskell). This problem asks teams to build a pair of client/server programs. These represent a bank and an ATM, which initiates account transactions (e.g., account creation, deposits, withdrawals, etc.). Build-it performance is measured by transaction latency. There are no optional features.

Teams should protect bank data integrity and confidentiality against an adversary acting as a man-in-the-middle (MITM), with the ability to read and manipulate communications between the client and server. Once again, build teams were expected to use cryptographic functions, and to consider challenges such as replay attacks and side-channels. Break-it teams demonstrate exploitations violating confidentiality or integrity of bank data by providing a custom MITM and a script of interactions. Confidentiality violations reveal the secret balance of accounts, while integrity violations manipulate the balance of unauthorized accounts.

Multiuser database (MD, Fall 2016, RI size: 1,080 lines of OCaml). This problem asks teams to create a server that maintains a secure key-value store. Clients submit scripts written in a domain-specific language. A script authenticates with the server and then submits a series of commands to read/write data stored there. Data is protected by role-based access control policies customizable by the data owner, who may (transitively) delegate access control decisions to other prin-

cipals. Build-it performance is assessed by script running time. Optional features take the form of additional script commands.

The problem assumes that an attacker can submit commands to the server, but not snoop on communications. Break-it teams demonstrate vulnerabilities with a script that shows a security-relevant deviation from the behavior of the RI. For example, a target implementation has a confidentiality violation if it returns secret information when the RI denies access.

Project Characteristics. Teams used a variety of languages in their projects. Python was most popular overall (39 teams, 41%), with Java also widely used (19, 20%), and C/C++ third (7 each, 7%). Other languages used by at least one team include Ruby, Perl, Go, Haskell, Scala, PHP, JavaScript Visual Basic, OCaml, C#, and F#. For the secure log problem, projects ranged from 149 to 3857 lines of code (median 1095). secure communication ranged from 355 to 4466 (median 683) and multiuser database from 775 to 5998 (median 1485).

3.1.3 Representativeness: In Favor and Against

Our hope is that the vulnerability particulars and overall trends that we find in BIBIFI data are, at some level, representative of the particulars and trends we might find in real-world code. There are several reasons in favor of this view:

• Scoring incentives match those in the real world. At build-time, scoring favors features and performance—security is known to be important, but is not (yet) a direct concern. Limited time and resources force a choice between uncertain benefit

later or certain benefit now. Such time pressures mimic short release deadlines.

• The projects are substantial, and partially open ended, as in the real world. For all three problems, there is a significant amount to do, and a fair amount of freedom about how to do it. Teams must think carefully about how to design their project to meet the security requirements. All three projects consider data security, which is a general concern, and suggest or require general mechanisms, including cryptography and access control. Teams were free to choose the programming language and libraries they thought would be most successful. While real-world projects are surely much bigger, the BIBIFI projects are big enough that they can stand in for a component of a larger project, and thus present a representative programming challenge for the time given.

• About three-quarters of the teams whose projects we evaluated participated in the contest as the capstone to an on-line course sequence (MOOC) [147]. Two courses in this sequence — software security and cryptography — were directly relevant to contest problems. Although these participants were students, most were also post-degree professionals; overall, participants had a average of 8.9 years software development experience. Further, prior work suggests that in at least some secure development studies, students can substitute effectively for professionals, as only security experience, not general development experience, is correlated with security outcomes [8, 9, 282, 284].

On the other hand, there are several reasons to think the BIBIFI data will not represent the real world:

• Time pressures and other factors may be insufficiently realistic. For example,

while there was no limit on team size (they ranged from 1 to 7 people with a median of 2), some teams might have been too small, or had too little free time, to devote enough energy to the project. That said, the incentive to succeed in the contest in order to pass the course for the MOOC students was high, as they would not receive a diploma for the whole sequence otherwise. For non-MOOC students, prizes were substantial, e.g., $4000 for first prize. While this may not match the incentive in some security-mature companies where security is "part of the job" [174] and continued employment rests on good security practices, prior work suggests that many companies are not security-mature [23].

• We only examine three secure-development scenarios. These problems involve common security goals and mechanisms, but results may not generalize outside them to other security-critical tasks.

• BIBIFI does not simulate all realistic development settings. For example, in some larger companies, developers are supported by teams of security experts [394] who provide design suggestions and set requirements, whereas BIBIFI participants carry out the entire development task. BIBIFI participants choose the programming language and libraries to use, whereas at a company the developers may have these choices made for them. BIBIFI participants are focused on building a working software package from scratch, whereas developers at companies are often tasked with modifying, deploying, and maintaining existing software or services. These differences are worthy of further study on their own. Nevertheless, we feel that the baseline of studying mistakes made by developers tasked with the development of a full (but small) piece of software is an interesting one, and may indeed support or

inform alternative approaches such as these.

• To allow automated break scoring, teams must submit exploits to prove the existence of vulnerabilities. This can be a costly process for some vulnerabilities that require complex timing attacks or brute force. This likely biases the exploits identified by breaker teams. To address this issue, two researchers performed a manual review of each project to identify and record any hard to exploit vulnerabilities.

• Finally, because teams were primed by the competition to consider security, they are perhaps more likely to try to design and implement their code securely [283, 284]. While this does not necessarily give us an accurate picture of developer behaviors in the real world, it does mirror situations where developers are motivated to consider security, e.g., by security experts in larger companies, and it allows us to identify mistakes made even by such developers.

Ultimately, the best way to see to what extent the BIBIFI data represents the situation in the real world is to assess the connection empirically, e.g., through direct observations of real-world development processes, and through assessment of empirical data, e.g., (internal or external) bug trackers or vulnerability databases. This paper's results makes such an assessment possible: Our characterization of the BIBIFI data can be a basis of future comparisons to real-world scenarios.

3.2 Qualitative Coding

We are interested in characterizing the vulnerabilities developers introduce when writing programs with security requirements. In particular, we pose the fol-

lowing research questions:

RQ1 What *types* of vulnerabilities do developers introduce? Are they conceptual flaws in their understanding of security requirements or coding mistakes?

RQ2 How much *control* does an attacker gain by exploiting the vulnerabilities, and what is the effect?

RQ3 How *exploitable* are the vulnerabilities? What level of insight is required and how much work is necessary?

Answers to these questions can provide guidance about which interventions—tools, policy, and education—might be (most) effective, and how they should be prioritized. To obtain answers, we manually examined 94 BIBIFI projects (67% of the total), the 866 breaks submitted during the competition, and the 42 additional vulnerabilities identified by the researchers through manual review. We performed a rigorous *iterative open coding* [379, pg. 101-122] of each project and introduced vulnerability. Iterative open coding is a systematic method, with origins in qualitative social-science research, for producing consistent, reliable labels ('codes') for key concepts in unstructured data.[4] The collection of labels is called a *codebook*. The ultimate codebook we developed provides labels for vulnerabilities—their type, attacker control, and exploitability—and for features of the programs that contained them.

This section begins by describing the codebook itself, then describes how we produced it. An analysis of the coded data is presented in the next section.

[4]Hence, our use of the term "coding" refers to a type of structured categorization for data analysis, not a synonym for programming.

Variable	Levels	Description	Alpha [179]
Type	(See Table 3.2)	What caused the vulnerability to be introduced	0.85, 0.82
Attacker Control	Full / Partial	What amount of the data is impacted by an exploit	0.82
Discovery Difficulty	Execution / Source / Deep Insight	What level of sophistication would an attacker need to find the vulnerability	0.80
Exploit Difficulty	Single step / Few steps / Many steps / Probabilistic	How hard would it be for an attacker to exploit the vulnerability once discovered	1

Table 3.1: Summary of the vulnerability codebook.

3.2.1 Codebook

Both projects and vulnerabilities are characterized by several labels. We refer to these labels as *variables* and their possible values as *levels*.

3.2.1.1 Vulnerability codebook

To measure the types of vulnerabilities in each project, we characterized them across four variables: *Type*, *Attacker Control*, *Discovery Difficulty*, and *Exploit Difficulty*. The structure of our vulnerability codebook is given in Table 3.1.[5] Our coding scheme is adapted in part from the CVSS system for scoring vulnerabilities [129]. In particular, *Attacker Control* and *Exploit Difficulty* relate to the CVSS concepts of *Impact*, *Attack Complexity*, and *Exploitability*. We do not use CVSS directly, in part because some CVSS categories are irrelevant to our dataset (e.g., none of the contest problems involve human interactions). Further, we followed

[5]The last column indicates Krippendorff's α statistic [179], which we discuss in Section 4.3.

qualitative best practices of letting natural (anti)patterns emerge from the data, modifying the categorizations we apply accordingly.

Vulnerability type. The *Type* variable characterizes the vulnerability's underlying source (RQ1). For example, a vulnerability in which encryption initialization vectors (IVs) are reused is classified as having the issue *insufficient randomness*. The underlying source of this issue is a conceptual misunderstanding of how to implement a security concept. We identified more than 20 different issues grouped into three types; these are discussed in detail in Section 3.3.

Attacker control. The *Attacker Control* variable characterizes the impact of a vulnerability's exploitation (RQ2) as either a full compromise of the targeted data or a partial one. For example, a secure-communication vulnerability in which an attacker can corrupt any message without detection would be a full compromise, while only being able to corrupt some bits in the initial transmission would be coded as partial.

Exploitability. We indicated the difficulty to produce an exploit (RQ3) using two variables, *Discovery Difficulty* and *Exploit Difficulty*. The first characterizes the amount of knowledge the attacker must have to initially find the vulnerability. There are three possible levels: only needing to observe the project's inputs and outputs (*Execution*); needing to view the project's source code (*Source*); or needing to understand key algorithmic concepts (*Deep insight*). For example, in

the secure-log problem, a project that simply stored all events in a plaintext file with no encryption would be coded as *Execution* since neither source code nor deep insight would be required for exploitation. The second variable, *Exploit Difficulty*, describes the amount of work needed to exploit the vulnerability once discovered. This variable has four possible levels of increasing difficulty depending on the number of steps required: only a single step, a small deterministic set of steps, a large deterministic set of steps, or a large probabilistic set of steps. As an example, in the secure-communication problem, if encrypted packet lengths for failure messages are predictable and different from successes, this introduces an information leakage exploitable over multiple probabilistic steps. The attacker can use a binary search to identify the initial deposited amount by requesting withdrawals of varying values and observing which succeed.

3.2.1.2 Project codebook

To understand the reasons teams introduced certain types of vulnerabilities, we coded several project features as well. We tracked several objective features including the lines of code (LoC) as an estimate of project complexity; the IEEE programming-language rankings [203] as an estimate of language maturity (*Popularity*); and whether the team included test cases as an indication of whether the team spent time auditing their project.

We also attempted to code projects more qualitatively. For example, the variable *Minimal Trusted Code* assessed whether the security-relevant functionality

was implemented in single location, or whether it was duplicated (unnecessarily) throughout the codebase. We included this variable to understand whether adherence to security development best practices had an effect on the vulnerabilities introduced [58, pg. 32-36]. The remaining variables we coded (most of which don't feature in our forthcoming analysis) are discussed in Appendix B.2.

3.2.2 Coding Process

Now we turn our attention to the process we used to develop the codebook just described. Our process had two steps: Selecting a set of projects for analysis, and iteratively developing a codebook by examining those projects.

3.2.2.1 Project Selection

We started with 142 qualifying projects in total, drawn from four competitions involving the three problems. Manually analyzing every project would be too time consuming, so we decided to consider a sample of 94 projects—just under 67% of the total. We did not sample them randomly, for two reasons. First, the numbers of projects implementing each problem are unbalanced; e.g., secure log comprises just over 50% of the total. Second, a substantial number of projects had no break submitted against them—57 in total (or 40%). A purely random sample from the 142 could lead us to considering too many (or too few) projects without breaks, or too many from a particular problem category.

To address these issues, our sampling procedure worked as follows. First, we

bucketed projects by the problem solved, and sampled from each bucket separately. This ensured that we had roughly 67% of the total projects for each problem. Second, for each bucket, we separated projects with a submitted break from those without one, and sampled 67% of the projects from each. This ensured we maintained the relative break/non-break ratio of the overall project set. Lastly, within the group of projects with a break, we divided them into four equally-sized quartiles based on number of breaks found during the competition, sampling evenly from each. Doing so further ensured that the distribution of projects we analyzed matched the contest-break distribution in the whole set.

One assumption of our procedure was that the frequency of breaks submitted by break-it teams matches the frequency of vulnerabilities actually present in the projects. We could not sample based on the latter, because we did not have ground truth at the outset; only after analyzing the projects ourselves could we know the vulnerabilities that might have been missed. However, we can check this assumption after the fact. To do so, we performed a Spearman rank correlation test to compare the number of breaks and vulnerabilities introduced in each project [421, pg. 508]. Correlation, according to this test, indicates that if one project had more contest breaks than another, it would also have more vulnerabilities, i.e., be ranked higher according to both variables. We observed that there was statistically significant correlation between the number of breaks identified and the underlying number of vulnerabilities introduced ($\rho = 0.70$, $p < 0.001$). Further, according to Cohen's standard, this correlation is "large," as ρ is above 0.50 [85]. As a result, we are confident that our sampling procedure, as hoped, obtained a good representation of

the overall dataset.

We note that an average of 27 teams per competition, plus two researchers, examined each project to identify vulnerabilities. We expect that this high number of reviewers, as well as the researchers' security expertise and intimate knowledge of the problem specifications, allowed us to identify the majority of vulnerabilities.

3.2.2.2 Coding

To develop our codebooks, I and another researcher first cooperatively examined 11 projects. For each, we reviewed associated breaks and independently audited the project for vulnerabilities. We met and discussed their reviews (totaling 42 vulnerabilities) to establish the initial codebook.

At this point, I and a third researcher independently coded breaks in rounds of approximately 30 each, and again independently audited projects' unidentified vulnerabilities. After each round, we met, discussed cases where our codes differed, reached a consensus, and updated the codebook.

This process continued until a reasonable level of inter-rater reliability was reached for each variable. Inter-rater reliability measures the agreement or consensus between different researchers applying the same codebook. To measure inter-rater reliability, we used the Krippendorff's α statistic [179]. Krippendorff's α is a conservative measure which considers improvement over simply guessing. Krippendorff et al. recommend a threshold of $\alpha > 0.8$ as a sufficient level of agreement [179]. The final Krippendorff's alpha for each variable is given in Table 3.1. Because the

Types observed in the MD problem were very different from the other two problems (e.g., cryptography vs. access control related), we calculated inter-rater reliability separately for this problem to ensure reliability was maintained in this different data. Once a reliable codebook was established, the remaining 34 projects (with 166 associated breaks) were divided evenly between myself and the other researcher and coded separately.

Overall, this process took approximately six months of consistent effort by two researchers.

3.3 Vulnerability Types

Our manual analysis of 94 BIBIFI projects identified 182 unique vulnerabilities. We categorized each based on our codebook into 23 different issues. Table 3.2 presents this data. Issues are organized according to three main types: *No Implementation*, *Misunderstanding*, and *Mistake* (RQ1). These were determined systematically using *axial coding*, which identifies connections between codes and extracts higher-level themes [379, pg. 123-142]. For each issue type, the table gives both the number of vulnerabilities and the number of projects that included a vulnerability of that type. A dash indicates that a vulnerability does not apply to a problem.

This section presents descriptions and examples for each type. When presenting examples, we identify particular projects using a shortened version of the problem and a randomly assigned ID. In the next section, we consider trends in this data, specifically involving vulnerability type prevalence, attacker control, and

exploitability.

3.3.1 *No Implementation*

We coded a vulnerability type as *No Implementation* when a team failed to even attempt to implement a necessary security mechanism. Presumably, they did not realize it was needed. This type is further divided into the sub-type *All Intuitive*, *Some Intuitive*, and *Unintuitive*. In the first two sub-types teams did not implement all or some, respectively, of the requirements that were either directly mentioned in the problem specification or were intuitive (e.g., the need for encryption to provide confidentiality). The *Unintuitive* sub-type was used if the security requirement was not directly stated or was otherwise unintuitive (e.g., using MAC to provide integrity [5]).

Two issues were typed as *All Intuitive*: not using encryption in the secure log (P=3, V=3) and secure communication (P=2, V=2) problems and not performing any of the specified access control checks in the multiuser database problem (P=0, V=0). The *Some Intuitive* sub-type was used when teams did not implement some of the nine multiuser database problem access-control checks (P=10, V=18). For example, several teams failed to check authorization for commands only `admin` should be able to issue For *Unintuitive* vulnerabilities, there were four issues: teams failed to include a MAC to protect data integrity in the secure log (P=16, V=16) and secure communication (P=7, V=7) problems; prevent side-channel data leakage through packet sizes or success/failure responses in the secure communication

(P=11, V=11) and multiuser database (P=4, V=4) problems, respectively; prevent replay attacks (P=7, V=7) in the secure communication problem; and check the chain of rights delegation (P=4, V=4) in the multiuser database problem.

3.3.2 *Misunderstanding*

A vulnerability type was coded as *Misunderstanding* when a team attempted to implement a security mechanism, but failed due to a conceptual misunderstanding. We sub-typed these as either *Bad Choice* or *Conceptual Error*.

3.3.2.1 *Bad Choice*

Five issues fall under this sub-type, which categorizes algorithmic choices that are inherently insecure.

The first three issues relate to the incorrect implementation of encryption and/or integrity checks in the SL and SC problems: use of an algorithm without any secret component, i.e., a key (P=8, V=8), weak algorithms (P=4, V=5), or homemade encryption (P=2, V=2). As an example of a weak algorithm, SL-69 simply XOR'd key-length chunks of the text with the user-provided key to generate the final ciphertext. Therefore, the attacker could simply extract two key-length chunks of the ciphertext, XOR them together and produce the key.

The next issue identifies a weak access-control design for the MD problem, which could not handle all use cases (P=5, V=6). For example, MD-14 implemented delegation improperly. In the MD problem, a *default delegator* may be set by the

administrator, and new users should receive the rights this delegator has when they are created. However, MD-14 granted rights not when a user was created, but when they accessed particular data. If the default delegator received access to data between time of the user's creation and time of access, the user would be incorrectly provided access to this data.

The final issue (potentially) applies to all three problems: use of libraries that could lead to memory corruption. In this case, team SL-81 chose to use *strcpy* when processing user input, and in one instance failed to validate it, allowing an overflow. Rather than code this as *Mistake*, we considered it a bad choice because a safe function (*strlcpy*) could have been used instead to avoid the security issue.

3.3.2.2 *Conceptual Error*

Teams that chose a secure design often introduced a vulnerability in their implementation due to a conceptual misunderstanding (rather than a simple mistake). This *Conceptual Error* sub-type manifested in six ways.

Most commonly, teams used a fixed value when an random or unpredictable one was necessary (P=26, V=26). This included using hardcoded account passwords (P=8, V=8), encryption keys (P=3, V=3), salts (P=3, V=3), or using a fixed IV (V=12, N=12).

```
var nextNonce uint64 = 1337
...
func sendMessage(conn *net.Conn, message
  []byte) (err error) {
  var box []byte
```

```
  var nonce [24]byte


  byteOrder.PutUint64(nonce[:], nextNonce)
  box = secretbox.Seal(box, message, &nonce,
    &sharedSecret)
  var packet = Packet{Size: uint64(len(box)),
    Nonce: nextNonce}
  nextNonce++
  writer := *conn
  err = binary.Write(writer, byteOrder, packet)
  ...
}
```

Listing 3.1: SC-76 Used a hardcoded IV seed.

Sometimes chosen values were not fixed, but not sufficiently unpredictable (P=7, V=8). This included using a timestamp- based nonce, but making the accepted window too large (P=3, V=3); using repeated nonces or IVs (P=3, V=4); or using predictable IVs (P=1, V=1). As an example, SC-76 attempted to use a counter-based IV to ensure IV uniqueness. Listing 3.1 shows that nonce nextNonce is incremented after each message. Unfortunately, the counter is re-initialized every time the client makes a new transaction, so all messages to the server are encrypted with the same IV. Further, both the client and server initialize their counter with the same number (1337 in Line 1 of Listing 3.1), so the messages to and from the server for the first transaction share an IV. If team SC-76 had maintained the counter across executions of the client (i.e., by persisting it to a file) and used a different seed for the client and server, both problems would be avoided.

Other teams set up a security mechanism correctly, but only protected a subset of necessary components (P=9, V=10). For example, Team SL-66 generated a MAC for each log entry separately, preventing an attacker from modifying an entry, but allowing them to arbitrarily delete, duplicate, or reorder log entries. Team SC-24 used an HTTP library to handle client-server communication, then performed encryption on each packet's data segment. As such, an attacker can read or manipulate the HTTP headers; e.g., by changing the HTTP return status the attacker could cause the receiver to drop a legitimate packet.

In three cases, the team passed data to a library that failed to handle it properly (P=3, V=3). For example, MD-27 used an access-control library that takes rules as input and returns whether there exists a chain of delegations leading to the content owner. However, the library cannot detect loops in the delegation chain. If a loop in the rules exists, the library enters an infinite loop and the server becomes completely unresponsive. (We chose to categorize this as a *Conceptual Error* vulnerability instead of a *Mistake* because the teams violate the library developers' assumption as opposed to making a mistake in their code.)

```
self.db = self.sql.connect(filename, timeout=30)
self.db.execute('pragma␣key="' + token + '";')
self.db.execute('PRAGMA␣kdf_iter='
  + str(Utils.KDF_ITER) + ';')
self.db.execute('PRAGMA␣cipher_use_MAC␣=␣OFF;')
...
```

Listing 3.2: SL-22 disabled automatic MAC in SQLCipher library.

Finally, one team simply disabled protections provided transparently by the

library (P=1, V=1). Team SL-22 used the SQLCipher library to implement their log as an SQL database. The library provides encryption and integrity checks in the background, abstracting these requirements from the developer. Listing 3.2 shows the code they used to initialize the database. Unfortunately, on line 5, they explicitly disabled the automatic MAC.

3.3.3 *Mistake*

Finally, some teams attempted to implement the solution correctly, but made a mistake that led to a vulnerability. The mistake type is composed of five sub-types. Some teams did not properly handle errors putting the program into an observably bad state (causing it to be hung or crash). This included not having sufficient checks to avoid a hung state, e.g., infinite loop while checking the delegation chain in the MD problem, not catching a runtime error causing the program to crash (P=5, V=9), or allowing a pointer with a null value to be written to, causing a program crash and potential exploitation (P=1, V=1).

```
def checkReplay(nonce,timestamp):
    #First we check for tiemstamp delta
    dateTimeStamp = datetime.strptime(timestamp,
      '%Y-%m-%d␣%H:%M:%S.%f')
    deltaTime = datetime.utcnow() - dateTimeStamp
    if deltaTime.seconds > MAX_DELAY:
        raise Exception("ERROR:Expired␣nonce␣")
    #The we check if it is in the table
    global bank
    if (nonce in bank.nonceData):
```

```
raise Exception("ERROR:Reinyected␣package")
```

Listing 3.3: SC-80 forgot to save the nonce.

Other mistakes led to logically incorrect execution behaviors. This included mistakes related to the control flow logic (P=5, V=10) or skipping steps in the algorithm entirely. Listing 3.3 shows an example of SC-80 forgetting a necessary step in the algorithm. On line 10, they check to see if the nonce was seen in the list of previous nonces (`bank.nonceData`) and raise an exception indicating a replay attack. Unfortunately, they never add the new nonce into bank.nonceData, so the check on line 10 always returns true.

3.4 Analysis of Vulnerabilities

This section considers the prevalence (RQ1) of each vulnerability type as reported in Table 3.2 along with the attacker control (RQ2), and exploitability (RQ3) of introduced types. Overall, we found that simple implementation mistakes (*Mistake*) were far less prevalent than vulnerabilities related to more fundamental lack of security knowledge (*No Implementation*, *Misunderstanding*). Mistakes were almost always exploited by at least one other team during the Break It phase, but higher-level errors were exploited less often. Teams that that were careful to minimize the footprint of security-critical code were less likely to introduce mistakes.

3.4.1 Prevalence

To understand the observed frequencies of different types and sub-types, we performed planned pairwise comparisons among them. In particular, we use a Chi-squared test—appropriate for categorical data [140]—to compare the number of projects containing vulnerabilities of one type against the projects with another, assessing the effect size (ϕ) and significance (p-value) of the difference. We similarly compare sub-types of the same type. Because we are doing multiple comparisons, we adjust the results using a Benjamini-Hochberg (BH) correction [40]. We calculate the effect size as the measure of association of the two variables tested (ϕ) [92, 282-283]. As a rule of thumb, $\phi \geq 0.1$ represents a small effect, ≥ 0.3 a medium effect, and ≥ 0.5 a large effect [85]. A p-value less than 0.05 after correction is considered significant.

Teams often did not understand security concepts. We found that both types of vulnerabilities relating to a lack of security knowledge—*No Implementation* ($\phi = 0.29$, $p < 0.001$) and *Misunderstanding* ($\phi = 0.35$, $p < 0.001$)—were significantly more likely (roughly medium effect size) to be introduced than vulnerabilities caused by programming *Mistake*s. We observed no significant difference between *No Implementation* and *Misunderstanding* ($\phi = 0.05$, $p = 0.46$). These results indicate that efforts to address conceptual gaps should be prioritized. Focusing on these issues of understanding, we make the following observations.

Unintuitive security requirements are commonly skipped. Of the *No Implementation* vulnerabilities, we found that the *Unintuitive* sub-type was much more common than its *All Intuitive* ($\phi = 0.44$, $p < 0.001$) or *Some Intuitive* ($\phi = 0.37$, $p < 0.001$) counterparts. The two more intuitive sub-types did not significantly differ ($\phi = 0.08$, $p = 0.32$) This indicates that developers do attempt to provide security — at least when incentivized to do so — but struggle to consider all the unintuitive ways an adversary could attack a system. Therefore, they regularly leave out some necessary controls.

Teams often used the right security primitives, but did not know how to use them correctly. Among the *Misunderstanding* vulnerabilities, we found that the *Conceptual Error* sub-type was significantly more likely to occur than *Bad Choice* ($\phi = 0.23$, $p = .003$). This indicates that if developers know what security controls to implement, they are often able to identify (or are guided to) the correct primitives to use. However, they do not always conform to the assumptions of "normal use" made by the library developers.

Complexity breeds *Mistakes*. We found that complexity within both the problem itself and also the approach taken by the team has a significant effect on the number of *Mistakes* introduced. This trend was uncovered by a poisson regression (appropriate for count data) [63, 67-106] we performed for issues in the *Mistakes* type.[6]

[6]We selected initial covariates for the regression related to the language used, best practices followed (e.g., *Minimal Trusted Code*), team characteristics (e.g., years of developer experience), and the contest

Table 3.3 shows that *Mistakes* were most common in the MD problem and least common in the SL problem. This is shown in the second row of the table. The log estimate (E) of 6.68 indicates that teams were 6.68× more likely to introduce *Mistakes* in MD than in the baseline secure communication case. In the fourth column, the 95% confidence interval (CI) provides a high-likelihood range for this estimate between 2.90× and 15.37×. Finally, the p-value of < 0.001 indicates that this result is significant. This effect likely reflects the fact that the MD problem was the most complex, requiring teams to write a command parser, handle network communication, and implement nine different access control checks.

Similar logic demonstrates that teams were only 0.06× as likely to make a mistake in the SL problem compared to the SC baseline. The SL problem was on the other side of the complexity spectrum, only requiring the team to parse command-line input and read and write securely from disk.

Similarly, not implementing the secure components multiple times (*Minimal Trusted Code*) was associated with an 0.36× decrease in *Mistakes*, suggesting that violating the "Economy of Mechanism" principle [352] by adding unnecessary complexity leads to *Mistakes*. As an example of this effect, MD-74 reimplemented their access control checks four times throughout the project. Unfortunately, when they realized the implementation was incorrect in one place, they did not update the other three.

problem. From all possible initial factor combinations, we chose the model with minimum Bayesian Information Criteria—a standard metric for model fit [329]. We include further details of the initial covariates and the selection process in Appendix B.3, along with discussion of other regressions we tried but do not include for lack of space.

Mistakes are more common in popular languages. Teams that used more popular languages are expected to have a 1.09× increase in *Mistakes* for every one unit increase in popularity over the mean *Popularity*[7] ($p = 0.009$). This means, for example, a language 5 points more popular than average would be associated with a 1.54× increase in *Mistakes*. One possible explanation is that this variable proxies for experience, as many participants who used less popular languages also knew more languages and were more experienced.

Finally, while the *LoC* were found to have a significant effect on the number of *Mistakes* introduced, the estimate is so close to one as to be almost negligible.

No significant effect observed for developer experience or security training. Across all vulnerability types, we did not observe any difference in vulnerabilities introduced between MOOC and non-MOOC participants or participants with more development experience. While this does not guarantee a lack of effect, it is likely that increased development experience and security training have, at most, a small impact.

3.4.2 Exploit Difficulty and Attacker control

To answer RQ2 and RQ3, we consider how the different vulnerability types differ from each other in difficulty to exploit, as well as in the degree of attacker control they allow. We distinguish three metrics of difficulty: our qualitative assessment of the difficulty of finding the vulnerability (*Discovery Difficulty*); our qualitative

[7]The mean *Popularity* score was 91.5. Therefore, C—whose *Popularity* score of 92 was nearest to the mean—can be considered representative the language of average popularity.

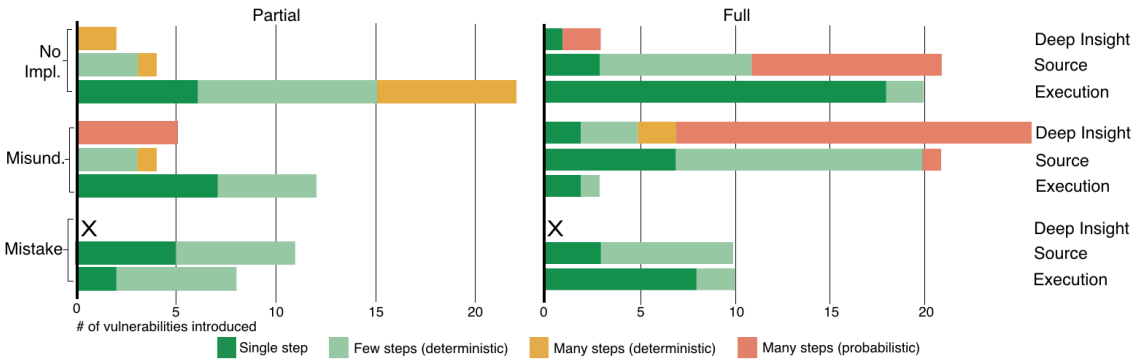

Figure 3.1: # vulnerabilities introduced for each type divided by *Discovery Difficulty*, *Exploit Difficulty* and *Attacker Control*.

assessment of the difficulty of exploiting the vulnerability (*Exploit Difficulty*); and whether a competitor team actually found and exploited the vulnerability (*Actual Exploitation*). Figure 3.1 shows the number of vulnerabilities for each type with each bar divided by *Exploit Difficulty*, bars grouped by *Discovery Difficulty*, and the left and right charts showing partial and full attacker control vulnerabilities, respectively.

To compare these metrics across different vulnerability types and sub-types, we primarily use the same set of planned pairwise Chi-squared tests described in Section 3.4.1. When necessary, we substitute Fisher's Exact Test (FET), which is more appropriate when some of the values being compared are less than five [130]. For convenience of analysis, we binned *Discovery Difficulty* into *Easy* (execution) and *Hard* (source, deep insight). We similarly binned *Exploit Difficulty* into *Easy* (single-step, few steps) and *Hard* (many steps, deterministic or probabilistic).

***Misunderstandings* are rated as hard to find.** Identifying *Misunderstanding* vulnerabilities often required the attacker to determine the developer's exact

approach and have a good understanding of the algorithms, data structures, or libraries they used. As such, we rated *Misunderstanding* vulnerabilities as hard to find significantly more often than both *No Implementation* ($\phi = 0.52$, $p < 0.001$) and *Mistake* ($\phi = 0.30$, $p = 0.02$) vulnerabilities.

Interestingly, we did not observe a significant difference in actual exploitation between the *Misunderstanding* and *No Implementation* types. This suggests that even though *Misunderstanding* vulnerabilities were rated as more difficult to find, sufficient code review can help close this gap in practice.

That being said, *Misunderstandings* were the least common *Type* to be actually exploited by Break It teams. Specifically, using a weak algorithm (Not Exploited=3, Exploited=2), using a fixed value (Not Exploited=14, Exploited=12), and using a homemade algorithm (Not Exploited=1, Exploited=1) were actually exploited in at most half of all identified cases. These vulnerabilities presented a mix of challenges, with some rated as difficult to find and others difficult to exploit. In the homemade encryption case (SL-61), the vulnerability took some time to find, because the implementation code was difficult to read. However, once an attacker realizes that the team has essentially reimplemented the Wired Equivalent Protocol (WEP), a simple check of Wikipedia reveals the exploit. Conversely, seeing that a non-random IV was used for encryption is easy, but successful exploitation of this flaw can require significant time and effort.

***No Implementations* are rated as easy to find.** Unsurprisingly, a majority of *No Implementation* vulnerabilities were rated as easy to find (V=42, 58% of *No*

Implementations). For example, in the SC problem, an auditor could simply check whether encryption, an integrity check, and a nonce were used. If not, then the project can be exploited. None of the *All Intuitive* or *Some Intuitive* vulnerabilities were rated as difficult to exploit; however, 45% of *Unintuitive* vulnerabilities were (V=22). The difference between *Unintuitive* and *Some Intuitive* is significant ($\phi = 0.38$, $p = 0.003$), but (likely due to sample size) the difference between *Unintuitive* and *All Intuitive* is not ($\phi = 0.17$, $p = 0.17$).

As an example, SL-7 did not use a MAC to detect modifications to their encrypted files. This mistake is very simple to identify, but it was not exploited by any of the BIBIFI teams. The likely reason for this was that SL-7 stored the log data in a JSON blob before encrypting. Therefore, any modifications made to the encrypted text must maintain the JSON structure after decryption, or the exploit will fail. The attack could require a large number of tests to find a suitable modification.

***Mistakes* are rated as easy to find and exploit.** We rated all *Mistakes* as easy to exploit. This is significantly different from both *No Implementation* ($\phi = 0.43$, $p = 0.001$) and *Misunderstanding* ($\phi = 0.51$, $p < 0.001$) vulnerabilities, which were rated as easy to exploit less frequently. Similarly, *Mistakes* were actually exploited during the Break It phase significantly more often than either *Misunderstanding* ($\phi = 0.35$, $p = 0.001$) or *No Implementation* ($\phi = 0.28$, $p = 0.006$). In fact, only one *Mistake* (0.03%) was not actually exploited by any Break It team. These results suggest that although *Mistakes* were least common, any that do find their way into

production code are likely to be found and exploited. Fortunately, our results also suggest that code review may be sufficient to find many of these vulnerabilities. (We note that this assumes that the source is available, which may not be the case when a developer relies on third-party software.)

No significant difference in attacker control. We find no significant differences between types or sub-types in the incidence of full and partial attacker control. This result is likely partially due to the fact that partial attacker control vulnerabilities still have practically important consequences. Because of this fact, our BIBIFI did not distinguish between attacker control levels when awarding points; i.e., partial attacker control vulnerabilities received as many points as full attacker control. The effect of more nuanced scoring could be investigated in future work. We do observe a trend that *Misunderstanding* vulnerabilities exhibited full attacker control more often (V=50, 70% of *Misunderstandings*) than *No Implementation* and *Mistake* (V=44, 61% and V=20, 51%, respectively); this trend specifically could be further investigated in future studies focusing on attacker control.

3.5 Discussion and Recommendations

Our results are consistent with real-world observations, add weight to existing recommendations, and suggest prioritizations of possible solutions.

Our vulnerabilities compared to real-world vulnerabilities. While we compiled our list of vulnerabilities by exploring BIBIFI projects, we find that our list

closely resembles both Mitre's CWE and OWASP's Top Ten [278,305] lists. Overlapping vulnerabilities include: broken authentication (e.g., insufficient randomness), broken access control, security misconfiguration (e.g., using an algorithm incorrectly or with the wrong default values), and sensitive data exposure (e.g. side-channel leak).

Get the help of a security expert. In some large organizations, developers working with cryptography and other security-specific features might be required to use security-expert determine tools and patterns to use or have a security expert perform a review. Our results reaffirm this practice, when possible, as participants were most likely to struggle with security concepts avoidable through expert review.

API design. Our results support the basic idea that security controls are best applied transparently, e.g., using simple APIs [155]. However, while many teams used APIs that provide security (e.g., encryption) transparently, they were still frequently misused (e.g., failing to initialize using a unique IV or failing to employ stream-based operation to avoid replay attacks). It may be beneficial to organize solutions around general use cases, so that developers only need to know the use case and not the security requirements.

API documentation. API usage problems could be a matter of documentation, as suggested by prior work [6, 283]. For example, teams SC-18 and SC-19 used TLS socket libraries but did not enable client-side authentication, as needed by the

problem. This failure appears to have occurred because client-side authentication is disabled by default, but this fact is not mentioned in the documentation.[8] Defaults within an API should be safe and without ambiguity [155]. As another example, SL-22 (Listing 3.2) disabled the automatic integrity checks of the SQLCipher library. Their commit message stated "Improve performance by disabling per-page MAC protection." We know that this change was made to improve performance, but it is possible they assumed they were only disabling the "per-page" integrity check while a full database check remained. The documentation is unclear about this.[9]

Security education. Even the best documented APIs are useless when teams fail to apply security at all, as we observed frequently. A lack of education is an easy scapegoat, but we note that many of the teams in our data had completed a cybersecurity MOOC prior to the competition. We reviewed lecture slides and found that all needed security controls for the BIBIFI problems were discussed. While only three teams failed to include *All Intuitive* requirements (5% of MOOC teams), a majority of teams failed to include *Unintuitive* requirements (P=33, 55% of MOOC teams). It could be that the topics were not driven home in a sufficiently meaningful manner. An environment like BIBIFI, where developers practice implementing security concepts and receive feedback regarding mistakes, could help. Future work should consider how well competitors from one contest do in follow-on contests.

[8] `https://golang.org/pkg/crypto/tls/#Listen` and `https://www.openssl.org/docs/manmaster/man3/SSL_new.html`

[9] `https://www.zetetic.net/sqlcipher/sqlcipher-api/#cipher_use_MAC`

Vulnerability analysis tools. There is significant interest in automating security vulnerability discovery (or preventing vulnerability introduction) through the use of code analysis tools. Such tools may have found some of the vulnerabilities we examined in our study. For example, static analyses like SpotBugs/Findbugs [20,196], Infer [62], and FlowDroid [22]; symbolic executors like KLEE [61] and angr [367]; fuzz testers like AFL [435] or libfuzzer [358]; and dynamic analyses like libdft [218] and TaintDroid [117] could have uncovered vulnerabilities relating to memory corruption, improper parameter use (like a fixed IV [105]), and missing error checks. However, they would not have applied to the majority of vulnerabilities we saw, which are often design-level, conceptual issues. An interesting question is how automation could be used to address security requirements at design time.

Determining security expertise. Our results indicate that the reason teams most often did not implement security was due to a lack of knowledge. However, neither years of development experience nor whether security training had been completed had a significant effect on whether any of the vulnerability types were introduced. This finding is consistent with prior research [302] and suggests the need for a new measure of security experience. In Chapter 4, we investigate the divide between experts and non-experts in vulnerability discovery and in Chapter 7 we present a lightweight psychometric scale for measuring this difference.

		Issue	Secure log P=52[1]	Secure log V=53[2]	Secure Comms. P=27	Secure Comms. V=64	Multiuser database P=15	Multiuser database V=65
No Implementation	*All Int.*	No encryption	3 (6%)	3 (6%)	2 (7%)	2 (3%)	–	–
		No access control	–	–	–	–	0 (0%)	0 (0%)
		Total	3 (6%)	3 (6%)	2 (7%)	2 (3%)	–	–
	Some Int.	Missing some access control	–	–	–	–	10 (67%)	18 (28%)
		Total	–	–	–	–	10 (67%)	18 (28%)
	Unintuitive	No MAC	16 (31%)	16 (30%)	7 (26%)	7 (11%)	–	–
		Side-channel	–	–	11 (41%)	11 (17%)	4 (15%)	4 (6%)
		Replay attack	–	–	7 (26%)	7 (11%)	–	–
		No recursive delegation check	–	–	–	–	4 (27%)	4 (6%)
		Total	16 (31%)	16 (30%)	18 (67%)	25 (39%)	8 (53%)	8 (12%)
		Total	17 (33%)	19 (36%)	18 (67%)	27 (42%)	12 (80%)	26 (40%)
Misunderstanding	*Bad Choice*	Unkeyed function	6 (12%)	6 (11%)	2 (7%)	2 (3%)	–	–
		Weak crypto	4 (8%)	5 (9%)	0 (0%)	0 (0%)	–	–
		Homemade crypto	2 (4%)	2 (4%)	0 (0%)	0 (0%)	–	–
		Weak AC design	–	–	–	–	5 (33%)	6 (9%)
		Mem. corruption	1 (2%)	1 (2%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
		Total	13 (25%)	14 (26%)	2 (7%)	2 (3%)	5 (33%)	6 (9%)
	Conceptual Error	Fixed value	12 (23%)	12 (23%)	6 (22%)	6 (9%)	8 (53%)	8 (12%)
		Randomness	2 (4%)	3 (6%)	5 (19%)	5 (8%)	0 (0%)	0 (0%)
		Security on data subset	3 (6%)	3 (6%)	6 (22%)	7 (11%)	0 (0%)	0 (0%)
		Library cannot handle input	0 (0%)	0 (0%)	1 (4%)	1 (2%)	2 (13%)	2 (3%)
		Disabled protections	1 (2%)	1 (2%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
		Resource exhaustion	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (7%)	1 (2%)
		Total	17 (33%)	19 (36%)	15 (56%)	19 (30%)	9 (60%)	11 (17%)
		Total	28 (54%)	33 (62%)	15 (56%)	21 (33%)	10 (67%)	17 (26%)
Mistake	–	Insufficient error checking	0 (0%)	0 (0%)	8 (30%)	8 (12%)	4 (27%)	4 (6%)
		Runtime error	0 (0%)	0 (0%)	1 (4%)	1 (2%)	4 (27%)	8 (12%)
		Ctrl flow mistake	0 (0%)	0 (0%)	1 (4%)	1 (2%)	4 (27%)	9 (14%)
		Skipped step	0 (0%)	0 (0%)	4 (15%)	6 (9%)	1 (2%)	1 (2%)
		Null write	1 (2%)	1 (2%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
		Total	1 (2%)	1 (2%)	11 (41%)	16 (25%)	8 (53%)	22 (34%)

[1] Number of projects submitted to the competition
[2] Number of unique vulnerabilities introduced
[3] Total percentages are based on the counts of applicable projects

Table 3.2: Number of vulnerabilities for each issue and the number of projects each vulnerability was introduced in.

Variable	Value	Log Estimate	CI	*p*-value
Problem	SC	–	–	–
	MD	**6.68**	**[2.90, 15.37]**	**$< 0.001*$**
	SL	**0.06**	**[0.01, 0.43]**	**0.006***
Min Trust	False	–	–	–
	True	**0.36**	**[0.17, 0.76]**	**0.007***
Popularity	C (91.5)	**1.09**	**[1.02, 1.15]**	**0.009***
LoC	1274.81	**0.99**	**[0.99, 0.99]**	**0.006***

***Significant effect** – Base case (Log Estimate defined as 1)

Table 3.3: Summary of regression over *Mistake* vulnerabilities. Pseudo R^2 measures for this model were 0.47 (McFadden) and 0.72 (Nagelkerke).

Chapter 4: Exploring the Vulnerability Discovery Processes of Experts and Non-Experts

Observing that developers commonly introduce vulnerabilities due to misunderstandings of security concepts, we next consider how people try to identify and remedy these vulnerabilities in practice. The job of finding vulnerabilities prior to release is often assigned to software testers who typically aim to root out all bugs — performance, functionality, and security — prior to release. Unfortunately, general software testers do not typically have the training or the expertise necessary to find all security bugs, and thus many are released into the wild [113].

Consequently, experts freelancers known as "white-hat hackers" examine released software for vulnerabilities that they can submit to bug bounty programs, often aiming to develop sufficient credibility and skills to be contracted directly by companies for their expertise [258, 431]. Bug bounty programs offer "bounties" (e.g., money, swag, or recognition) to anyone who identifies a vulnerability and discloses it to the vendor. By tapping into the wide population of white-hat hackers, companies have seen significant benefits to product security, including higher numbers of vulnerabilities found and improvements in the expertise of in-house software testers and developers as they learn from the vulnerabilities reported by

others [55, 127, 163, 258, 269, 438].

This vulnerability-finding ecosystem has important benefits, but overall it remains fairly ad-hoc, and there is significant room for improvement. Discovering more vulnerabilities prior to release would save time, money, and company reputation; protect product users; and avoid the long, slow process of patch adoption [38, 243, 303, 374, 390, 441]. Bug bounty markets, which are typically dominated by a few highly-active participants [55, 127, 163, 269], lack cognitive diversity[1], which is specifically important to thoroughly vet software for security bugs [26, 258]. Bug bounty programs can also exhibit communication problems that lead to low signal-to-noise ratios [438]. Evidence suggests that simply raising bounty prices is not sufficient to address these issues [199, 439].

To improve this overall ecosystem, therefore, we must better understand how it works. Several researchers have considered the economic and security impact of bug bounty programs [12, 127, 214, 307, 331]; however, little research has investigated the human processes of benign vulnerability finding. In this chapter [2], along with my colleagues, I take a first step toward improving this understanding. We interviewed 25 software testers and white-hat hackers (collectively, *practitioners*), focusing on the process of finding vulnerabilities in software: why they choose specific software to study, what tools they use, how they develop the necessary skills, and how they communicate with other relevant actors (e.g., developers and peers).

[1] The way people think and the perspectives and previous experiences they bring to bear on a problem [381, pg. 40-65].

[2] Published as [413]

4.1 Methodology

To understand the vulnerability discovery processes used by our target populations, we conducted semi-structured interviews with software testers and white-hat hackers (henceforth *hackers* for simplicity) between April and May 2017. To support rigorous qualitative results, we conducted interviews until new themes stopped emerging (25 participants) [78, pg. 113-115]. Because we interview more than the 12-20 participants suggested by qualitative research best practices literature, our work can provide strong direction for future quantitative work and generalizable design recommendations [158].

Below, we describe our recruitment process, the development and pre-testing of our interview protocol, our data analysis procedures, and the limitations of our work. This study was approved by our university's Institutional Review Board (IRB).

4.1.1 Recruitment

Because software testers and hackers are difficult to recruit [17, 125, 168], we used three sources to find participants: software testing and vulnerability discovery organizations, public bug bounty data, and personal contacts.

Related organizations. To recruit hackers, we contacted the leadership of two popular bug bounty platforms and several top-ranked Capture-the-Flag (CTF) teams. We gathered CTF team contact information when it was made publicly available on

CTFTime.org [97], a website that hosts information about CTF teams and competitions.

To reach software testers, we contacted the most popular Meetup [268] groups with "Software Testing" listed in their description, all the IEEE chapters in our geographical region, and two popular professional testing organizations: the Association for Software Testing [1] and the Ministry of Testing [2].

Public bug bounty data. We also collected publicly available contact information for hackers from bug bounty websites. One of the most popular bug bounty platforms, HackerOne [164], maintains profile pages for each of its members which commonly include the hacker's contact information. Additionally, the Chromium [148] and Firefox [279] public bug trackers provide the email addresses of anyone who has submitted a bug report. To identify reporters who successfully submitted vulnerabilities, we followed the process outlined by Finifter et. al. by searching for specific security-relevant labels [127].

Personal contacts. We asked colleagues in related industries to recruit their coworkers. We also used snowball sampling (asking participants to recruit peers) at the end of the recruitment phase to ensure we had sufficient participation. This recruitment source accounts for three participants.

Advertisement considerations. We found that hackers were highly privacy-sensitive, and testers were generally concerned with protecting their companies' in-

tellectual property, complicating recruiting. To mitigate this, we carefully designed our recruiting advertisements and materials to emphasize the legitimacy of our research institution and to provide reassurance that participant information would be kept confidential and that we would not ask for sensitive details.

Participant screening. Due to the specialized nature of the studied populations, we asked all volunteers to complete a 20-question survey to confirm they had the necessary skills and experience. The survey assessed participants' background in vulnerability discovery (e.g., number of security bugs discovered, percent of income from vulnerability discovery, programs they have participated in, types of vulnerabilities found) and their technical skills (e.g., development experience, reverse engineering, system administration). It also concluded with basic demographic questions. We drew these questions from similar surveys distributed by popular bug bounty platforms [55, 163]. We provide the full set of survey questions in Appendix A.1.

We selected participants to represent a broad range of vulnerability discovery experience, software specializations (i.e., mobile, web, host), and technical skills. When survey responses matched in these categories, we selected randomly. To determine the participants' software specialization, we asked them to indicate the percent of vulnerabilities they discovered in each type of software. We deem the software type with the highest reported percentage the participant's speciality. If no software type exceeded 40% of all vulnerabilities found, we consider the participant a generalist (i.e., they do not specialize in any particular software type).

4.1.2 Interview protocol

We performed semi-structured, video teleconference[3] interviews, which took between 40 and 75 minutes. All interviews were conducted by a single interviewer. Using a semi-structured protocol, the interviewer focused primarily on the set of questions given in Appendix A.2, with the option to ask follow-ups or skip questions that were already answered [176]. Each interview was divided along three lines of questioning: general experience, task analysis, and skill development.

Prior to the main study, we conducted four pilot interviews (two testers, two hackers) to pre-test the questions and ensure validity. We iteratively updated our protocol following these interviews, until we reached the final protocol detailed below.

General experience. We began the interviews by asking participants to expand on their screening-survey responses regarding vulnerability discovery experience. Specifically, we asked their motivation behind doing this type of work (e.g. altruism, fun, curiosity, money) and why they focus (or do not focus) on a specific type of vulnerability or software.

Task analysis. Next, we asked participants what steps they take to find vulnerabilities. Specifically, we focused on the following sub-tasks of vulnerability discovery:

- **Program selection.** How do they decide the pieces of software to investigate?

[3]Interviews were conducted via video teleconference because it was geographically infeasible to meet face-to-face.

- **Vulnerability search.** What steps are taken to search for vulnerabilities?
- **Reporting.** How do they report discovered vulnerabilities? What information do they include in their reports?

To induce in-depth responses, we had participants perform a hierarchical task analysis focused on these three sub-tasks. Hierarchical task analysis is a process of systematically identifying a task's goals and operations and decomposing them into sub-goals and sub-operations [15]. Each operation is defined by its goal, the set of inputs which conditionally activate it, a set of actions, and the feedback or output that determine when the operation is complete and which follow-on operations are required. Hierarchical task analysis was developed to analyze complex, non-repetitive, cognitively loaded tasks to identify errors or inefficiencies in the process. We adopted this for our study, as it provides a useful framework for eliciting details from experts who typically perform some parts of tasks automatically and subconsciously [15].

For each sub-operation identified, we also asked participants to discuss any specific tools they use, what skills are useful to complete this step, how they learned and developed their process for completing the necessary actions, and how the steps they take differ across software and vulnerability types.

Skill development. Finally, we asked participants to describe how they developed the skills necessary to find vulnerabilities. Here, we focused on their learning process and how they interact with other members of their respective communities.

During the task analysis portion of the interview, we asked participants to explain how they learned how to complete certain tasks. In this segment, we broadened this line of questioning and asked what development steps they recommend to newcomers to the field. This question was intended to elicit additional learning sources that may have been missed previously and highlight steps participants believe are the most important.

Finally, we asked each participant to describe their interactions with other members of their local community and the vulnerability discovery and software tester community at large. Specifically, we discussed who they interact with, the forms of their interaction (e.g., one-to-one, large groups), how these interactions are carried out (e.g., conferences, online forums, direct messaging), and what types of information are discussed.

4.1.3 Data analysis

The interviews were analyzed using iterative open coding [379, pg. 101-122]. When all the interviews were completed, four members of the research team transcribed 10 interviews. The remaining 15 interviews were transcribed by an external transcription service. I and another researcher independently coded each interview, building the codebook incrementally and re-coding previously coded interviews. This process was repeated until all interviews were coded. Our codes were then compared to determine inter-coder reliability using the ReCal2 software package [138]. We use Krippendorff's Alpha (α) to measure inter-coder reliability as it

accounts for chance agreements [179].

The α after coding all the interviews was .68. Krippendorff recommends using α values between .667 and .80 only in studies "where tentative conclusions are still acceptable" [230]; and other work has suggested a higher minimum threshold of .70 for exploratory studies [253]. To achieve more conclusive results, we recoded the 16 of our 85 codes with an α less than .70. For each code, the coders discussed a subset of the disagreements, adjusted code definitions as necessary to clarify inclusion/exclusion conditions, and re-coded all the interviews with the updated codebook. After re-coding, the α for the study was .85. Additionally, all individual codes' αs were above .70.

Next, we grouped the identified codes into related categories. In total, there were six categories describing the participants' discovery process (i.e., Information Gathering, Program Understanding, Attack Surface, Exploration, Vulnerability Recognition, and Reporting) and four categories regarding factors that influenced participants' implementation of this process (i.e., Vulnerability Discovery Experience, Underlying System Knowledge, Access to the Development Process, and Motivation). We then performed an axial coding to find connections between categories and between codes within categories [379, pg. 123-142]. Based on the categories and connections between them, we derive a theory describing the process practitioners use to find vulnerabilities, the factors that influence their implementation of this process, and how testers and hackers differ with respect to their process and implementation.

4.1.4 Limitations

Our study has several limitations common to exploratory qualitative research. A lack of complete recall is especially prominent in studies like ours, where participants are asked to describe expert tasks [15]. We employ a hierarchical task analysis in our interview protocol to improve the thoroughness of information elicited. Participants may have also adjusted their answers to portray themselves as more or less skilled, if they were concerned with the interviewer's perception of them [190, 397]. Additionally, there could be selection bias among the testers and hackers studied. Because we explicitly stated the purpose of the study when recruiting, it is possible that those with experience or an interest in security were more likely to respond to our request. Also, since some hackers tend to be more privacy sensitive, some may have decided not to participate in order to protect their identity or intellectual property. To partially mitigate these issues, we recruited through a wide variety of sources and interviewed a diverse pool of participants to increase the likelihood that relevant ideas would be stated by at least one participant. Finally, for each finding, we give the number of testers and hackers that expressed a concept, to indicate prevalence. However, if a participant did not mention a specific idea, that does not necessarily indicate disagreement; they may have simply failed to state it. For these reasons, we do not use statistical hypothesis tests for comparison among participants. Our results do not necessarily generalize beyond our sample; however, they suggest many directions for future work and provide novel insights into the human factors of software vulnerability discovery.

4.2 Participants

We received 49 responses to our screening survey. We selected 10 testers and 15 hackers. (Themes related to their vulnerability discovery process converged more quickly with testers than with hackers, so we required fewer interviews [78]). Table 5.1 shows our participants' demographics, including their self-reported vulnerability-discovery skill level (on a scale from 0-5, with 0 indicating no skill and 5 indicating an expert), self-reported number of vulnerabilities they have discovered, company size (only applicable for testers), and the method used to recruit them.

Hacker demographics match prior surveys. Our study demographics are relatively congruent with hacker demographics reported in studies from the popular bug bounty services HackerOne [163] and BugCrowd [56]. 90% of HackerOne's 70,000 users were younger than 34; 60% of BugCrowd's 38,000 users are 18-29 and 34% are 30-44 years old. Our hacker population was 60% under 30 and 90% under 40 years old. Regarding education, 84% of BugCrowd hackers have attended college and 21% have a graduate degree; 93% of our hackers have attended college and 33% have a graduate degree.

Testers are more diverse than hackers. In contrast to our hacker population, none of our software testers were under 30 and only 60% were under 40 years old. All of our testers have some college education, but only one has a graduate degree.

With respect to ethnicity and gender, the software tester group was much more diverse, at 60% male and 60% Caucasian.

Hackers reported higher vulnerability discovery skills. As expected, there is a contrast in vulnerability finding skills between testers and hackers. The hacker population self-reported an average skill level of 3.5, whereas software testers self-reported an average skill of 2.5. This self-reported measure cannot be used to directly compare participants' abilities; instead, it indicates their self-efficacy, telling us that testers tend to be less confident in their ability to find security bugs.

Interestingly, despite the hacker population possessing more vulnerability finding experience, more software testers self-reported having discovered more than 500 vulnerabilities. However, we note that the number of vulnerabilities is not necessarily representative of participant skill. It may instead depend on their specialization. For example, participants who focused on web applications reported finding more vulnerabilities, but these are generally considered less complex and therefore are less profitable in bug bounties [53].

4.3 Vulnerability discovery process

Perhaps our most surprising result is that hackers and testers described a similar exploratory process for vulnerability finding. They first focus on learning what the program does, then use their intuition and experience to find ways to perform unintended, malicious actions. Across participants, this process was generally broken

ID[1,2]	Educ.	Skill	Vulns. Fnd	Org. Sz	Src[3]
T1W	B.S	1	0-3	$> 20K$	O
T2W	B.S.	2	0-3	100	O
T3W	B.S	3	26-50	150	O
T4G	SC	5	> 500	200-10K	O
T5W	B.S.	4	> 500	200	O
T6W	B.S.	3	51-100	60K	O
T7G	B.S.	4	> 500	50	O
T8H	Assoc.	0	101-500	2K	O
T9W	B.S.	0	0-3	2K	C
T10W	M.S.	3	0-3	10-50K	O
H1H	B.S.	4	11-25	-	O
H2H	B.S.	4	51-100	-	O
H3G	M.S.	5	> 500	-	O
H4H	M.S.	4	26-50	-	O
H5M	B.S.	4	101-500	-	O
H6G	M.S.	3	101-500	-	O
H7W	M.S.	3	26-50	-	O
H8M	SC	5	101-500	-	C
H9G	H.S.	4	26-50	-	O
H10H	SC	2	11-25	-	O
H11W	B.S.	4	51-100	-	O
H12W	B.S.	1	11-25	-	O
H13W	B.S.	4	> 500	-	C
H14W	M.S.	4	101-500	-	P
H15W	B.S.	2	26-50	-	P

[1] IDs are coded by population (T: Tester, H: Hacker) in date order
[2] Software Specialization – W: Web, H: Host, M: Mobile, G: General
[3] Recruitment method – O: Related Organization, P: Public Bug Bounty Data, C: Personal Contact

Table 4.1: Participant demographics.

into five phases: *Information Gathering*, *Program Understanding*, *Attack Surface*, *Exploration*, *Vulnerability Recognition*, and *Reporting*. Participants described the second (Program Understanding) through fourth (Exploration) phases as a loop that they iterate until a vulnerability is found. Figure 4.1 shows the process our participants described, as well as the factors that influence the process. In all the category graphs in this paper, we represent process categories as hexagons, influencing categories as rectangles, and items that determine the influencing categories as ovals. Additionally, we represent relationships between categories with arrows whose direction indicates the direction of influence. For readability, we color arrows from influencers to process categories to indicate the influence category they are

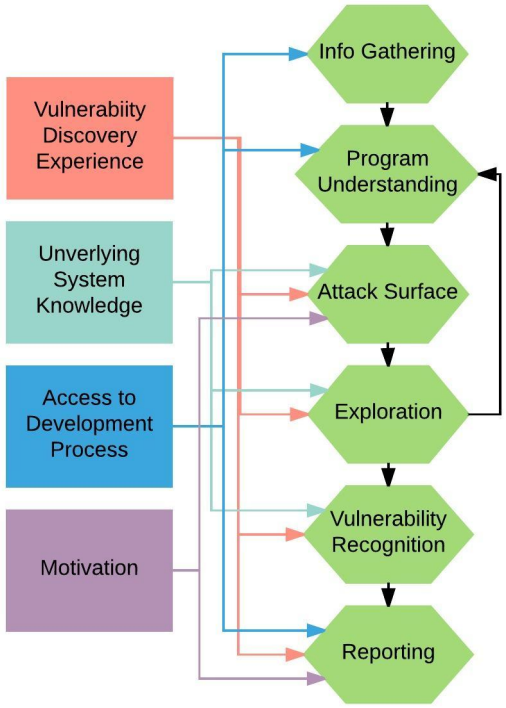

Figure 4.1: Vulnerability-finding process and influencing factors.

derived from.

In this section, we briefly outline the overall vulnerability discovery process. In the following section, we discuss in detail the factors which influence the execution of each phase, and which exhibit greater differences between our two populations.

Information gathering. In the initial phase of vulnerability discovery, practitioners quickly collect preliminary information about the program to develop context prior to reading any code or executing the program (T=6, H=14). This includes finding any previous bugs reported in the program (T=3, H=9), determining the age of the code and update history (i.e., looking at change logs to identify old or recently updated code segments) (T=5, H=9), and identifying the libraries used

(T=1, H=2). This phase's goal is to develop an understanding of prior efforts, as well as the technologies the program is built on. Information Gathering is also used specifically by some hackers to decide whether to expend additional effort or move on to a different target (T=0, H=3).

Program understanding. Next, our participants try to determine how the program operates, how it interacts with its environment (i.e., the user, network, etc.), and how individual components interact with each other. Initially, this step is based on communication with developers (T=7, H=0) or on reading documentation (T=5, H=5), when available. Through iterations of the loop (i.e., Program Understanding, Attack Surface, and Exploration), as they execute the program and read code, practitioners learn about the program by its behavior (T=6, H=11). T4G described iteratively building up an idea of the program structure by "touching a little bit everything, and then you are organizing that structure in your head... [and] you can formalize it [with the code]." H9G tries to get into the developer's mindset by rewriting the code himself. He starts by thinking about the possible "inputs from the management or business side that [go] into the specification," then he writes a version of the program himself and "look[s] for matches between the machine code I'm seeing [the original program] and the machine code that my C++ program produces."

Attack surface. Our participants then identify how a user can interact with the program (T=9, H=15). Their goal is to determine what an attacker can manipulate

and how they can influence program execution. This allows our participants to focus only on critical components of the program. H4H explains "I look at things that I can touch, [for example] what can I get to from the network. ... That's necessary to narrow down the target[s]." Our participants discussed looking for direct program inputs (T=9, H=10), such as website form fields, as well as indirect inputs (T=4, H=10), such as data pulled from their social media page or backend communication to a server.

Exploration. Next, practitioners explore the effect of a range of inputs to see whether it is possible to perform some malicious action by providing data the program mishandles. H5M described this as "enumerating all the variable[s] and all the parameters... I can quickly make a bunch of accounts and see how the user ID changes and how it associates one user with multiple devices." Our participants described a range of approaches to exploring program behavior, typically a combination of executing the program with a set of test inputs (T=9, H=11) and code inspection (T=6, H=12).

Of the tools mentioned during our interviews, almost all were used in this phase. Our participants reported preferring tools that automate simple, repetitive tasks so that they can focus on more complicated problems (T=4, H=13). Such tasks include quickly searching through code or network captures (T=2, H=10), providing suggestions for test cases (T=5, H=3), or making code easier to read (e.g., callee-caller method cross-referencing, variable renaming) (T=1, H=6). We found that hackers were much more likely to utilize tools to automate this phase

of the process, preferring dynamic analyses (e.g., fuzzing) (T=5, H=12) over static analyses (e.g., symbolic execution) (T=0, H=2), which matches Hafiz and Fang's findings [168].

On the other hand, seven hackers mentioned specifically focusing on doing things manually, or using tools that aided them in doing so, because they feel this gives them a competitive advantage (T=0, H=7). For example, H15W says he avoids fully automated tools because "I assume that [large companies] already run static and dynamic analysis tools... so there's not much of a point of me doing it."

Vulnerability recognition. Participants iterate through the prior three phases until they eventually identify a vulnerability. This phase can be as simple as seeing a crash that produces a known bad behavior or getting a tool output that indicates a problem. However, in most cases our participants described relying on their intuition and system knowledge to recognize where an assumption is violated or a simple crash shows a bigger security problem (T=6, H=14).

Reporting. Finally, once the vulnerability is found, it must be reported. In their reports, our participants focus on making sure the information is presented in a way that is easily understandable by developers (T=8, H=11). Specifically, they stressed communicating the importance of fixing the vulnerability (T=10, H=12). T4G explained that even after you find a vulnerability, "you have to have the weight of someone else to agree that [it] is a bug... you do have to [convince] someone that there's a risk... It's quite timely [time consuming], running a ticket." This emphasis

on selling the importance of the vulnerability mirrors the findings of Haney and Lutters [172].

Exception to the process. Four hackers in our study reported a notable exception to the order of phases described above. These hackers stated that they, in some cases, first recognize a vulnerability and reverse the normal process by looking for an execution path to the insecure code (T=0, H=4). This occurs whenever they find known insecure code (e.g., *memcpy* or *printf* in a C program) using a string search or other simple static analysis. Then they trace the execution path back through the code manually to find any input that triggers the vulnerable code. While this is a different order of operations, the general phases of the process remain the same.

4.4 Influencing factors

While all our participants described a similar process, their implementation of this process differed. These differences can be loosely grouped into four influencing factors: *Vulnerability Discovery Experience*, *Underlying System Knowledge*, *Access to the Development Process*, and *Motivation*. We found that both groups of practitioners expect increases in Vulnerability Discovery Experience and Underlying System Knowledge to improve vulnerability discovery success. Further, we found that hackers and testers reported similar levels of underlying system knowledge, yet the most important difference between our hackers and testers was in their vulnerability discovery experience. To our surprise, we did not find a straightforward

relationship between increased access to the development process and successful vulnerability finding. Finally, the impact of different motivational influencing factors was likewise more complex than expected.

4.4.1 Vulnerability discovery experience

Overall, hackers and testers agreed that prior experience finding vulnerabilities significantly improves their vulnerability discovery process (T=10, H=13); the key difference is that hackers reported notably more experience than testers.

In particular, we find that regardless of role, experience improves a practitioner's ability to efficiently identify the attack surface, select test cases, recognize vulnerabilities, and sell the resulting report. Both groups reported that the best approaches to gaining the relevant experience are real-world code analysis, hacking exercises, learning from their community, and prior bug reports. However, hackers were more likely than testers to rely on hacking exercises and bug reports. Further, hackers are exposed to a wider variety of vulnerabilities across all these learning approaches. Figure 4.2 shows the effect of vulnerability discovery experience on phases of the process and the ways practitioners develop experience.

4.4.1.1 How does experience affect the process?

Across both groups of practitioners, participants identified several key ways that experience adds to vulnerability discovery success.

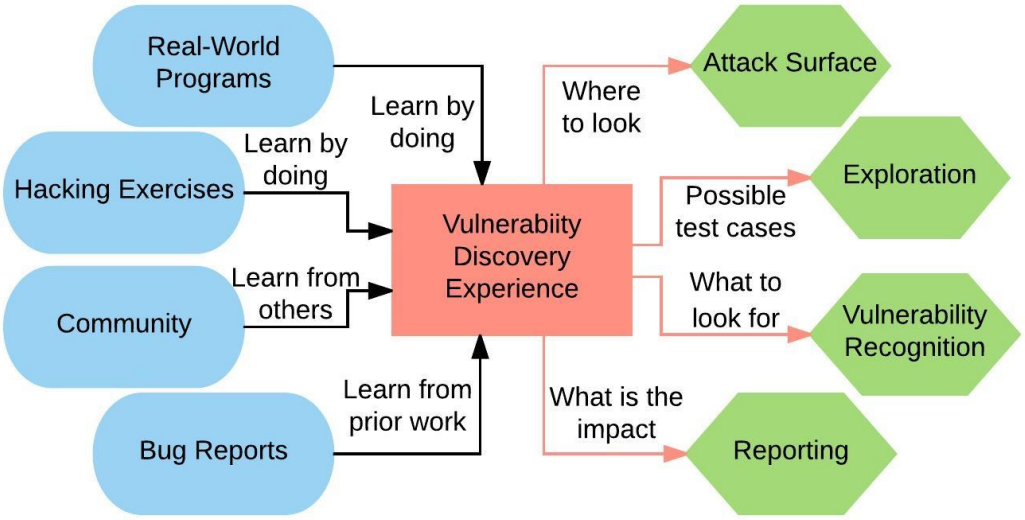

Figure 4.2: Vulnerability Discovery Experience Category Graph.

Helps recognize a problem quickly. Our participants frequently mentioned learning to recognize patterns that indicate a problem based on prior experience (T=6, H=14). For example, as he explores a program for possible bugs, H10H stated that he has a set of potential problems that he knows could occur based on prior experience. He said "I know that if there's a loop [that's] going through my input, it could be going out of bounds on the array, could be underflow, overflow." Relatedly, most participants discussed maintaining a mental or physical list of all the vulnerabilities they have seen through past experience and checking for these whenever they test a new piece of software (T=9, H=11).

Informs test case selection. In complex real-world systems, it is impractical to perform a complete search of all possible program inputs, so our participants stated that they rely on their intuition, learned from prior experience, to triage (T=9,

H=8). For example, T3W discussed creating a list of standard test cases "based on things we've found from Rapid7 [web security scanning tool]" or after asking "one of the developers... if there is other security testing we should be doing." From her experience, she "broadened the scope of security testing at the time [was just SQL injection], and brought in cross-site scripting." H2H explained how he built a set of file formats that he tries to open "in some random image parser, and half the time it would [cause a crash]." He said that he created his list based on his experience working with other security professionals in an "apprentice"-like situation where "You watch them, they watch you, and soon you're doing it on your own."

Helps identify the attack surface. We observed that only participants with more experience mentioned indirect inputs as part of the attack surface (T=4, H=10). Indirect inputs are more difficult to identify because they require a complex combination of events that may not occur frequently. Typically, our practitioners suggested that they only know to look for these complex interactions because they have seen something similar previously. T3W discussed learning about indirect inputs after incidentally finding a vulnerability in the way a program accepted user input from a LinkedIn third-party login service, "as soon as I found the LinkedIn problem, I made sure to test [FB and Twitter] to make sure [they were processed correctly]. And if we did allow login with another 3rd party in the future, I would check that too."

Helps describe a vulnerability's impact. Testers and hackers leverage prior

experience to explain how a vulnerability could be used by a malicious actor when arguing its impact to developers. T10W recalled a time where he used the story of a denial of service attack, caused by the same type of vulnerability, to explain the importance of fixing a new problem quickly.

Without experience, slower and more random. Without prior experience guiding triage, our practitioners relied on stumbling across vulnerabilities incidentally (T=5, H=4); or on their curiosity (T=8, H=5), personal creativity (T=3, H=6), and persistence (T=2, H=9) with ample time (T=4, H=9) to dig through the complexity of a program. Such incidental discovery is time consuming and haphazard, with little consistency in results. H1H described looking for a vulnerability in a complex program and "spent the whole summer on it and failed", but after reviewing bug reports for similar programs, he returned to searching the same program and found a vulnerability after "about a month". Thus, prior experience provides a useful and, in the opinion of some of our hackers, critical stimulus to the bug finding process (T=0, H=4).

4.4.1.2 How is experience developed?

Our participants developed experience through hands-on practice, supplemented by support from their peers and by reading other practitioners' vulnerability reports. Most notably, hackers reported a greater variety of learning methods, and more diverse experiences within each method, than testers; as a result, hackers

developed more, and more valuable, experience.

Gained by searching real-world programs. All of our testers mentioned gaining experience through their job, supporting findings from Lethbridge et al [244]. Six reported gaining vulnerability-discovery experience by incidentally finding security vulnerabilities while seeking out functionality bugs. Four reported learning something from their company's security-training best practices, but also reported that these best practice guides provide, at best, limited information.

The hackers in our study also develop hands-on experience through employment, which tended to be in security-specific roles such as full-time bug bounty participation and contracted penetration testing (H=13). As might be expected, this security-focused experience provides a strong advantage. Additionally, the ad-hoc and frequently changing nature of hackers' employment exposes them to a wider variety of programs, and therefore types of vulnerabilities, compared to testers who focus only on a single program or a few programs developed by their company and change focus less frequently.

In addition to their full-time jobs, we found that many of our hackers and some testers performed vulnerability discovery on real-world programs as a hobby (T=3, H=11). These participants explained that they searched for vulnerabilities, though there was no expected economic benefit, for the purpose of hands-on learning and personal enjoyment.

Gained through hacking exercises. Many of our hackers and some of our

testers participate in hacking exercises like capture-the-flag competitions or online war games [96, 295] (T=4, H=13). These exercises expose players to a variety of vulnerabilities in a controlled, security-specific setting with little program functionality aside from vulnerable components. H3G explained that hacking exercises help players focus on important details without becoming "overloaded"; these exercises also offer a "way of measuring the progress of your skills." Notably, the four testers had participated in only a few narrowly-focused workplace competitions, while our hackers mentioned many broad-ranging exercises.

Learned from their community. Both hackers and testers reported similar experiences learning through colleagues, both within and external to their workplace. Participants mention learning from co-workers (T=7, H=7); from hobbyist (T=7, H=10) and professional (T=2, H=0) organizations in which they are a member; and from informal personal contacts (T=6, H=12). Within these communities, practitioners are taught by those with more experience (T=10, H=13) and learn by working through and discussing difficult problems with their peers (T=6, H=9). For example, T5W described "Just watching other people test, grabbing what one person uses and then another and adding it to your own handbook." H2H explained that starting his career at a security company with "a lot of institutional knowledge" was critical to his development because he had "a lot of folks that I was able to pick their brain." Whenever personal contacts are not sufficient, practitioners also seek out information published online, typically in the form of expert blog articles and web forum posts (T=6, H=10).

Learned from prior vulnerability reports. Additionally, many participants—but particularly hackers—regularly read other individuals' vulnerability reports or discussed vulnerabilities found by colleagues to learn about new vulnerability types and discovery techniques, essentially gaining practical experience vicariously (T=6, H=15). H1H described using bug reports to test his vulnerability finding skills. Before reading a report, he asks himself, "Can I see the bug?" in the vulnerable version of the program. If the answer is no, he looks at the report to see "what the issue was and what the fix was and then where in the source the bug was." However, testers commonly only look at internal reports (T=5, H=0), whereas hackers view reports from a variety of programs (T=1, H=15), exposing them to a wider range of experiences.

Rarely learned through formal education. Finally, some participants mentioned more formal training such as books (T=1, H=7), academic courses (T=2, H=6), and certifications (T=0, H=1). In all cases, however, these methods were perceived to only support attaining the skills to participate in hands-on methods, not to be sufficient on their own.

4.4.2 Underlying system knowledge

Almost all participants in both populations (T=8, H=15) emphasized the importance of underlying system knowledge (e.g., operating systems, programming languages, network protocols, and software libraries) for successful vulnerability

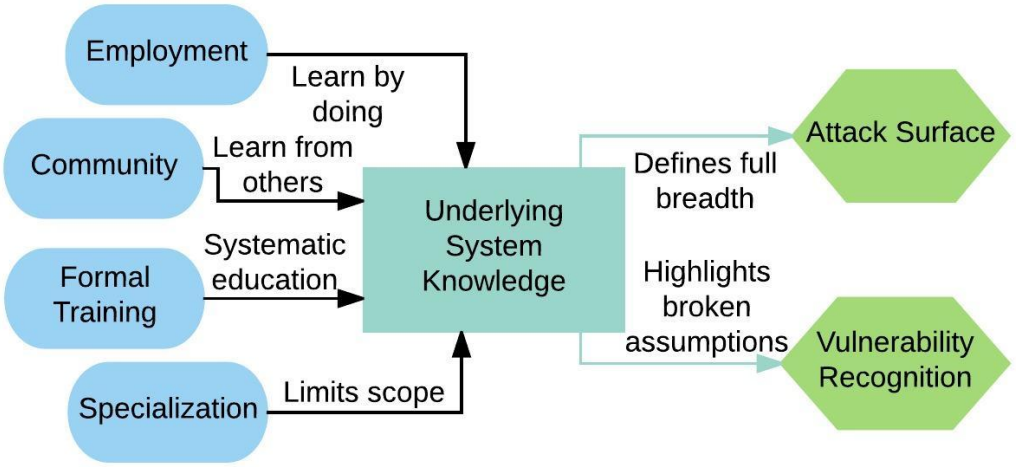

Figure 4.3: Underlying System Knowledge Category Graph.

discovery. Unlike with vulnerability discovery experience, our hackers and testers expressed similar levels of system knowledge. Instead, the biggest variation is in which underlying systems participants understand, primarily due to software specialization. Many participants reported focusing on a particular type of software (e.g., web, mobile, host) out of necessity, such as limited time to maintain proficiency in all software types (T=5, H=11). We found that practitioners in both populations limit their vulnerability searches based on their specialty (e.g., mobile specialists only consider a mobile app and not its associated web server).

Both populations indicated that system knowledge plays a role in the Attack Surface phase; hackers were more likely to report that it also plays a role in the Vulnerability Recognition phase (See Figure 4.3.)

4.4.2.1 How does system knowledge affect the process?

Understanding the underlying system components allows practitioners to recognize vulnerabilities caused by discrepancies between the developer's assumptions about how the system behaves and what actually occurs (T=2, H=6). H14W described how his understanding of Mozilla web add-ons helps him recognize vulnerabilities in other developers' code, saying that add-on developers "have no idea what they are doing there, and I see they do horrible stuff."

Strong system knowledge helps practitioners identify more input vectors into a program, as well as the full range of potential inputs (T=5, H=12). H1H gave an example of better system understanding improving his view of the attack surface: "I took [Operating Systems] where I was writing a kernel, and that was incredibly important. It wasn't until I took this that I really understood what the attack surfaces really were... The idea of being the [Virtual Machine] host where you're communicating with the GPU via some channel, I wouldn't have thought about that layer if I hadn't written a kernel."

4.4.2.2 How is system knowledge developed?

The development of system knowledge closely parallels the development of vulnerability discovery experience, with participants relying on hands-on experience and the community. Participants indicated learning through a mixture of on-the-job learning as a tester or hacker (T=10, H=13) and experience as a developer (T=6,

H=11) or systems administrator (T=0, H=3). H5M discussed the impact of his prior employment; "I worked at an antivirus company and you had to look at a lot of samples really quick... and [now] it's easy to say 'Ok, here's where they're doing this'... and just quickly looking at it."

Participants also mentioned using community (T=7, H=7) and online resources (i.e., expert blogs and web forums) (T=7, H=8) to supplement their experiential learning. Participants also learn from standards documents such as network protocol RFCs and assembly language instruction set documentation (T=2, H=4). Again, very few mentioned formal education (T=2, H=2), and of those who did, none considered it necessary. H6G explained that he has read some good books and has a computer science degree, but finds hands-on learning the best because "For me I need ten hours reading a book. It's the same as 2 [or] 3 hours trying to solve a challenge."

4.4.3 Access to development process

Another factor that influences the vulnerability discovery process is whether a practitioner has access to the development process. Figure 4.4 shows the effect of this access on the phases of vulnerability discovery. Clearly, because testers serve in an internal role, they have greater access to the source code, program requirements, and the developers themselves; they are also commonly involved in program-design decisions (T=7, H=0). All of our hackers, conversely, are (intentionally) outsiders (H=15) approaching the program as a black box with access at most to the source

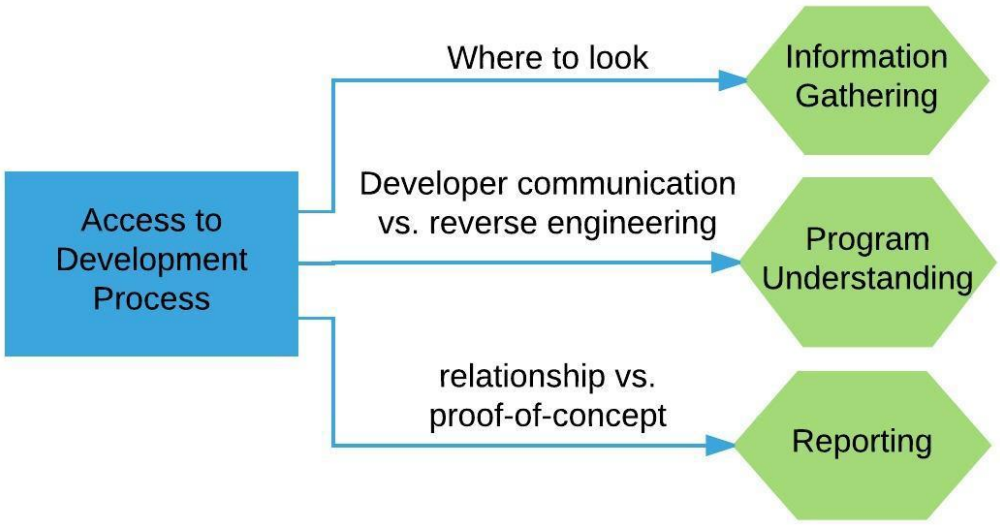

Figure 4.4: Access to development process category graph.

code if it is an open-source project or unobfuscated client-side script. Our participants report that both perspectives have key advantages and disadvantages: as outsiders by design, hackers are not biased by the assumptions of the developers, but testers have potentially valuable inside knowledge as well as an advantage in communicating findings to developers.

Internal efforts rely on documentation and direct developer communication. When gathering information, most testers rely on internal code tracking databases and communication with developers and other testers to determine the results of prior vulnerability discovery efforts (T=9, H=0). When trying to understand the program, because testers are involved in the program's design, they get into the mindset of developers by talking to them (T=8, H=0). T6W described participating with developers and other stakeholders in "a design session. That's

where we are going to put down the requirements." This session includes discussions about how to build the system and "what kind of testing we are going to [do]."

Having internal access to the development process can reveal flawed assumptions that would never be found by an outsider. However, knowing too much about the program going into the vulnerability search can blind the investigator to certain issues. T4G explains this, saying, "I try to learn as much about it without knowing too much about it. ... It's hard to ignore certain details once you know about certain areas already." However, T4G still recognized the value of communicating with his developers, saying, "You can give feedback to your teammates, your developers, your product owners. ... You're coming back with information, and then they react on it. Then you have to go back there [to explore] again."

External efforts use black-box testing and reverse engineering techniques. Because hackers do not have access to internal resources, they have to use more complicated methods to directly interrogate the system. When initially gathering information about a program, hackers use network-scanning tools and other black-box enumeration techniques to determine how the program is built and what underlying technologies it uses (T=0, H=5). During the program understanding phase, hackers rely only on their ability to reverse engineer the developer's intentions by reading and executing the code in lieu of talking to developers (T=0, H=15). H9G builds a clear picture of how the developer is thinking and what they are trying to do by looking directly at their code, because "when you look at the binary... you get a more intimate look into how the programmer was thinking." He reads the code to

"see certain implementations and certain patterns in the code... that can potentially allow you to make an assumption about a part of the specification."

Building rapport with developers. In contrast to the mixed effect on the vulnerability search, our participants indicated that having greater access to the development process provides an advantage when reporting the vulnerability. Our testers discussed using this connection to develop a shared language about the program (T=8, H=0) and build a relationship where they can go to the developers directly to discuss the issue and mitigate the problem (T=9, H=0). T1W stated that he tries "to use the same verbiage, so if for example I'm testing an application and I'm referencing certain parts, ... [I'll] see how they name those specific fields... and I'll try to use the terms they're using versus regular colloquial terms." He explained that the shared language and relationship allows him to avoid misunderstandings that could slow or even stop the remediation process.

Our hackers rarely have the same rapport because, as external participants, they communicate with developers only when they find a vulnerability, which may only occur once per program. In a few cases, our hackers were able to develop a strong relationship with a particular company (H=2), but this only occurred after they submitted multiple reports to that company. H8M focuses on a very specific program type, mobile device firmware, and therefore has developed a relationship with most of the major companies in this area. He described adjusting the information he reports depending on previous interactions. For less security-proficient companies he needs "to go into full details, as well as sending a fully compiled,

weaponized exploit," but for companies he has a better relationship with, he just says "In this application in this class, you don't handle this right," and they can identify and fix the issue quickly. This avoids wasting his time creating a lengthy report or developers' time reading it.

Hackers make up for lack of access with proofs-of-concept. Because most hackers have minimal communication with developers, they stressed the necessity of proving the existence and importance of the vulnerability with a *proof-of-concept* exploit to avoid spending significant amounts of time explaining the problem. H3G explained that "including the proof-of-concept takes more time to develop, but it saves a lot of time and communication with the [developers], because you show that you can do an arbitrary [code execution]... and that this theoretical vulnerability cannot be mitigated." While this approach is straightforward, developing an exploit can be the most time-consuming part of the process (H=2), and developers may not accept a report even in the face of evidence (T=7, H=9) or appropriately fix the code because they do not understand the root of the problem (T=7, H=9). H15W gave an example of a time when he was reviewing a bug report and found that "they didn't fix it properly. [It was] still exploitable in other ways." Testers overcome these challenges by spending time in discussion with the developers to clear up misunderstandings, but hackers typically do not have these necessary relationships and access to developers.

4.4.4 Motivation

The final influencing factor on the discovery process is a practitioner's motivation for looking for vulnerabilities. Figure 4.5 illustrates how motivations affect the discovery process. Most of our participants select which programs to search and what parts of the code to look at based on a calculation of likelihood to find vulnerabilities versus the value of the vulnerabilities found (T=10, H=11). The four hackers who did not describe this likelihood versus value calculation still consider likelihood as a factor (T=10, H=15), but either are paid a fixed rate as part of an internal security team or contracted review or are motivated by some non-monetary benefit (see Section 4.4.4). Overall, our hackers and testers estimate vulnerability likelihood similarly, but differ significantly when determining value. Additionally, we found that all participants were motivated to report clearly, no matter their likelihood vs. value calculation.

Estimating likelihood. Because most modern code bases are very large, both populations expressed the need to triage which program components to search based on the likelihood of finding a vulnerability. Both populations described several similar heuristics for this. First, practitioners focus on code segments that they expect were not heavily tested previously (T=5, H=11). H7W, for example, considers where developers are "not paying attention to it [security] as much."

Next, testers and hackers look at parts of the code where multiple bugs were previously reported (T=3, H=9). As T2W said, "There were issues with those

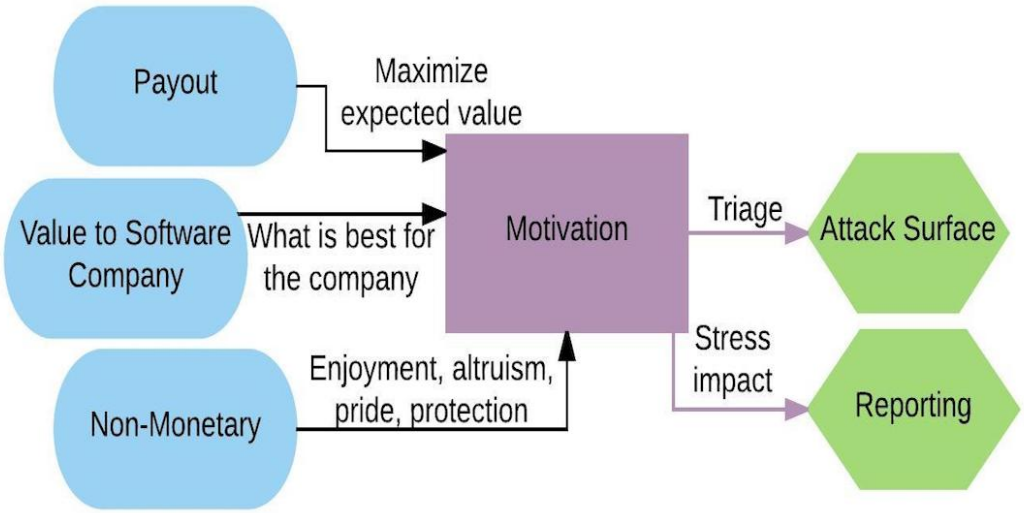

Figure 4.5: Motivation category graph.

areas anyway. . . so I figured that that was probably where there was most likely to be security issues. . . bugs cluster."

Both populations mentioned situations when code is new (e.g., rushed to release to fix a major feature issue) (T=5, H=5), or when they do not think the developers understand the underlying systems they are using (e.g., they noticed an odd implementation of a standard feature) (T=1, H=3). Additionally, some hackers also looked at old code (e.g., developed prior to the company performing stringent security checks) (T=0, H=7) and features that are rarely used (T=0, H=3).

Testers determine value by impact to company. As we would expect, testers determine value by estimating the negative effect to the company if exploited (T=8, H=3) or if the program fails a mandated audit (e.g., HIPAA, FERPA) (T=4, H=0). Because of this motivation, they tend to focus on features that are most commonly

used by their user base (T=2) and areas of the code that handle sensitive data (e.g., passwords, financial data) (T=8). T5W said he considers "usage of the site, [that is] how many people are going to be on a certain page or certain area of the site, [and] what's on the page itself, [such as] forms" to determine where a successful attack would have the most impact.

Hackers maximize expected payout using several strategies. Previous research has shown that hackers are more likely to participate in a program whenever the bounties are higher [438], and bounty prices increase with vulnerability severity [127]. We also observed that hackers cite the size of the bounty payout as their key motivator; however, we found that hackers follow one of two strategies when deciding how to best maximize their collective payouts.

The first strategy seeks out programs where the hacker has a competitive advantage based on specialized knowledge or experience that makes it unlikely that others will find other similar vulnerabilities (H=9). Hackers following this strategy participate in bug bounties even if they are unlikely to receive immediate payouts, because they can gain experience that will help them later find higher-payout vulnerabilities. H1H said that he focuses on more complex problems even though "I had no success for the first year, I knew that the barrier to entry was so high, that once I got good enough, then it would work out consistently that I could find bugs and get rewards. . . once you get good at it there's less competition."

The other payout maximizing strategy we observed is to primarily look for simple vulnerabilities in programs that have only recently started a bug bounty program

(H=8). In this strategy, hackers race to find as many low-payout vulnerabilities as possible as soon as a program is made public. Hackers dedicate little time to each program to avoid the risk of report collisions and switch to new projects quickly. H12W said that he switches projects frequently, just looking for "low-hanging fruit," because "somebody else could get there before you, while you are still hitting your head on the wall on this old client." This aligns with the phenomenon observed by Maillart et al., where hackers switch quickly to new bug bounties because they are more likely to have more vulnerabilities [258]. We found that hackers typically consider this approach when searching for web vulnerabilities, which have a "lower barrier to entry" than finding vulnerabilities in host software for which "the process to become proficient is higher [harder]" (H1H).

Additionally, some hackers completely avoid any company they have previously had poor relations with, either because they do not think it is likely they will be compensated fairly for their efforts or because the payment is not worth the administrative struggle (T=0, H=6). H9G described submitting a remote-code-execution vulnerability, but never receiving a response, when it should have garnered a large bounty based on the company's published policy. He said that "When we encounter that hostile behavior, that's pretty much an instant turn-off" from working with that company again.

Some participants also consider non-monetary value. Specifically, participants cited motivations including altruism (i.e., bounty paid to charity or improved security for the greater good) (T=2, H=7), enjoyment (T=1, H=11), peer pressure

(T=0, H=1), and personal protection (i.e., fix security bugs in products they use to avoid personal exploitation) (T=0, H=2). However, these factors are commonly secondary to monetary value.

All practitioners are motivated to report well. Practitioners' motivations also influence how they communicate with developers when reporting. Both populations expressed the need to make developers aware of the importance of fixing these bugs (T=10, H=12). Testers are only able to prevent harm to the company if developers accept and adequately fix the vulnerabilities they report. Hackers, motivated by a bug bounty payout, receive their payment only when the company accepts their report and are only paid at the level they expect if the developers agree with their assessment of severity. Participants defined importance as a function of the business impact on the company (T=8, H=3), how much control the vulnerability gives the attacker (e.g., limited data leakage vs. arbitrary code execution) (T=3, H=6), and how easily an attacker can exploit the vulnerability (T=2, H=6). T10W said ,"You need to be able to express not only what the problem is and where the problem lies, but also how this could be used to do X amount of damage." Additionally, some hackers discussed spending time after finding a vulnerability to understand the full implications of the issue (T=0, H=4). H9G said "When I find an issue, I don't necessarily rush to the developer... I could probably chain the vulnerability to other vulnerabilities to be more impactful and more impressive... [and] I get paid more, which is certainly a factor."

Our practitioners also emphasized the need to make their reports easy for

developers to understand by considering the technical and security background of their audience (T=7, H=11). As T2W stated, when "there's not enough experience with security across the [development] team, I tend to give them more information to make it easier." Some practitioners also use phrasing and wording that are easy to read (T=4, H=5). T1W said he checks to see if "I missed anything grammar-wise... does it have proper flow?" If he thinks it might be hard to read, he "pull[s] another tester and say[s], 'Hey, does this make sense?' " In some cases, practitioners use a fixed format (T=6, H=3) so that developers know where to look for specific information based on previous reports or by looking at the headings. Finally, many participants discussed maintaining an open-minded, respectful tone when discussing the vulnerability to avoid triggering a defensive response (T=8, H=5). T2W stressed the importance of respectful tone, saying, "Probably the biggest thing is keeping it factual and neutral. Some developers take any [report] as an attack on their ability to code."

4.5 Discussion and Recommendations

Our results show that two factors are necessary for success in vulnerability discovery: vulnerability discovery experience and underlying system knowledge. Further, we found that testers typically develop sufficient system knowledge through their employment and interactions with their community, but fall short in experience. Even though hackers and testers develop vulnerability discovery experience through similar means, hackers are exposed to a wider variety of programs and

vulnerabilities through the different types of employments, exercises, and communities they are involved in and the more diverse bug reports they read. Also, while not necessary to their success, participants' access to the development process and motivations have effects on the way they way they perform vulnerability discovery. Of note, we found that access to the development process facilitates reporting for testers as it helps build a rapport with developers and a shared language.

With these findings in mind, we suggest recommendations for organizations and individuals involved in software vulnerability discovery and directions for future work.

4.5.1 Training in the workplace

Our results suggest that extending testers' vulnerability discovery experience will improve their efficacy at finding vulnerabilities before release.

Security champions. Many of our testers described learning from more experienced testers (T=8). As a first change, we recommend hiring a small number (one or two) hackers to work alongside testers, highlighting potential vulnerabilities and sharing security-testing techniques. Deliberately introducing hackers to the team should cultivate learning opportunities. T8H discussed the success of this approach in her organization, saying, "I had two gentlemen... who were really into security testing. ... They eventually went on to create a whole new security team. ... Most of my security testing is all from what I've learned from them." T8H emphasized that this effort, which began with two testers experienced in security pointing out

problems to their less experienced co-workers, led within three years to development of a company-wide security consciousness. Further, she said that external security reviews of their product now find many fewer vulnerabilities than they did prior to introducing security champions.

Bug-report-based exercises. Many of our testers spend time discussing interesting bugs found by their peers in regular training sessions (T=7). However, simply discussing a vulnerability does not allow the hands-on practice our participants considered necessary. Instead, we suggest hands-on training based on vulnerabilities previously found in the company's code, either via formal exercises or simply by asking testers to search the pre-fix code and try to find the vulnerability (as suggested by H1H in Section 4.4.1.2. Such exercises will allow testers not only to learn about different vulnerabilities, but also to gain practical experience looking for them.

Future work to increase variety of experiences. The aforementioned approaches, however, will still only expose testers to a limited range of vulnerabilities within the program(s) on which they work. Further research is required to determine the best way to provide broader exposure to testers. Many of our testers participate in internal hacking exercises (T=6), but it was not clear why they do not participate in external exercises. Prior research has found that these exercises typically require a significant time commitment and prior knowledge, which we hypothesize are not a good fit for testers [80,396]. In Chapter 6, we consider this problem further through

a review currently available public hacking exercises.

Similarly, many testers cited internal bug reports as a learning source (T=6), but they do not spend time reading external reports like hackers do (T=1, H=13). One possible reason could be that it is difficult to find vulnerability reports without knowing the correct online resources to consult. Currently bug reports are dispersed among corporate vulnerability disclosure sites [54,164,385], personal GitHub repos [108], community mailing lists [141], and public vulnerability databases [298, 401]. Creating a single aggregated repository or searchable source for bug reports and discovery tutorials, and pointing testers to it, could expose testers to a wider range of information.

4.5.2 Improved hacker support

While improving testers' vulnerability-finding skills could meaningfully improve security, companies will likely still need security experts to find the most complex problems. In Chapter 5, we focus specifically on improving tool support for hackers to help them perform tasks more efficiently. Unfortunately, many of our hackers described difficulties communicating with developers after find a vulnerability, resulting in their findings either not being accepted or not being fixed properly (T=9). To solve this challenge, we look to learn from the strengths of our testers.

Establish consistent relationships early. We found that testers have an advantage in the reporting phase because they have built a relationship with developers through their inherent access to the development process. Additionally, the two

hackers who mentioned cultivating a close relationship with a particular company described similar benefits. We therefore recommend companies make efforts to build relationships with hackers as early as possible.

First, we recommend that organizations maintain a consistent point of contact with hackers, so that any time a hacker reports a vulnerability, they communicate with the same person and build a shared language, understanding, and trust. Obviously, a single point of contact is not always useful because a hacker may only report one vulnerability. In these cases, it is important for companies to be as open as possible when providing requirements and expectations to the hacker. Some potential improvements might be to provide more detailed templates and examples of effective reporting, to give feedback or ratings on (specific sections of) reports and how they did or didn't help developers, and answer hacker questions throughout the process to avoid confusion.

Further, our results support industry-wide standardization of vulnerability reporting procedures. This includes agreeing on report templates, "good" reporting examples, and vulnerability definitions. Standardizing expectations and vocabulary should provide consistency across programs and reduce the burden to build individual relationships with each company.

Hackers as security advocates. Additionally, further work is needed to understand how hackers can best convey the importance of a vulnerability, given limited communication channels and time to influence developer decisions. Future research should therefore focus on improving hacker reporting through improved resources

and training. For example, a centralized repository of real-world cases where an attacker has exploited a vulnerability that hackers can use as examples could help with demonstrating a vulnerability's importance. Relatedly, Haney and Lutter suggest providing hackers with formal training in how to best advocate for cybersecurity within complex organizational and structural environments [172].

Chapter 5: Observing Reverse Engineers' Processes

In the prior chapter, I showed that reverse engineering is the central task in software vulnerability discovery. Before being able to divine where a vulnerability might be, it is necessary to understand the functionality and behavior of the target program. (For brevity, we will refer to this task as RE and its practitioners as REs.) RE can be complex and time consuming, often requiring expert knowledge and extensive experience to be successful [137, 432]. In one study, participants analyzing small decompiled code snippets with less than 150 lines required 39 minutes on average to answer common malware-analysis questions [432].

Researchers, companies, and practitioners have developed an extensive array of tools to support RE [16, 25, 26, 61, 73, 106, 117, 133, 186, 187, 264, 349, 368, 378, 382, 383, 386, 404, 405, 432]. However, there is limited theoretical understanding of the RE process itself. While existing tools are quite useful, design decisions are currently ad-hoc and based on each designer's personal experience. With a more rigorous and structured theory of REs' processes, habits, and mental models, I believe existing tools could be refined, and even better tools could be developed. This follows from recommended design principles for tools supporting complex, exploratory tasks, in which the designer should "pursue the goal of having the computer vanish" [363, pg.

19-22].

In contrast to RE, there is significant theoretical understanding of more traditional program comprehension—how developers read and understand program functionality—including tasks such as program maintenance and debugging [21, 50, 102, 159, 228, 237, 246, 252, 316, 342, 409]. However, RE differs from these tasks, as REs typically do not have access to the original source, the developers who wrote the program, or internal documentation [116, pg. 141-196], [81]. Further, REs often must overcome countermeasures, such as symbol stripping, packing, obfuscation, and anti-debugging techniques [116, pg. 327-356], [301], [249, pg. 441-481], [175, pg. 660-661]. As a result, it is unclear which aspects of traditional program comprehension processes will translate to RE.

In this chapter [1], my colleagues and I develop a theoretical model of the RE process, with an eye toward building more intuitive RE tools. In particular, we set out to answer the following research questions:

RQ1. What high-level process do REs follow when examining a new program?

RQ2. What technical approaches (i.e., manual and automated analyses) do REs use?

RQ3. How does the RE process align with traditional program comprehension? How does it differ?

Specifically, when considering REs' processes, we sought to determine the types of questions they had to answer and hypotheses they generated; the specific steps

[1] published as [412]

taken to learn more about the program; and the way they make decisions throughout the process (e.g., which code segments to investigate or which analyses to use).

As there is limited prior work outlining REs' processes and no theoretical basis on which to build quantitative assessments, we chose an exploratory qualitative approach, building on prior work in expert decision-making [65, 222, 225] and program comprehension [21, 50, 102, 159, 228, 237, 246, 252, 316, 342, 409]. While a qualitative study cannot indicate prevalence or effectiveness of any particular process, it does allow us to enumerate the range of RE behaviors and investigate in depth their characteristics and interactions. Through this study, we can create a theoretical model of the RE process as a reference for future tool design.

To this end, we conducted a 16-participant, semi-structured observational study. During each session, we observed participants performing the RE task in a natural setting while recording their behaviors and probing their mental processes.

5.1 Background

While little work has investigated expert RE, there has been significant effort studying similar problems of naturalistic decision-making (NDM) and program comprehension. Because of their similarity, we draw on theory and methods that have been found useful in these areas [21, 50, 159, 223, 226, 237, 246, 342, 409] as well as in initial studies of RE [52].

5.1.1 Naturalistic Decision-Making

Significant prior work has investigated how experts make decisions in real-world (naturalistic) situations and the factors that influence them. Klein et al. proposed the theory of Recognition-Primed Decision-Making (RPDM) [223, pg. 15-33]. The RPDM model suggests experts recognize components of the current situation—in our case, the program under investigation—and quickly make judgments about the current situation based on experiences from prior, similar situations. Therefore, experts can quickly leverage prior experience to solve new but similar problems. Klein et al. have shown this decision-making model is used by firefighters [222, 225], military officers [65, 344], medical professionals [442, pg. 58-68], and software developers [224]. Votipka et al. found that vulnerability-discovery experts rely heavily on prior experience [413], suggesting that RPDM may be the decision-making model they use.

NDM research focuses on these decision-making processes and uses interview techniques designed to highlight critical decisions, namely the Critical Decision Method, which has participants walk through specific notable experiences while the interviewer records and asks probing follow-up question about items of interest to the research (see Section 5.2.1) [226]. Using this approach prior work has driven improvements in automation design. Specifically, these methods have identified tasks within expert processes for automation [221, 226], and inferred mental models used to support effective interaction design [333] in several domains, including automobile safety controls [299, 433], military decision support [10, 18, 226, 274],

and manufacturing [227, 300]. Building on its demonstrated success, we apply the Critical Decision Method to guide our investigation.

5.1.2 Program Comprehension

Program comprehension research investigates how developers maintain, modify, and debug unfamiliar code—similar problems to RE. Researchers have found that developers approach unfamiliar programs from a non-linear, fact-finding perspective [21, 50, 159, 237, 246, 342, 409]. They make hypotheses about program functionality and focus on proving or disproving their hypotheses.

Programmers' hypotheses are based on *beacons* recognized when scanning through the program. Beacons are common schemas or patterns, which inform how developers expect variables and program components to behave [21, 102, 228, 316]. To evaluate their hypotheses, developers either mentally simulate the program by reading it line by line, execute it using targeted test cases, or search for other beacons that contradict their hypotheses [21, 68, 102, 252, 342]. Von Mayrhauser and Lang showed developers switch among these methods regularly, depending on the program context or hypothesis [408]. Further, when reading code, developers focus on data- and control-flow dependencies to and from their beacons of interest [228, 238].

We anticipated that REs might exhibit similar behaviors, so we build on this prior work by focusing on hypotheses, beacons, and simulation methods during interviews (Section 5.2.1). However, we also hypothesized some process divergence, as RE and "standard" program comprehension differ in several key respects. Reverse

engineers generally operate on obfuscated code and raw binaries, which are harder to read than source code. Further, REs often focus on identifying and exploiting flaws in the program, instead of adding new functionality or fixing known errors.

5.2 Method

We are interested in developing a theoretical model of the RE process with respect to both overall strategy and specific techniques used. In particular, we focus on the three research questions given at the beginning of this chapter.

To answer these questions, we employ a semi-structured, observation-based interview protocol, designed to yield detailed insights into RE experts' processes. The full protocol is given in Appendix A.3. Interviews lasted 70 minutes on average. Audio and video were recorded during each interview. All interviews were led by the first author, who has six years of professional RE experience, allowing him to understand each RE's terminology and process, ask appropriate probing questions, and identify categories of similar actions for coding. Participants were provided a $40 gift card in appreciation of their time. Our study was reviewed and approved by the University of Maryland's Institutional Review Board. In this section, we describe our interview protocol and data analysis process, and we discuss limitations of our method.

5.2.1 Interview Protocol

We performed semi-structured, observational video-teleconference interviews. We implemented a modified version of the Critical Decision Method, which is intended to reveal expert knowledge by inquiring about specific cases of interest [226]. We asked participants to choose an interesting program they recently reverse engineered, and had them recall and demonstrate the process they used. Each observation was divided into the two parts: program background and RE process. Throughout, the interviewer noted and asked further questions about multiple items of interest.

Program background. We began by asking participants to describe the program they chose to reverse engineer. This included questions about the program's functionality and size, what tools (if any) they used, and whether they reverse engineered the program with others.

Reverse engineering process. Next, we asked participants about their program-specific RE goals, and then asked them to recreate their process while sharing their screen (RQ1)[2]. We chose to have participants demonstrate their process, asking them to open all tools they used and perform all original steps, so we could observe automatic and subconscious behaviors—common in expert tasks [15]—that might be missed if simply asked to recall their process. As the participant recreated their

[2]The only participant who did not share their screen did so because of technical difficulties that could not be resolved in a timely manner.

process, we asked several directed questions intended to probe their understanding while allowing them to delve into areas they felt were important. We encouraged participants to share their entire process, even if a particular speculative step did not end up supporting their final goal. For example, they may have decided to reverse a function that turned out to be a common library function already documented elsewhere, resulting in no new information gain.

Instead of asking participants to demonstrate a recent experience, we could have asked them to RE a program new to them. This could be more representative of the real-world experience of approaching a new program and might highlight additional subconscious or automatic behaviors. However, it would likely require a much longer, probably unreasonable period of observation. When asked how much time participants spent reverse engineering the programs demonstrated, answers ranged from several hours to weeks. Alternatively, we could have asked participants to RE a toy program. However, this approach restricts the results, both in depth of process and in terms of the program type(s) selected. Demonstration provides a reasonable compromise, and is a standard practice in NDM studies [226]. In practice, we believe the effect of demonstration was small, especially because the interviewer asked probing questions to reveal subconscious actions.

Items of interest. The second characteristic of the Critical Decision Method is that the interviewer asks follow-on questions about items of interest to the research. We selected our items of interest from those identified as important in prior NDM (*decision*) and program comprehension (*questions/hypotheses*, *beacons*, *simulation*

methods) literature—discussed in Sections 5.1.1 and 5.1.2, respectively. These items were chosen to identify specific approaches used (RQ2) and differences between RE and other program comprehension tasks (RQ3). Below, we provide a short description of each and a summary of follow-on questions asked:

• **Decisions.** These are moments where the RE decides between one or more actions. This can include deciding whether to delve deeper into a specific function or which simulation method to apply to validate a new hypothesis. For decision points, we asked participants to explain how they made the decision. For example, when deciding to analyze a function, the RE might consider what data flows into the function as arguments or what calls it.

• **Questions/Hypotheses.** These are questions that must be answered or conjectures about what the program does. Reverse engineers might form a hypothesis about the main purpose of a function, or whether a certain control flow is possible. Prior work has shown that hypotheses are central part to program comprehension [21, 68, 237, 342], so we expected hypothesis generation and testing to be central to RE. For hypotheses, we asked participants to explain why they think the hypothesis might be true and how they tested it. As an example, if a RE observes a call to strcpy, they might hypothesize that a buffer overflow is possible. To validate their hypothesis, they would check whether unbounded user input can reach this call.

• **Simulation methods.** Any process where a participant reads or runs the code to determine its function. We asked REs about any manual or automated simulation methods used: for example, using a debugger to determine the program's

memory state at a specific point. We wanted to know whether they employed any tools and if they were custom, open source, or purchased. Further, we asked them to evaluate any tools used, and to discuss their effectiveness for this particular task. Additionally, we asked participants why they used particular simulation methods, whether they typically did so, the method's inputs and outputs, and how they know when to switch methods.

• **Beacons.** These include patterns or tells that a RE recognizes, allowing them to quickly generate hypotheses about the program's functionality without reading line-by-line. For example, if a RE sees an API call to get a secure random number with several bit-shift operations, they may assume the associated function performs a cryptographic process. For beacons, we had REs explain why the beacon stood out and how they recognized it as that sort of beacon rather than some other pattern. The goal in inquiring into this phenomenon is to understand how REs perform pattern matching, and identify potentially common beacons of importance.

Additionally, we noted whenever participants referenced documentation or information sources external to the code — e.g., StackOverflow, RE blogs, API documentation — to answer a program functionality question. We asked whether they use that resource often, and why they selected that resource.

To make the interviews more fluid and less repetitive, we intentionally skipped questions that had already been answered in response to prior questions. To ensure consistency, all the interviews were conducted by the first author.

We conducted two pilot interviews prior to the main study. After the first pilot, we made adjustments to ensure appropriate terminology was used and improve

question flow. However, no changes were required after the second interview, so we included the second pilot interview in our main study data.

5.2.2 Data Analysis

We applied iterative open coding to identify interview themes [379, pg. 101-122]. After completing each interview, the audio was sent to an external transcription service. Another researcher and I first collaboratively coded three interviews — reviewing both the text and video — to create an initial codebook[3]. Then, we independently coded 13 interviews, comparing codes after every three interviews to determine inter-coder reliability. To measure inter-coder reliability, we used Krippendorff's Alpha (α), as it accounts for chance agreements [179].[4] After each round, we resolved any differences, updated the codebook as necessary, and re-coded previously coded interviews. We repeated this process four times until they achieved an α of 0.8, which is above the recommended level for exploratory studies [179, 253].

Next, we sought to develop our theoretical model by extracting themes from the coded data. First, we grouped identified codes into related categories. Specifically, we discovered three categories associated with the phases of analyses performed by REs (i.e., Overview, Sub-component Scanning, and Focused Experimentation). Then, we performed an axial coding to determine relationships between and within each phase and trends across the three phases [379, pg. 123-142]. From these phases and their connections, we derive a theory of REs' high-level processes and

[3]The final codebook can be found in Appendix B.4

[4]The ReCal2 software package was used to calculate Krippendorff's Alpha [138]

specific technical approaches. We also present a set of interaction-design guidelines for building analysis tools to best fit REs.

5.2.3 Limitations

There are a number of limitations innate to our methodology. First, participants likely do not recall all task details they are asked to relay. This is especially common for expert tasks [15]. We attempt to address this by using the CDM protocol, which has been used successfully in prior decision-making research on expert tasks [226]. Furthermore, we asked participants to recreate the RE task while the interviewer observed. This allowed the interviewer to probe subconscious actions that would likely have been skipped without observation.

Participants also may have skipped portions of their process to protect trade secrets; however, in practice we believe this did not impact our results. Multiple participants stated they could not demonstrate certain confidential steps, but the secret component was in the process's operationalization (e.g., the keyword list used or specific analysis heuristics). In all cases, participants still described their general process, which we were able to include in our analysis.

Finally, we focus on experienced REs to understand and model expert processes. Future work should consider newer REs to understand their struggles and support their development.

5.3 Recruitment and Participants

We recruited interview participants from online forums, vulnerability discovery organizations, and relevant conferences.

Online forums. We posted recruitment notices on a number of RE forums, including forums for popular RE tools such as IDAPro and BinaryNinja. We also posted ads on online communities like Reddit. Dietrich et al. showed online chatrooms and forums are useful for recruiting security professionals, since participants are reached in a more natural setting where they are more likely to be receptive [103].

Related organizations. We contacted the leadership of ranked CTF teams[5] and bug bounty-as-a-service companies asking them to share study details with their members. Our goal in partnering with these organizations was to gain credibility with members and avoid our messages dismissed as spam. Prior work found relative success with this strategy [413]. To lend further credibility, all emails were sent from an address associated with our institution, and detailed study information was hosted on a web domain owned by our institution.

Relevant conferences. Finally, we recruited at several conferences commonly attended by REs. We explained study details and participant requirements in person and distributed business cards with study information. Recruiting face-to-face allowed us to clearly explain the goal of the research and its potential benefits to the

[5]Found via `https://ctftime.org/`

RE community.

Participant screening. We asked respondents to our recruitment efforts to complete a short screening questionnaire. Our questionnaire[6] asked participants to self-report their level of RE expertise on a five-point Likert-scale from novice to expert; indicate their years of RE experience; and answer demographic questions. As our goal is to produce interaction guidelines to fit REs' processes, building on less experienced REs' approaches may not be beneficial. Therefore, we only selected participants who rated themselves at least a three on the Likert scale and had at least three years of RE experience.We contacted volunteers in groups of ten in random order, waiting one week for their response before moving to the next group. This process continued until we reached sufficient interview participation.

Participants. We conducted interviews between October 2018 and January 2019. We received 68 screening survey responses; 42 met our expertise criteria. Of these volunteers, 16 responded to randomly ordered scheduling requests and were interviewed. We stopped further recruitment after 16 interviews, when we reached *saturation*, meaning we no longer observed new themes emerging. This is the standard stopping criteria for a rigorous qualitative process [78, pg. 113-115]. Because our participant count is within the range recommended by best practice literature (12-20 participants), our results provide useful insights for later quantitative inquiry and generalizable recommendations [158].

[6]The screening full questionnaire can be found in an extended form of this paper at `https://ter.ps/REStudy2020`

Table 5.1 shows the type of program each participant reverse engineered during the interview and their demographics, including their self-reported skill level, years of experience, and the method used to recruit them. Each participants' ID indicates their assigned ID number and the primary type of RE tasks they perform. For example, P01M indicates the first interviewee is a malware analyst. Note that three interviewees used a challenge binary[7] during the interview. These participants could not show us any examples from their normal work due to the proprietary or confidential nature of their work. Instead, we asked them to discuss where their normal process on a larger program differed from process they showed with the challenge binary.

While we know of no good RE demographics surveys, our participant demographics are similar to bug-bounty hunters, who commonly perform RE tasks. Our population is mostly male (94%), young (63% $<$ 30) and well educated (75% with a bachelor's degree). HackerOne [163] and Bugcrowd report similar genders (91% of Bugcrowd hunters), ages (84% $<$ 35 and 77% $<$ 30, respectively), and education levels (68% and 63% with a bachelor's, respectively) for bug-bounty hunters.

5.4 Results: An RE Process Model

Across all participants, we observed at a high-level (RQ1) their RE process could be divided into three distinct phases: Overview, Sub-component scanning, and Focused experimentation. Beginning with a general goal—e.g., identifying vulnerabilities or malicious behaviors—REs seek a broad overview of the program's

[7] An exercise program designed to expose REs to interesting concepts in a simple setting

ID[1]	Program	Edu.	Skill[2]	Exp.	Recruitment
P01M	Malware	B.S.	4	7	Conference
P02V	System	HS	4	8	Conference
P03V	Challenge	M.S.	4	6	Conference
P04V	Challenge	B.S.	5	11	Conference
P05V	Application	M.S.	5	6	Forum
P06V	Challenge	HS	4	10	Forum
P07V	System	M.S.	5	10	Forum
P08V	Firmware	Assoc.	4	5	Forum
P09V	Firmware	B.S.	4	14	Forum
P10B	Malware	M.S.	5	15	Organization
P11M	Malware	Ph.D.	3	10	Forum
P12V	System	B.S.	3	8	Forum
P13V	Application	B.S.	5	21	Forum
P14M	Malware	M.S.	4	5	Forum
P15V	Application	HS	3	4	Forum
P16M	Malware	M.S.	3	3	Forum

[1] M: Malware analysis, V: Vulnerability discovery, B: Both

[2] Scale from 0-5, with 0 indicating no skill and 5 indicating an expert

Table 5.1: Participant demographics.

functionality (*overview*). They use this to establish initial hypotheses and questions which focus investigation on certain sub-components, in which they only review subsets of information (*sub-component scanning*). Their focused review produces more refined hypotheses and questions. Finally, they attempt to test these hypotheses and answer specific questions through execution or in-depth static analysis (*focused experimentation*). Their detailed analysis results are then fed back to the second phase for further investigation, iteratively refining questions and hypotheses until the overall goals are achieved. Each phase has its own set of questions, methods, and beacons that make up the technical approaches taken by REs (RQ2). In this section, we describe each phase in detail and highlight differences between RE and traditional program comprehension tasks (RQ3). In the next section, we discuss trends observed across these phases, including RE process components common to

multiple phases, such as factors driving their decision-making. Figure 5.1 provides an overview of each phase of analysis.

Note, in this section and the next, we give the number of REs who expressed each idea. We include counts to indicate prevalence, but a participant not expressing an idea may only mean they failed to state it, not that they disagree with it. Therefore, we do not perform comparisons between participants using statistical hypothesis tests. It is uncertain whether our results generalize past our sample, but they suggest future work and give novel insights into the human factors of RE.

Somewhat to our surprise, we generally observed the same process and methods used by REs performing both malware analysis and vulnerability discovery. In a sense, malware analysts are also seeking an exploit: a unique execution or code pattern that can be exploited as a signature or used to recover from an attack (e.g., ransomware). We did observe differences between groups, but only in their operationalization of the analysis process. For example, the two groups focused on different APIs and functionality (e.g., vulnerability finders looked at memory management functions and malware analysts focused on network calls). However, because our focus is on the high-level process and methods used, we discuss both groups together in the following sections.

5.4.1 Overview (RQ1)

Reverse engineers may have a short description of the program they are investigating (N=2), some familiarity with its user interface (N=2), or an intuition from

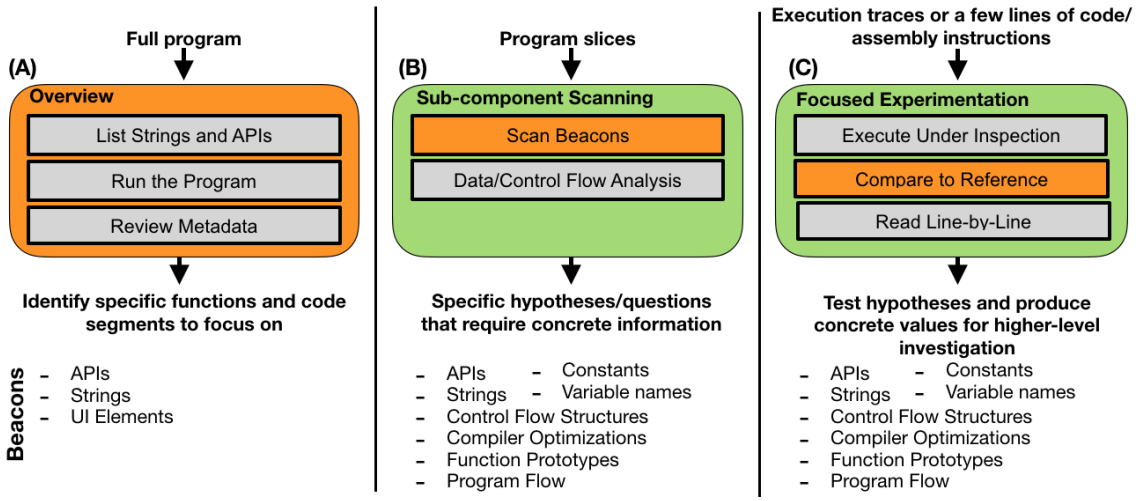

Figure 5.1: Overview of REs' three analysis phases. For each phase, the analyzed program scope is shown at the top, simulation methods used are in rectangles, and the analysis results are below the phase. Finally, the phase's beacons are at the bottom of the figure. Segments differing the most from the program comprehension literature are colored orange.

prior experience about the functions the program likely performs (N=7). However, they generally do not have prior knowledge about the program's organization or implementation (N=16). They might guess that the program performs cryptographic functions because it is a secure messaging app, but they do not know the algorithm or libraries used, or where in the code cryptographic protocols are implemented. Therefore, they start by seeking a high-level program view (N=16). This guides which parts of the program to prioritize for more complex investigation. P01M said this allows him to "get more to the core of what is going on with this binary." Reverse engineers approach this phase in several ways. The left section of Figure 5.1 summarizes the overview phase's simulation methods, beacons, and outputs. We discuss these items in more detail below.

Identify the strings and APIs used (RQ2). Most REs begin by listing the

strings and API calls used by the program (N=15). These lists allow them to quickly identify interesting components. P03V gave the example that "if this was a piece of malware... and I knew that it was opening up a file or a registry entry, I would go to imports and look for library calls that make sense. Like refile could be a good one. Then I would find where that is called to find where malicious behavior starts." In some cases, REs begin with specific functionality they expect the program to perform and search for related strings and APIs (N=7). As an example, P08V performed a "grep over the entire program looking for httpd because a lot of times these programs have a watchdog that includes a lot of additional configuration details."

Run the program and observe its behavior (RQ2). Many REs execute the program to see how it behaves under basic usage (N=7). When running the program, some REs look at UI elements (e.g., error messages), then search for them in the code, marking associated program components for further review (N=3). For example, P13V began by "starting the software and looking for what is being done." He was shown a pop-up that said he had limited features with the free version. He observed that there was "no place I can put a [access] code, so it must be making a web services check" to determine license status. Next, he opened the program in a disassembler and searched for the pop-up's text "because you expect there to be a check around where those strings are."

Review program metadata (RQ2). Some REs looked at information beyond the binary or execution trace, such as the file metadata (N=3), any additional

resources loaded (N=3) (e.g., images or additional binaries), function size (N=2), history of recent changes (N=1), where vulnerabilities were found previously (N=1), and security mitigations used (N=1) (e.g., DEP or ASLR). This information gives further insights into program functionality and can help REs know what not to look for. P04V said "I've been burned in the past. You kind of end up down a long rabbit hole that you have to step completely back from if you don't realize these things... For example, for PIE [Position Independent Executables] there has to be some sort of program relative read or write or some sort of address disclosure that allows me to defeat the randomization. So that's one thing to look for early on."

Malware analysts perform overview after unpacking (RQ2). Many malware binaries are stored in obfuscated form and only deobfuscated at execution time to complicate RE. This is commonly referred to as *packing*. Therefore, REs must first unpack the binary before strings and imported APIs become intelligible (N=2). However, once unpacking is performed and the binary is in a readable state, REs perform the same overview analyses described above (N=2).

Overview is unique to RE (RQ3). In most other program comprehension tasks, the area of code to focus on is known at the outset based on the error being debugged [436] or the functionality being modified or updated [228, 339]. Additionally, developers performing program comprehension tasks typically have access to additional resources, such as documentation and the original developers, to provide high-level understanding [343], making overview analyses unnecessary.

5.4.2 Sub-component Scanning (RQ1)

Based on findings from their overview, REs next shift their attention to program sub-components, searching for insights into the "how" of program functionality. By focusing on sub-components, sub-component scanning allows REs to quickly identify or rule out hypotheses and refine their view of the program. P08V explained that he scanned the code instead of reading line-by-line, saying, "I'm going through it at a high level, because it's really easy to get caught in the weeds when there could be something much better to look at." The middle column of Figure 5.1 gives an overview of this analysis phase.

Scan for many beacons (RQ2). Most commonly, REs scan through functions or code segments prioritized in the overview (N=15), looking for a variety of beacons indicating possible behaviors. These include APIs (N=15), strings (N=15), constants (N=11), and variable names (N=11). For example, while investigating a piece of malware, P02V saw GetProcAddress was called. This piqued his interest because "it's a very common function for obfuscation... it's likely setting up an alternate input table" to hide obviously malicious calls from an RE looking only at the standard import table.

REs infer program behaviors both from individual instances (N=16) and specific sequences (N=12) of these items. For example, while reverse engineering the code in Figure 5.2, P11M first scanned the strings on lines 44-46 and recognized them as well-known websites, generally reachable by any device connected to the

```
42 var zzo = function() {
43     var ttw = [
44         "http://www.microsoft.com/",
45         "http://www.google.com",
46         "http://www.bing.com"
47     ];
48     for (var i = 0, h, wep; i < ttw.length; i++){
49         try {
50             var h = new ActiveXObject("MSXML2.ServerXMLHTTP.6.0");
51             h.open("GET", ttw[i]);
52             h.setRequestHeader("User-Agent", _.u);
53             h.setRequestHeader("Cache-Control", "no-cache");
54             h.setRequestHeader("Pragma", "no-cach");
55             h.setRequestHeader("Connection", "close");
56             h.send("");
57             wep =
58                 new Date(
59                     h
60                         .getAllResponseHeaders()
61                         .split("Date: ")
62                         .pop()
63                         .split("\n")
64                         .shift()
65                     ).getTime() / 1000;
66                 if (1388534400 < wep) {
67                     return wep;
68                 }
69             } catch (e) {}
70         }
```

Figure 5.2: Screenshot of botnet code investigated by P11M, which performs a network connectivity check. This provides an example of API calls and strings recognized during sub-component scanning giving program functionality insights.

Internet. He then looked at the API calls and strings on lines 51-56 and said that "it's just trying to make a connection to each of those [websites]." By looking at the constant checked on line 66, he inferred that "if it's able to make a connection, it's going to return a non-zero value [at line 66]." Putting this all together and comparing to past experience, P11M explained, "usually you see this activity if something is trying to see if it has connectivity."

REs also make inferences from less obvious information. Many review control-flow structures (N=13) for common patterns. When studying a router's firmware,

P08V noticed an assembly code structure corresponding to a switch statement comparing a variable to several constants. From this, he assumed that it was a "comparison between the device's product ID and a number of different product IDs. And then it's returning different numbers based off that. So it looks like it's trying to ascertain what product it is and then doing something with it," because he has "seen similar behavior before where firmware is written in generically." Other REs consider the assembly instructions chosen by the compiler (N=8) or function prototypes (N=5) to determine the data types of variables. P02V explained, "It is very important to understand... how compilers map code to the actual binary output." As an example, he pointed out instructions at the start of a function and said, "that's just part of saving the values... I can safely skip those." Then he identified a series of registers and observed "those are the function's arguments... after checking the codebase of FreeBSD, I know the second argument is actually a packed structure of arguments passed from outside the kernel. This is [the data] we control in this function context." Finally, REs consider the code's relation to the overall program flow (N=6). For example, P08V identified a function as performing "tear down" procedures—cleaning up the state of the program before terminating—because it "happened after the main function."

Focused on specific data-flow and control-flow paths (RQ2). Some REs also scanned specific data- (N=8) and control-flow (N=7) paths, only considering instructions affecting these paths. These analyses were commonly used to understand how a function's input (N=7) or output (N=4) is used and whether a particular path

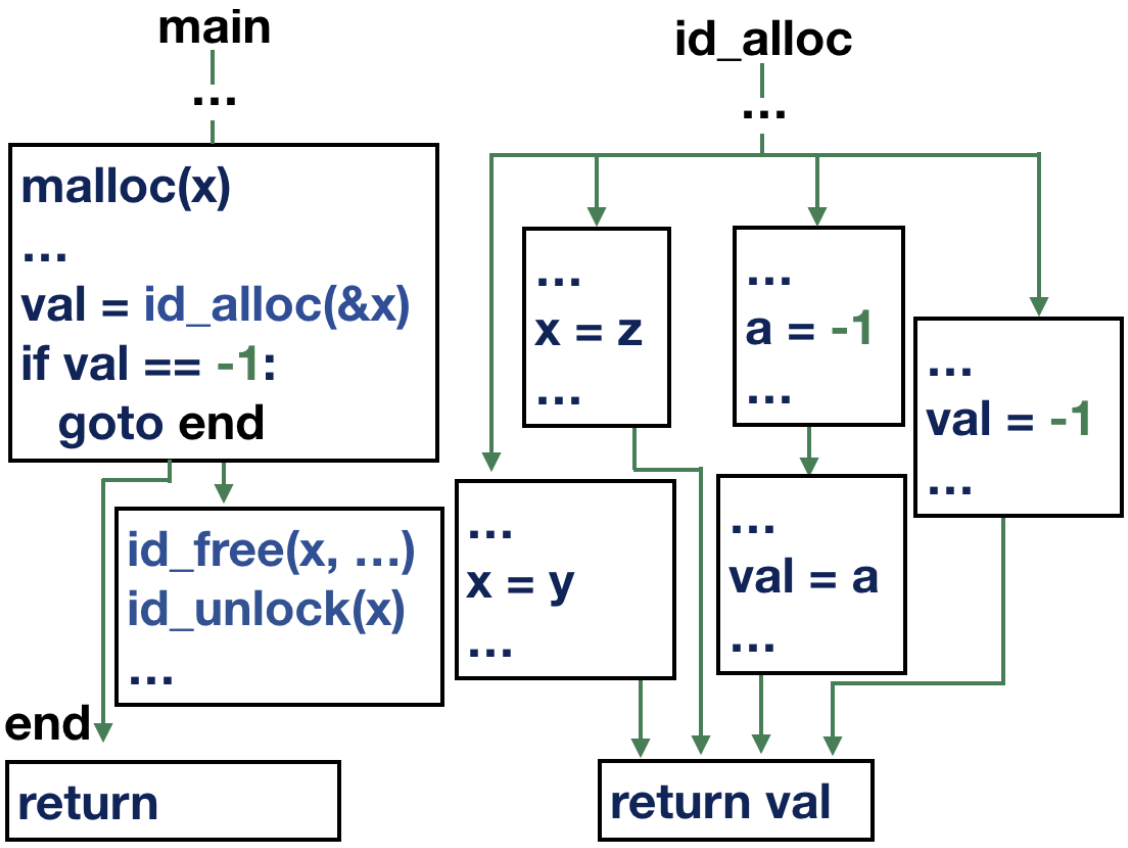

Figure 5.3: Program investigated by P02V to determine whether he could trigger an undefined memory read. The code has been converted to a pseudo-code representation including only relevant lines. It shows the control flow graph for two functions: main and id_alloc . Rectangles represent basic blocks, and arrows indicate possible control flow paths.

is realizable (N=4). For example, while reviewing the program summarized in Figure 5.3, P02V asked whether a control-flow path exists through id_alloc in which x is not written. Memory for x is allocated before the id_alloc call and read after, so if such a path is possible, "we can have it read from undefined memory." To answer this question, P02V scanned each control flow path through the function from the bottom of the graph up. If he saw a write to x, he moved on to the next path. This check invalidated the first two control-flow paths (counting left-to-right) in Figure 5.3. Additionally, in main, the program exits if the return value of id_alloc

is −1. Thus his next step was to check the data flow to id_alloc 's return value to see whether it was set to −1. He found the return value was set to −1 in both remaining control-flow paths, indicating it was not possible to read from undefined memory.

The diversity of beacons represents a second difference from program comprehension (RQ3). While program comprehension research has identified several similar beacons (API calls, strings, variable names, sequences of operations, and constants [21, 102, 228, 316]), developers have been shown to struggle when variable names and other semantic information are obfuscated [102]. However, REs adapt to the resource-starved environment and draw on additional beacons (i.e., control flow structures, compiler artifacts, and program flow).

5.4.3 Focused Experimentation (RQ1)

Finally, when REs identify a specific question or hypothesis, they shift to focused experimentation: setting up small experiments, varying program inputs and environmental conditions, and considering the program's behavior in these states to find a concrete answer or prove whether specific hypotheses hold. This phase's results are fed back into sub-component scanning, to refine high-level hypotheses and the RE's interpretation of observed beacons. Again, REs rely on a wide range of methods for this analysis.

Execute the program (RQ2). In most cases, REs validate their hypotheses by

running the code under specific conditions to observe whether the expected behavior occurs (N=13). They may try to determine what value a certain variable holds at a particular point (e.g., input to a function of interest) under varying conditions (N=13) or whether user input flows to an unsafe function (N=9). For example, after reviewing the data-flow path of the program's arguments, P03V hypothesized that the program required two input files with a specific string in the first line to allow execution to reach potentially vulnerable code. To test this hypothesis, she ran the program in a debugger with the expected input and traced execution to see the state of memory at the potentially vulnerable point.

While running the program, REs gather information in a variety of ways. Most execute the code in a debugger (N=12) to probe memory and have full control over execution. Some use other tools like packet capturers and file monitors to observe specific behaviors (N=8). In some cases, REs manipulate the execution environment by dynamically changing registry values (N=7) or patching the binary (N=5) to guide the program down a specific path. As an example, while analyzing malware that "checks for whether it is being run in a debugger," P16M simply changes the program "so that the check will always just return false [not run in debugger]."

Finally, some REs fuzz program inputs to identify mutation-specific behavior changes. In most cases, fuzzing is performed manually (N=6), where the RE hand-selects mutations. Automation is used in later stages, once a good understanding of the program is established (N=1). P08V explained, "I wait until I have a good feel for the inputs and know where to look, then I patch the program so that I can quickly pump fuzzed inputs from angr [366] into the parts I care about."

Compare to another implementation (RQ2). Some REs chose to re-write code segments in a high-level language based on the expected behavior (N=8) or searched for public implementations (e.g., libraries) of algorithms they believed programs used (N=5). They then compared the known implementation's outputs with the subject program's outputs to see if they matched. For example, once P10B recognized the encryption algorithm he was looking at was likely Blowfish, he downloaded an open-source Blowfish implementation. He first compared the open-source code's structure to the encryption function he was reviewing. He then ran the reference implementation and malware binary on a file of all zeros saying, "we can then verify on this sample data whether it's real Blowfish or if it's been modified."

Read line-by-line only for simple code or when execution is difficult (RQ2). Finally, REs resorted to reading the code line-by-line and mentally tracking the program state when other options became too costly (N=9). In some cases, this occurred when they were trying to answer a question that only required reading a few, simple lines of code. For example, P05V described a situation where he read line-by-line because he wanted to fully understand a small number of specific checks, saying, "After Google Project Zero identified some vulnerabilities in the system, the developers tried to lock down that interface by adding these checks. Basically I wanted to figure out a way to bypass these specific checks. At this point I ended up reading line-by-line and really trying to understand the exact nature of the checks." While no participants quantified the number of lines or code complexity they were

willing to read line-by-line, we did not observe any participants reading more than 50 lines of code. Further, this determination appeared goal- and participant-dependent, with wide variation between participants and even within individual participants' own processes, depending on the current experiment they were carrying out.

REs also chose to read line-by-line instead of running the program when running the program would require significant setup (e.g., when using an emulator to investigate uncommon firmware like home routers). P09V explained, "The reason I was so IDA [disassembler] heavy this time is because I can't run this binary. It's on a cheap camera and it's using a shared memory map. I mean, I could probably run this binary, but it's going to take a while to get [emulation] set up."

During this line-by-line execution, a few REs said they used symbolic execution to track inputs to a control flow conditional of interest (N=2). P03V explained, "I write out the conditions to see what possible states there are. I have all these variables with all these constraints through multiple functions, and I want to say for function X, which is maybe 10 deep in the program, what are the possible ranges for each of these variables?" In both cases, the REs said they generally performed this process manually, but used a tool, such as Z3, when the conditions became too complicated. As P03V put it, "It's easier if you can just do it in your brain of course, but sometimes you can't... if there are 10 possibilities or 100 possibilities, I'll stick it in a SAT solver if I really care about trying to get past a barrier [conditional]."

Beacons are still noticed and can provide shortcuts (RQ2). While REs focus on answering specific questions in this phase, some also notice beacons missed

in prior analyses. If inferences based on these beacons invalidated prior beliefs, REs quickly stop focused experimentation that becomes moot. For example, while P04V was reverse engineering a card-game challenge binary, he decided to investigate a reset function operating on an array he believed might be important. There were no obvious beacons on initial inspection and there were only a few instructions, so he decided to read line-by-line. However, he quickly recognized two constants that allowed him to infer functionality. He saw that "it's incrementing values from 0 to 51. So at this point, I'm thinking it's a deck of cards. And then it has this variable hold. Hold is a term for poker, and it sets 0 to 4." Once he realized what these variables were, he decided he had sufficient information to stop analyzing the function, and he moved back to the calling function to resume sub-component scanning.

Simulation methods mostly overlap with program comprehension (RQ3). Most of the methods described above, including using a debugger and reading code line-by-line, are found in the program comprehension literature. However, comparing program execution to another implementation appears unique to REs. As in sub-component scanning, this extra method is likely necessitated by the additional complexity inherent in an adversarial environment.

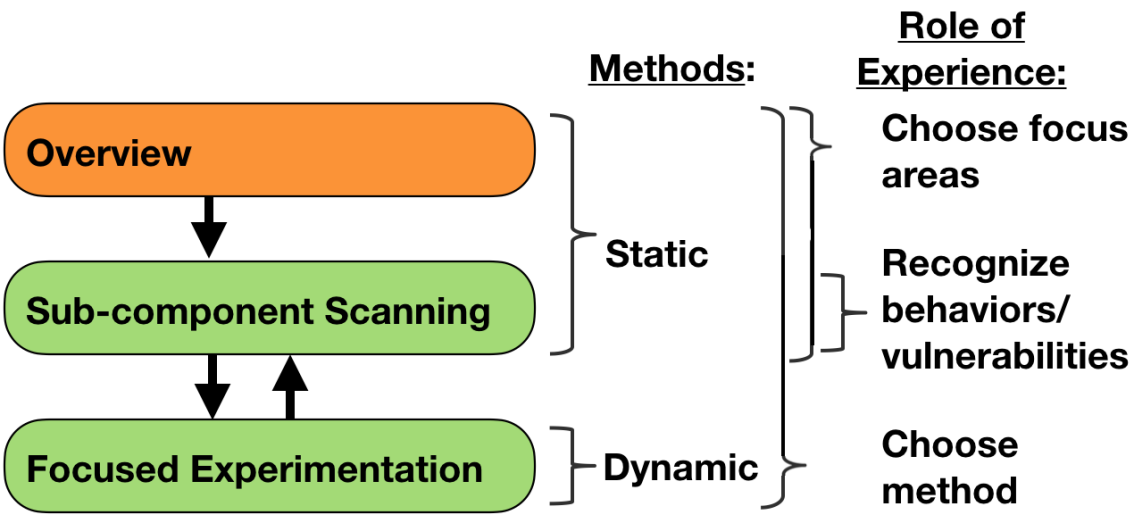

Figure 5.4: Overview of the analysis phases and trends observed across them. The arrows shown between the phases indicates information flow. The brackets indicate which phases the adjacent item is relevant to.

5.5 Results: Cross-phase Trends

In addition to the phases themselves, we observed several cross-phase trends in our participants' RE approaches, which we discuss in this section. This includes both answers to our research questions which were not unique to a specific phase and additional observations regarding tool usage which inform future tool development. Figure 5.4 includes some of these trends as they interact with the phases.

Begin with static methods and finish with dynamic (RQ2). Most of the simulation methods described in the first two analysis phases focused on static program representations, i.e., the binary or decompiled code. In contrast, focused experimentation was mainly performed dynamically, i.e., by running the program. Reverse engineers typically make this switch, as P05V stated, "because this thing is so complex, it's hard to trace the program flow [statically], but you can certainly

tell when you analyze an [execution] trace. You could say this was hit or this wasn't hit." However, REs sometimes choose not to switch when they perceive the switch to be difficult. P15V explained "[switching] was a little daunting to me. I just wanted to work in this environment I'd already set up."

Unfortunately, in most cases, switching contexts can be difficult because REs have to manually transfer information back and forth between static and dynamic tools (e.g., instructions or memory states) (N=14). To overcome this challenge, some REs opened both tools side-by-side to make comparisons easier (N=4). For example, P08V opened a debugger in a window next to a disassembler and proceeded to step through the main function in the debugger while following along in the assembly code. As he walked through the program, he regularly switched between the two. For example, he would scan the possible control-flow paths in the disassembler to decide which branch to force execution down and the necessary conditions would be set through the debugger. Whenever he came across a specific question that could not be answered just by scanning, he would switch to the debugger. Because he stepped through the program as he scanned, he could quickly list register values and relevant memory addresses to get concrete variable values.

Experience and strategy guide where to look in the first two phases (RQ1). Initially, REs have to make decisions about which metadata to look at, e.g., all strings and APIs or specific subsets, (N=4) and what inputs to provide to exercise basic behaviors (N=2). Once they run their overview analyses, they must determine which outputs (strings, APIs, or UI elements) are relevant to their

investigation (N=16) and in what order to process them (N=11). Reverse engineers first rely on prior experience to guide their actions (N=14). P04V explained that when he looks for iPhone app vulnerabilities, he has "a prioritized list of areas [APIs] I look at...it's not a huge list of things that can go horribly wrong from a security standpoint when you make an iPhone app...So, I just go through my list of APIs and make sure they're using them properly." If REs are unable to relate their current context to prior experience, then they fall back on basic strategies (N=16) such as looking at the largest functions first. P03V said, "If I have no clue what to start looking at...I literally go to the function list and say the larger function is probably interesting...as long as I can distinguish the actual code versus library code, this technique is actually pretty useful." Similarly, REs employ heuristics to decide which functions not to investigate. For example, P16M said, "If the function is cross-referenced 100 times, then I will avoid it. It's probably something like an error check the compiler added in."

In sub-component scanning, experience plays an even more important role. As in the previous analysis phase, REs must decide which data- (N=8) and control-flow paths (N=7) to consider. Again, this is done first by prior experience (N=6) and then by simple strategies (N=4). As they perform their analyses, REs must also determine potential hypotheses regarding program functionality (N=16) and possible vulnerabilities (N=9) — exploitable flaws in the case of vulnerability discovery, or signaturable behaviors for malware analysis. In most cases, these determinations are made by recognizing similarities with previous experiences (N=15). For example, when P08V saw a function named httpd_ipc_init , he recognized this might introduce

a vulnerability, saying, "IPC generally stands for inter-process communication, and many router firmwares like this set up multiple processes that communicate with each other. If it's doing IPC through message passing, then that opens up the attack surface to anything that can send messages to this httpd binary." If the RE is unable to generate hypotheses based on prior experience, they instead make determinations based on observed behaviors (N=16), obtained via more labor intensive investigation of the program execution or in-depth code review.

Experience used to select analysis method throughout (RQ1). There were typically multiple ways to answer a question. The most common example, as discussed in Section 5.4.3, was deciding between executing the program or reading line-by-line during focused experimentation (N=9). Similar decisions occurred in the other phases. For example, some REs choose to simply skip the overview phase all together and start with the main function (N=5) whenever, as P03V said, "it's clear where the actual behavior starts that matters."

REs also decide the granularity of analysis, weighing an approximation's benefits against the inaccuracy introduced (N=5). For example, several participants discussed choosing to use a decompiler to make the code easier to read, knowing that the decompilation process introduces inaccuracies in certain circumstances. P04V said, "I actually spend most of my time in Hex-Rays [decompiler]. A few of my friends generally argue that this is a mistake because Hex-Rays can be wrong, and disassembly can't be. And this is generally true, but Hex-Rays is only wrong in specific ways." Further, because these are explicit decisions, REs are also able

to recognize situations where the inaccuracies are common and can switch analysis granularities to verify results (N=5). For example, when using a decompiler, the RE has some intuition regarding what code should look like. P04V explained, "I've had many situations where I think this looks like an infinite loop, but it can't be. It's because Hex-Rays is buggy. Basically, in programming, no one does anything all that odd."

Preferred tools presented output in relation to the code (RQ2). In almost all cases, the tools REs choose to use provide a simple method to connect results back to specific lines of code (N=16). They choose to list strings and API calls in a disassembler (N=15), such as IDA, which shows references in the code with a few clicks, as opposed to using the command-line strings command (N=0). Similarly, those participants who discussed using advanced automated analyses, i.e., fuzzing (N=1) and symbolic execution (N=1), reported using them through disassembler plugins which overlaid analysis results on the code (e.g., code coverage highlighting for fuzzing). P03V used Z3 for symbolic execution independently of the code, supplying it with a list of possible states and manually interpreting its output with respect to the program. However, she explained this decision was made because she did not know a tool that presented results in the context of the code that could be used with the binary she was reversing. She said, "The best tool for this is PAGAI... If you have source it can give you ranges of variables at certain parts in a program, like on function loops and stuff." Specifically, PAGAI lets REs annotate source code to define variables of interest and then presents results in context of

these annotations [182].

Focused on improving readability (RQ2). Throughout, REs pay special attention to improving code readability by modifying it to include semantic information discovered during their investigation. In most cases, the main purpose of tools REs used was to improve code readability (N=9). Many REs used decompilers to convert the assembly code to a more readable high-level language (N=9), or tools like IDA's lumina server [184] to label well-known functions (N=2). Additionally, most REs performed several manual steps specifically to improve readability, such as renaming variables (N=14), taking notes (N=14), and reconstructing data structures (N=8). P01M explained the benefit of this approach when looking at a file reading function by saying, "It just says call DWORD 40F880, and I have no idea what that means... so, I'll just rename this to read file... [now I know] it's calling read file and not some random function that I have no idea what it is." Taking notes was also useful when several manipulations were performed on a variable. For example, to understand a series of complex variable manipulations, P05V said "I would type this out. A lot of times I could just imagine this in my head. I think usually I can hold in my head two operations...If it's anything greater than that I'll probably write it down."

Online resources queried to understand complex underlying systems (RQ2). Regarding external resources, REs most often reference system and API documentation (N=10). They reference this documentation to determine specific details

about assembly opcodes or API arguments and functionality. They also reference online articles (N=4) that provide in-depth breakdowns of complicated, but poorly documented system functions (e.g., memory management, networking, etc.). When those options fail, some REs also reference question-answering sites like StackOverflow (N=4) because "sometimes with esoteric opcodes or functions, you have to hope that someone's asked the question on StackOverflow because there's not really any good documentation" (P3). Many participants also google specific constants or strings they assume are unique to an algorithm (N=7). P10 explained, "For example, MD5 contains an initialization vector with a constant. You just google the constant and that tells you the algorithm."

5.6 Discussion

Our key finding is the identification and description of a three-phase RE process model, along with cross-phase trends in REs' behaviors. This both confirms and expands on prior work, which described an RE model of increasingly refined hypotheses [52]. We demonstrate a process of hypothesis generation and refinement through each phase, but also show the types of questions asked, hypotheses generated, actions taken, and decisions made at each step as the RE expands their program knowledge.

Our model highlights components of RE for tool designers to focus on and provides a language for description and comparison of RE tools. Building on this analysis model, we propose five guidelines for RE tool design. For each guideline,

	RE Tool Design Guidelines	Example Application
G1	**Match interaction with analysis phases** Reverse engineering tools should be designed to facilitate each analysis phase: overview, sub-component scanning, and focused experimentation.	**IDAPro [186], BinaryNinja [404], Radare2 [328]** Provide platforms for REs to combine analyses, but previously lacked thorough RE process model to guide analysis development and integration.
G2	**Present input and output in the context of code** Integrate analysis interaction into the disassembler or decompiled code view to support tool adoption	**Lighthouse [143]** Highlights output in the context of code, but does not support input in code context.
G3	**Allow data transfer between static and dynamic contexts** Static and dynamic analyses should be tightly coupled so that users can switch between them during exploration.	**None we are aware of** We do not know of any complex analysis examples. This is possibly due to challenges with visualization and incremental analysis.
G4	**Allow selection of analysis methods** When multiple options for analysis methods or levels of approximation are available, ask the user to decide which to use.	**Hex-rays decompiler [185]** Minimally applies G4 by giving users a binary option of a potentially imprecise decompiled view or a raw disassembly view.
G5	**Support readability improvements** Infer semantic information from the code where possible and allow users to change variable names, add notes, and correct decompilation to improve readability.	**DREAM++ decompiler [432]** Provides significantly improved decompiled code readability through several heuristics, but is limited to a preconfigured set of readability transformations.

Table 5.2: Summary of guidelines for RE tool interaction design.

we discuss the tools closest to meeting the guideline (if any), how well it meets the guideline, and challenges in adopting the guideline in future tool development. Table 5.2 provides a summary, example application, and challenges for each guideline. While these guidelines are drawn directly from our findings, further work is needed to validate their effectiveness.

G1. Match interaction with analysis phases. The most obvious conclusion is that RE tools should be designed to mesh with the three analysis phases identified in Section 5.4. This means REs should first be provided with a program overview for familiarization and to provide feedback on where to focus effort (overview). As they explore sub-components, specific slices of the program (beacons and data/control-

flow paths) should be highlighted (sub-component scanning). Finally, concrete, detailed analysis information should be produced on demand, allowing REs to refine their program understanding (focused experimentation).

While this guideline is straightforward, it is also significant, as it establishes an overarching view of the RE process for tool developers. Because current RE tool development is ad-hoc, tools generally perform a single part of the process and leave the RE to stitch together the results of several tools. G1 provides valuable insights to single-purpose tool developers by identifying how they should expect their tools to be used and the input and output formats they should support. Additionally, with the growing effort to produce human-assisted vulnerability discovery systems [137], G1 shows when and how human experts should be queried to support automation.

The closest current tools to fulfilling G1 are popular reverse engineering platforms such as IDAPro [186], BinaryNinja [404], and Radare [328], which provide disassembly and debugger functionality and support user-developed analysis scripts. These tools allow REs to combine different analyses (N=16). However, due to these tools' open-ended nature and the lack of a prior RE process model, there are no clear guidelines for script developers, and users often have to perform significant work to find the right tool for their needs and incorporate it into their process.

G2. Present input and output in the context of code. We found that most REs only used tools whose interactions were tightly coupled with the code. This suggests that tool developers should place a high priority on allowing users to interact directly with (disassembled or decompiled) code. The best example of this

we observed was given by P05V in the code-coverage visualization plugin Lighthouse, which takes execution traces and highlights covered basic blocks in a disassembler view [143]. It also provides a "Boolean query where you can say only show me covered blocks that were covered by this trace and not that trace, or only show blocks covered in a function whose name matches a regular expression." However, Lighthouse does not fully follow our recommendation, as there is no way to provide input in the context of the code. For example, the user might want to determine all the inputs reaching an instruction to compare their contents. However, this is not currently possible in the tool.

G3. Allow data transfer between static and dynamic contexts. We found that almost all participants switched between static and dynamic program representations at least once (N=14). This demonstrates tools' need to consider both static and dynamic information, associate relevant components between static and dynamic contexts, and allow REs to seamlessly switch between contexts. For example, P04V suggested a dynamic taint analysis tool that allows the user to select sinks in the disassembler view, run the program and track tainted instructions, then highlight tainted instructions again in the disassembler view. This tool follows our suggested guideline, as it provides results from a specific execution trace, but also allows the user to contextualize the results in a static setting.

We did observe one participant using a tool which displayed the current instruction in the disassembly view when stepping through the code in a debugger, and there have been several analyses developed which incorporate static and dynamic

data [109, 170, 378, 417, 429, 440]. However, we are unaware of any more complex analyses that support user interaction with both static and dynamic states. Following G3 requires overcoming two difficult challenges. First, the analysis author must determine how to best represent dynamic information in a static setting and vice versa. This requires careful design of the visualization to ensure the user is provided relevant information in an interpretable manner. Second, we speculate that incremental program analyses (such as those of Szabo et al. [387]) may be necessary in this setting to achieve acceptable performance compared to current batch-oriented tools.

G4. Allow selection of analysis methods. Throughout the RE process, REs choose which methods to use based on prior experiences and specific needs, weighing the method's benefit against any accuracy loss (N=5). These tradeoff decisions are inherent in most analyses. Therefore, we recommend tool designers leverage REs' ability to consider costs and also recognize instances where the analysis fails. This can be done by allowing REs to select the specific methods used and tune analyses to fit their needs. One example we observed was the HexRays decompiler [185], which allows users to toggle between a potentially imprecise, but easier to read, decompiled program view and the more complex disassembled view. This binary choice, though, is the minimum implementation of G4, especially when considering more complex analyses where the analysis developer must make several nuanced choices involving analyses such as context, heap, and field sensitivity [371]. This challenge becomes even more difficult if the user is allowed to mix analysis precision

throughout the program, as static analysis tools generally use uniform analysis sensitivity. However, recent progress indicates that such hybrid analyses are beginning to receive attention [146, 216].

G5. Support readability improvements. We found most REs valued program readability improvements. Therefore, RE tool designers should allow the user to add notes or change naming to encode semantic information into any outputs. Further, because annotation is such a common behavior (N=14), tools should learn from these annotations and propagate them to other similar outputs. The best example of a tool seeking to follow this recommendation is the DREAM++ compiler by Yakdan et al. [432]. DREAM++ uses a set of heuristics derived from feedback from REs to provide semantically meaningful names to decompiled variables, resulting in significant readability improvements. One improvement to this approach might be to expand beyond DREAM++'s preconfigured set of readability transformations by observing and learning from developer input through renaming and annotations. This semantic learning problem poses a significant challenge for the implementation of G5, as it likely requires the analysis to consider minor nuances of the program context.

RE tool designers should consider the exploratory visual analysis (EVA) literature. In addition to the guidelines drawn directly from our results, we believe RE tool designers can draw inspiration from EVA. EVA considers situations where analysts search large datasets visually to summarize their main characteris-

tics. Based on a review of the EVA literature, Battle and Heer define a process similar to the one we observed REs to perform, beginning with a high-level overview, generating hypotheses, and then iteratively refining these hypotheses through a mix of scanning and detailed analysis [37]. Further, Shneiderman divided EVA into three phases, similar to those we suggest, with his Visual Information Seeking Mantra: "Overview first, zoom and filter, then details-on-demand" [362]. While techniques from this field likely cannot be applied as-is due to differences in the underlying data's nature, these similarities suggest insights from EVA could be leveraged to guide similar development in RE tools, including methods for data exploration [181, 213, 317, 369], interaction [180, 207, 323, 434], and predicting future analysis questions [36, 104, 154, 403].

Chapter 6: The State of Online Hacking Exercise Pedagogy

Moving from supporting security experts to helping non-experts build the experience needed to find vulnerabilities, in this chapter, I investigate the current state of online hacking exercises — the method of learning almost all security experts reported using in Chapter 4. Historically, the security community has used online hacking exercises to provide practical education, exposing participants to a variety of vulnerabilities and security concepts. In these exercises, participants demonstrate their understanding of security concepts by finding, exploiting, and fixing vulnerabilities in programs. They offer — in contrast to more traditional project-based learning — discrete practice sets that can be undertaken in a modular fashion, similarly to the exercises commonly included at the end of each chapter in mathematics textbooks. In fact, hacking exercises are commonly considered a very useful educational tool, with security experts often reporting that they rely on these exercises for their education [413], bug bounty platforms directing those interested in security to start with these exercises [162,195], and a significant amount of recent security education work focused on creating new hacking exercises [27,44,107,110,275,306,416]. Further, prior work has provided some evidence that hacking exercises can provide valuable immediate feedback to learners in academic settings [59,136,306,327,346].

However, this evidence is limited for several reasons. First, these studies only consider a few exercises [27, 44, 83, 107, 110, 275, 276, 306, 346, 416], producing *sparse* results and providing limited understanding of the broad set of popular exercises students participate in. Next, this work only focuses on a few measures of learning and engagement [107, 275, 306, 327, 395, 416], making the evidence *narrow*. They do not consider significant learning factors which are difficult to control for and measure. Therefore, exercise organizers have little guidance for building effective exercises, educators can not know which exercises provide the most effective learning, and researchers do not have a broad view of the landscape of current exercises.

In this chapter, my colleagues and I review online hacking exercises to provide perspective on the current landscape of these educational interventions. Specifically, we set out to answer two main research questions:

- **RQ1**: Do current exercises apply pedagogical principles suggested by the learning sciences literature? If so, how are these principles implemented?
- **RQ2:** What challenges do exercise organizers face in applying these principles?

To answer these questions we reviewed 30 popular online hacking exercises (68% of all online exercises we identified), completing a sample of 306 unique challenges across all reviewed exercises. For each exercise, we considered whether pedagogical principles recommended by the learning theory literature [13, 47] were implemented. Additionally, we interviewed 14 exercise organizes to verify our results and understand their reasoning for implementing (or not implementing) each principle.

6.1 Methods

To understand the current landscape of online hacking exercises, we performed a two-phase study: a qualitative review of popular online exercises and interviews with the organizers of these exercises. In this section, we discuss how we identified and selected exercises for review, our review process, and interview protocol.

6.1.1 Exercise Selection

There are many kinds of resources available to security students, such as vulnerability write-ups, certifications, academic coursework, and books. To limit the scope of our inquiry, we chose to focus on popular online educational exercises, based on prior work suggesting this type of intervention as preferred among security experts [413]. Specifically, we only consider exercises which meet the following criteria:

- **Educational** - Because we are evaluating the educational benefit of each exercise, we only include exercises which explicitly state education as a goal. Therefore, we do not consider competitions, such as the DefCon Qualifiers, whose goal is to identify the "best" hackers.

- **Hands-on** - The exercise included a hands-on component requiring students to actively practice security concepts. This component could be the central focus of the exercise—as was the case in many CTFs—or an auxiliary component, e.g., presented after a series of associated lectures.

- **Online and publicly accessible** - From a practical standpoint, we chose to focus on online exercises so that we could perform a full and fair analysis of all exercises by actually participating in them. This prevents us from having to make possibly incorrect assumptions based on an offline exercise's description.

- **Popular** - Our goal is to understand the hacking exercises students are most likely to participate in. Therefore, we chose to specifically focus on the most popular sites. To estimate a site's popularity, we used it's Tranco rank—a secure method for ranking sites based on user visits [241]. We chose to use Tranco over Alexa or other site ranking services because it provides better stability and security guarantees. We used the version of the list from October 15th, 2019. Because Tranco only tracks the top one million sites, we used Alexa rankings whenever no Tranco ranking was available. Each site's rank is given in Table 7.5. Note, this is only an approximation of popularity as the ranking indicates the domain's popularity, not the specific sub-domain of the exercise, causing exercises associated with popular websites to be ranked higher (e.g., gCTF whose URL capturetheflag.withgoogle.com). However, this was not a common problem in our data, so we believe this popularity ranking is a reasonable approximation.

6.1.1.1 Exercise Identification

We began our search for currently available exercises meeting our criteria by contacting eight security education experts recruited through one author's personal

and professional relationships. We asked each to recommend specific exercises, publicly available lists of exercises, and possible search keywords. Based on their recommendations, we used the Google search engine with all possible combinations of the terms "cybersecurity," "computer security," and "security" with "capture the flag," "CTF," and "war games." We also included the search term "hacking exercises." For each query term, we reviewed the first 10 result pages for candidates. Our experts also suggested several lists of exercises curated to help newer hackers find exercises to get started [97, 236, 348, 364, 426], which we reviewed for exercises meeting our criteria.

Additionally, for each exercise and recommendation list identified, we also included the top three similar sites identified by Alexa.com.[1] We continued this process until no new exercises were identified. Our candidate identification process was completed in October 2019.

While almost all of the exercises we identified were set up to be joinable year round, we observed that many were initially designed to be played as a live, short-term competition (over the course of a day or week). Because this difference in design is likely to impact the exercise's structure, for comparison purposes, we divided exercises into two categories based on the original participation context for which they were originally designed:

- *Synchronous* (N=12) - These exercises were originally designed for students to participate at the same time over a short time period (i.e., a few days or weeks). This includes most capture-the-flag (CTF) competitions. Challenges in these

[1] https://www.alexa.com/siteinfo

exercises are made available after the competition for additional students to try at their own pace.

- *Asynchronous* (N=18) - These exercises are designed to be joined at any time and worked through at the student's pace. These are often referred to as "wargames" in the computer security community.

6.1.1.2 Sample Selection

Our identification process revealed 45 exercises meeting our criteria. Due to the significant time required to complete each review (2.5 hrs on average), we chose to sample 60% of exercises for in-depth review—approximately 60 total person hours. Because our goal was to focus on exercises reaching the most participants, we first selected all of the top 30% of exercises (by popularity rank) in each group (i.e., 6 Synchronous, 8 Asynchronous). We then randomly sampled from the remaining exercises until we selected 60% of the total number of exercises for each group (i.e., an additional 6 Synchronous, 10 Asynchronous). We chose to include these less visited exercises to account for newer exercises that are still growing in popularity. The final list of selected exercises is given in Table 7.5. Note, my colleagues and I organize one of the randomly selected exercises (Build It, Break It, Fix It [69]). However, due to its unique organizational structure, we could not find a reasonably analogous exercise and chose to retain it to ensure our review covered the breadth of available exercises.

6.1.2 Pedagogical Review (RQ1)

To determine the set of pedagogical principles for review, we drew on previous efforts to synthesize major theoretical and empirical learning sciences and education research findings into actionable principles [47]. This led us to five core pedagogical principles: *connecting to learners' prior knowledge* [320, 321], *organizing declarative knowledge* [47], *active practice and feedback* [13,119], *encouraging metacognitive learning* [131,178], and *establishing a supportive and collaborative learning environment* [13]. To identify actionable dimensions of each core principle, we started with the 24 dimensions used by Kim and Ko [220] in their similar review of online coding exercises. Two of the authors then updated these dimensions to be specific to hacking exercises through collaborative open coding of five exercises. This process resulted in 36 total pedagogical dimensions, across the 5 core principles. We discuss each dimension in further detail in Section 6.2.

For each selected exercise, we performed a qualitative coding where another researcher and I evaluated each exercise independently according to the pedagogical dimensions. For reporting simplicity, we used three options, *yes* (●), *no* (○), and *partial* (◐), for each dimension — unless otherwise indicated — , as shown in Table 7.5. Using an open coding approach, the definition for these levels of each dimension was determined based on the analyzed exercises. That is, the details differentiating a partial from a complete implementation emerged from our exercise review. In most cases, a "yes" indicates the dimension was implemented in a significant majority of challenges. Conversely, a "partial" indicates this dimension

is only implemented in a small number of challenges and "no" means all reviewed challenges covered independent concepts. We give specific examples of dimension implementation levels throughout Section 6.2 when they differ from this general definition.

The other researcher and I completed at least one logical unit of the exercise (e.g., all questions in a category or a single specified path through the exercise), or five challenges if no logical relationship was present, to gain a full understanding of the exercise. We completed ten challenges on average per exercise (306 total).

After establishing our initial codebook, the other researcher and I independently reviewed 20 exercises, comparing dimension codes after every five exercises for inter-coder reliability. Note, in cases where the dimension could be assessed without any judgment decisions, inter-coder reliability was not calculated, as it is unnecessary [262]. For example, when evaluating whether solutions were available, exercises were given a "yes" if they offered direct links to solutions or we could find a walkthrough on the first page of a google search for the challenge and exercise name. To measure inter-coder reliability, we used Krippendorff's Alpha (α), which accounts for chance agreements [179]. After each round, the other researcher and I resolved coding differences, modified the codebook when necessary, and re-coded previously reviewed exercises. This process was repeated until an α of at least 0.8 — the recommended threshold for result reliability [179] — was achieved. The remaining exercises were divided evenly between the two researchers and each coded by a single researcher. The final α values for relevant dimensions are given in the first row of Table 7.5.

6.1.3 Organizer Interviews (RQ2)

Because we reviewed a limited set of challenges in each exercise, we wanted to confirm our analysis results by giving the organizers an opportunity to provide clarifying information. Also, to answer our second research question, we needed additional context from the organizers to understand their decision-making process. As such, we reached out to the organizers of all 30 exercises. We provided each organizer with a report describing our review and invited them to participate in a 45 minute structured interview or respond to our review via email. Each report gave all pedagogical dimension definitions, our coding for their exercise, and the reasoning behind our decisions. Note, in our report and throughout our interviews, we were careful to ensure organizers understood our goal was to understand their decision-making, not critique it. We made sure to adopt a constructive tone instead of presenting findings in an accusatory manner. We let organizers know we invited and expected disagreements with our evaluation, as there were likely exercise elements or viewpoints we had not considered.

In our interviews, we walked organizers through the report and asked whether they agreed with our assessment and if not, why. Based on organizer feedback, we revisited our results and updated codings when necessary. Specifically, these updates only occurred when the organizers pointed us to challenges or pages on the exercise we may have missed. If on further review of these missed elements, we determined a dimension was actually implemented we updated our results. For dimensions not implemented, we asked organizers whether they considered the dimension when

building their exercise and if so, why they did not implement it. Our full interview protocol is given in Appendix A.4. Because this component of our study constituted human-subjects research, it was reviewed and approved by our organization's ethics review board. All raw records including organizer personally identifiable information were maintained securely, only accessible to authorized members of the research team.

To identify themes in organizers' decision-making we again performed open coding, this time coding reasons dimensions were not implemented in the 14 organizer responses via email or interview. Responding organizers are indicated in Table 7.5. To establish our codebook (detailed in Appendix B.5), two researchers reviewed three interviews in collaboration. Then, those researchers independently coded 12 interviews, comparing codes after every three interviews until a sufficient level of inter-rater reliability (α) of 0.86 was reached [179]. The remaining interviews were divided evenly between researchers and coded by a single researcher.

6.1.4 Limitations

This study has several limitations, some related to our sampling method and some common to exploratory qualitative research. First, it is likely that we did not identify all all exercises meeting our stated criteria through our review. Additionally, because we only perform our review on a sample of exercises, we may have missed a particularly good implementation of one of our educational interventions. However, because of our thorough search process and by weighting our sample toward more

popular exercises, our results are likely representative of most students' experience.

In our pedagogical review, we adopt a conservative approach, checking whether the dimension is implemented, but not whether it is implemented *well*. We did this so we could broadly evaluate the types of pedagogy considered and establish an initial understanding of the current landscape. However, this broad view does not allow us to make statements about the efficacy of specific approaches taken. We encourage future work to build on our established roadmap through more focused review.

Further, there is likely self-selection bias related to which organizers agreed to be interviewed. In general, organizers who are more engaged in supporting student learning experiences may be more likely to respond to a request to discuss pedagogy. We also observed anecdotally that organizers whose exercises implemented more of our pedagogical dimensions were more likely to agree be interviewed. While this may reflect engagement in pedagogy, it may also indicate that — despite our best attempts to ensure our feedback was positive and constructive — some organizers found our comments or interview request pejorative. In addition, social desirability bias suggests that organizers may (consciously or unconsciously) tailor their responses to appear in the best possible light. To partially mitigate this, we only revised our dimension coding if organizers identified exercise elements we missed in our initial review, but did not allow organizers to argue for pedagogical dimension redefinition to better suit their exercise. Overall, our findings regarding organizer decision-making should be interpreted within this context, and may reflect a higher-than-average degree of interest in improving student learning. Nonetheless,

we believe they provide novel insights into security education and directions for future work.

Finally, throughout the next section, we give the number of exercises (N) and organizers (O) that demonstrated or expressed, respectively, each concept, to indicate prevalence. Note, if an organizer did not indicate a specific reason for not implementing a given pedagogical dimension, this is not necessarily indicative of disagreement; instead, they may have simply failed to mention it.

6.2 Results

Our review's final results are given in Tables 6.1-6.4 and B.8. Each exercise was assessed on all 30 pedagogical dimensions. Exercises are grouped into synchronous and asynchronous, then sorted with the most popular exercises first. Overall, we found that while some exercises implemented more pedagogical dimensions than others, there were no exercises implementing all dimensions. Additionally, we observed innovative approaches to education distributed among all exercises. In this section, we organize our discussion around our five core principles, considering each evaluated dimension in detail. For brevity, we only discuss the 23 dimensions included in Tables **??** and **??**, which included reasonable differentiation between exercises. The remaining 7 dimensions can be found in Appendix B.6.

6.2.1 Connecting to students' prior knowledge

Previous research in learning science has shown that people develop new knowledge based on their pre-existing knowledge and beliefs [84, 320, 321, 414, 415]. This prior knowledge can be in the form of facts, perceptions, beliefs, values, and attitudes [84, 320]. Students interpret any new information they are presented through the lens of their current view of the world and they bring a variety of prior experiences into learning. Therefore, exercises should consider these to provide effective education.

Students develop their understanding through the production of analogies and connections to previously learned concepts—in the same domain or otherwise. The prior knowledge a student brings to a new context can have a significant effect, facilitating learning if their knowledge is accurate and complete, or hindering learning if they bring inaccurate or incomplete preconceptions. Therefore, careful consideration of the students' prior knowledge and deep connection to and activation of that knowledge is important to successfully build new knowledge.

Additionally, supporting tailored education can have a positive effect on student motivation. That is, if challenges are appropriately tailored to the student, they will not feel out of their depth, instead growing their self-confidence in their learning ability [359].

To evaluate how the reviewed exercises connected to students' prior knowledge, we considered two groups of dimensions: *personalization* and *utilization*.

6.2.1.1 Personalization

Our first dimension considered the knowledge students bring into the exercise. Each student has a unique background, so exercises should adjust the presentation and difficulty of challenges to account for these differences, or target specific subpopulations [325]. In our review, we considered three dimensions we believe are most likely to affect student learning background: *age*, *educational status*, and *security experience*.

Experience-based personalization was common. Most (N=22) exercises allow some personalization based on experience. These exercises used a mix of difficulty indicators, including difficulty labels (e.g., Easy, Medium, Hard) (N=10), the number of other students who have solved the challenge (N=14), and point values (i.e., more points indicate increased difficulty) (N=18). This lets participants attempt problems appropriate to their experience level, avoiding burnout on a problem beyond their reach or boredom with too many challenges they can easily solve. This student-guided personalization method has the added benefit of providing autonomy, making the student feel more involved in their own learning process and therefore more motivated to continue participation [359].

Difficulty levels and point assignments are not optimal. These assignments are made based on the best judgment of the organizers and it can be hard for students to determine what "Easy" or "10pts" means to the organizers. In our review,

we observed multiple cases where more complicated challenges were rated easier or assigned fewer points than less complex challenges in the same exercise. This supports prior findings, which have shown similar issues with difficulty labeling [83], commonly due to inconsistencies between multiple challenge authors—a common practice in security exercises (N=18). The Vulnhub organizer explained this problem, saying "What you find easy I will find difficult and vice versa... someone new to the industry [might say], 'This is the first time I've seen this, this is really super hard.' Then give it to a seasoned pen tester and he thinks 'I saw that two weeks ago.' And he's like, 'Eh."' This is problematic because it not only hinders students' ability to personalize their learning, but also could lead students to lose self-confidence. Several exercises appear to understand this problem and try to mitigate it by allowing students to rate or comment on exercises (N=5). However, in HackTheBox, challenges can only be rated after solving the challenge, missing feedback from a likely important segment of students, i.e., those stuck on the challenge due to its difficulty. Additionally, two exercises, picoCTF and Root-me.org, only allow students to indicate whether they liked the challenge, which might not correlate with challenge difficulty.

Only two exercises activated prior knowledge. While most exercises implicitly leveraged prior knowledge, only two—Root-me.org and HackthisSite—activated it by drawing the students attention to previously learned concepts they should remember when trying to solve the challenge. They both did this by including a list of related pre-requisite knowledge (e.g., HackthisSite listed "some encryption knowl-

edge" as a pre-requisite for a Caesar cipher decryption challenge). By pointing to specific prior knowledge, the exercise helps the student not only build on their prior knowledge, but select the appropriate knowledge to build on, therefore helping them avoid potential misconceptions [292].

Some exercises required challenges to be solved in increasing complexity order. When exercises did not personalize by experience (N=8), they always had a single problem path for the student to follow (e.g., success in one challenge required to unlock the next), or more complex problems were only unlocked when the student solved enough low-level problems. This could get tedious when students have to solve several of the same types of problems. For example, in picoCTF more experienced students may be frustrated as they are required to solve several simple problems—designed for new learners—before they can unlock more interesting challenges. When asked the reason for this design, the picoCTF organizers explained it was "just for convenience, since the year-round version is not really any different from the actual competition period." This was a common sentiment among organizers, with all but the XSS-Game organizers stating the lack of experience personalization was related to the competitive element of the exercise. XSS-Game's organizers forced students to follow a specific path to prevent students from jumping in too far and feeling overwhelmed.

Few exercises personalized based on age or education. Eight made explicit mention of the age or education level targeted. The remaining exercises appeared

to target university-level students or above. Pwnadventure's organizer informed us that the exercise was originally designed to be live in conjunction with the finals of a larger university-level CTF—CSAW CTF [234]. While targeting a more educated audience is likely necessary for more complicated concepts, it should be clearly stated—possibly with links to other resources—to help younger, less educated students who might otherwise be deterred from hacking exercises entirely. Pwnadventure's organizer agreed, saying "It wouldn't be a bad idea to give people that context and just say, 'This is how it was designed. So if you find this too hard, that's expected. This was intended for this audience."'

6.2.1.2 Utilization

For *utilization*, we checked whether the exercises required students to leverage knowledge accumulated through prior challenges in solving later challenges, i.e., building on within-exercise prior knowledge.

Exercise designers build clear challenge concept progressions. Almost all exercises (N=29) include some challenges whose concepts build one on top of the others where appropriate. As an example, Microcorruption offers a progression across several challenges to teach buffer overflow concepts in a binary exploitation challenge. It begins with a program that requires the student to disassemble the program and read a hardcoded password string. The next challenge, forces the student to actually read the assembly code and understand the stack to reconstruct the password from a set of characters. Then, the student must exploit a simple

buffer overflow with no mitigations in place to force execution down a successful path. The progression then continues by adding further mitigations to complicate the exploitation process.

In the case where subsequent utilization of knowledge was not observed (Infosec Institute), all the challenges covered a disparate set of unrelated concepts. This was likely becasue this exercise had among the least number of challenges, but chose to cover a breadth of topics. We expect we would see additional challenges building on this knowledge if the creators chose to include additional challenges.

6.2.2 Organizing Declarative Knowledge

Another key to effective learning comes in students' ability to transform facts into robust declarative knowledge [47]. To achieve mastery, students must go beyond simply memorizing tricks to solve specific challenges, but also organize these disconnected facts based on their underlying abstract concepts into a structured knowledge base [13, 321, 414, 415]. Prior work comparing experts and novices has found that while experts do tend to know more facts, their biggest improvement comes from rich structure of their information base [13]. This structure allows them to improve recall, recognize patterns, make analogies with previously observed scenarios, and identify key differences between contexts, supporting improved knowledge transfer [13, 223]. For example, after solving a Caesar cipher challenge that requires rotating the ciphertext to produce intelligible plaintext, and a cryptographic hashing challenge requiring a dictionary attack, the student should identify the common

cryptographic weakness related to limited key spaces. They can then apply this abstract concept to solve a future challenge: decrypting text encrypted with RSA where small prime numbers were used to generate the public/private key pair.

To support students' construction of a deeper conceptual understanding, exercises should organize challenges according to the concepts taught to make these knowledge structures clear [51, 79]. For this core principle, we considered how the information itself was organized and the context in which it was presented. We also considered the types of security concepts covered by each exercise, but found little differentiation across exercises. For brevity, we leave reporting on these results to Appendix B.6.

6.2.2.1 Organization

Students can organize knowledge into a variety of possible structures. When asked to organize facts, experts often sort them into hierarchical structures, demonstrating their deeper understanding of the complex interrelationship of information [13]. Through the use of textual and visual cues, exercises can help students make these organizational connections. In our review, we considered how concepts were organized across challenges and whether exercises included any cues (e.g., textual, visual, or structural) to help students recognize and overarching structure. Specifically, we looked at whether the exercises grouped challenges by concept, creating a hierarchy, or highlighted a related problem path, showing linear concept progressions through challenges.

Many exercises lacked clear structure. Providing explicit cues, such as using challenge names that indicate a hierarchical concept structure or suggesting a progression through conceptually-related problems, can help students associate individual facts [13, 79]. Unfortunately, a majority (N=17) of exercises did not clearly organize problems so challenges with related concepts were grouped together. Similarly, several exercises did not provide students with a path to follow through more than two to three challenges as an organizing guide to the concepts (N=11).

Interestingly, almost every exercise that relied on crowd-submitted challenges — useful for reducing organizer workload to produce a greater number of challenges — (N=6) did not provide a clear structure. Vulnhub's organizer explained "There is metadata for about a quarter of the challenges on the backend that's saying, this one has file inclusion, this one has an apache vulnerability, whatever. I was going to implement another feature that would take this metadata and help plot it out. Then VMs went up and I have never implemented it... I've just not kept up to date with it because I was going through everything manually and you can't trust each author's opinion." The authors of Root-me.org offer the counterexample to this trend, deciding to present author-provided metadata, either without review or by dedicating more organizer time to this task. This shifts some of the organizational work to the challenge developers (e.g., identifying tasks), reducing the effort required of organizers. Note, the author-provided categorizations are quite broad (e.g., buffer overflow, format string, Javascript, etc.), indicating only general relationships across large groups of challenges. Students still only get more fine-grained connections between challenges on a per-author level (e.g., when authors use the same challenge

name with incrementing indexes to show a progression).

6.2.2.2 Context

We also reviewed the *context* within which concepts were organized, which can impact information retention and conceptualization [13, 47, 51]. We considered four dimensions of context. First, we looked at whether authoritative content was presented with the challenge (e.g., video or textual lecture). Next, we asked whether the exercise used a goal-driven project approach, which can help students' active engagement in practice, as they see how individual challenges fit within a broader, more realistic context [119]. We also considered whether any overarching story or narrative was provided to connect learning, as people are more likely to remember information when presented in narrative form [35]. Finally, we assessed whether any challenge programs demonstrated realistic complexity, i.e., significant functionality representative of programs seen outside the educational setting. We note that there is significant benefit in simplifying challenge complexity, as it removes the need to perform repetitive tasks (e.g., port scanning), focuses student attention on specific problems [384, 402], and provides less experienced students an entry point into the exercise. However, having some realistic challenges helps students see concept relevance, improving intrinsic motivation [359], and can support knowledge transfer [223] and the development of practical skills [118].

Stories are the only commonly used method to provide context. Many exercises included narrative elements throughout the exercise (N=11). The GirlsGo

CyberStart organizer said they chose to embed each challenge within a narrative to teach "Why is a hacker or bad guy using this action? As an educator, you're trying not to just state facts and have them absorb them or try a technique and just do it. You want to give context." While this was not used in a majority of exercises, it was by far the most prevalent practice. Conversely, very few exercises used lectures (N=6) and only one included challenges that built on each other to reach a broader goal. Interviewed organizers who did not include these contextual elements (O=4) said they "got in the way of creativity" (Angstrom) (O=2) or because "they're submitted by several different people" (O=2) (Crackmes.one).

Few exercises included realistic challenges. Few exercises included any challenges representative of real-world-scale programs (N=8). This practice potentially prevents students from learning practical skills necessary for scaling analyses to larger programs. Many organizers indicated they chose to avoid realistic challenges because they believed focusing students on specific concepts was more important (O=9) and developing challenges with this level of complexity is difficult (O=1). Others chose not to include complexity—and therefore require the student to perform extraneous tasks—because they wanted to make sure their exercise was fun and engaging (O=5). The gCTF organizers explained "you focus mostly on the problem solving part that gives the players joy... The recon part is required in real world pen testing..., but in some cases, it either will be mostly luck based if you are looking in the right place at the right time, or developing the [necessary] infrastructure will just take most of your weekend." While realistic challenges were not common,

HackEDU presented the most promising approach to incorporating real-world programs into their challenges: providing vulnerable programs reproduced from public vulnerability reports on HackerOne, a popular bug bounty platform.

6.2.3 Practice and Feedback

Prior work shows students must perform a task to achieve domain mastery [118, 223, 345]. Through deliberate, active practice, students can translate abstract concepts into practical knowledge. To support this practice, students must also receive tailored feedback to guide their learning to specific goals [13]. Without feedback, students may become stuck or misunderstand the challenge's learning objective [13, 47, 79]. For this core principle, we considered two dimension groups: *actionability* and *feedback*.

6.2.3.1 Actionability

For *Actionability*, we considered the types of tasks exercises ask students to complete. Specifically, whether students had to exploit insecure programs (e.g., perform a buffer overflow or decrypt a weak ciphertext) or write secure programs—from scratch or by patching vulnerable code. Note, for the latter, we did not consider exercises that required students to write programs for exploitation purposes (e.g., to brute-force a cryptographic challenge). We only considered an exercise as meeting this dimension if the code students produced was evaluated for security vulnerabilities. We found that all exercises required students to exploit programs, so we show

these results in Appendix B.6 for brevity.

Secure development practice was uncommon. Very few exercises (N=4) included any challenges asking students to write secure code and two of those (Hellbound Hackers and Pwnable) only included a few. Instead, students are left to make the logical jump from identifying and exploiting to preventing a vulnerability without educational support. For example, XSS-Game—explicitly targeted at training developers—has students identify XSS vulnerabilities, but does not include any information regarding the impact of these types of vulnerabilities or how to avoid them. In many cases, organizers simply felt that including secure development practice was difficult to evaluate (O=7). This included the XSS-Game organizers, who said "the problem is that it's really hard to test that [the vulnerability is] fixed properly... You actually either have to have somebody manually test it, or a really good checker that's checking a ton of edge cases." Other organizers chose not to include secure development challenges because they wanted to limit the scope of their exercise to focus students on exploitation (O=7).

The two notable exceptions were HackEDU [161] and BIBIFI [69], both using different approaches. HackEDU used a similar structure to other exercises, asking students to first identify and exploit the vulnerability in sample code. However, students then patch the vulnerable program to make it secure. BIBIFI took a different approach, asking participants to first write a medium-sized program, considering tradeoffs between security and functionality. Other teams then search for vulnerabilities in the student's code; when found, the original students are asked to patch

the identified vulnerabilities.

6.2.3.2 Feedback

To analyze *Feedback*, we considered several forms feedback might take. The first, and most helpful, was direct feedback, in which guidance is tailored to the approach the student took to try and solve the challenge, directing them down the "correct path." We also considered whether the exercise included less tailored feedback in the form of static hints or opportunities for students to seek out feedback through forums or challenge walkthroughs.

Exercises rarely provided direct feedback throughout. Most exercises only provided direct feedback in the form of a "flag check" (N=18): allowing a student to verify they have identified the correct solution by submitting a random string associated with challenge success. This string matching is problematic, as simple typos or copy/pasting issues can lead students to misinterpret rejected submissions as incorrect solutions. This problem is further exacerbated when some exercises (N=4) do not use consistent flag formats, causing students to question whether the string they found is actually a flag.

Some challenges provide "correct path" markers. In the exercises that awarded partial credit (N=10), some challenge developers included checks in the target program's execution to update their output if the student's exploit was following the *correct path*, even if the exploit was not yet fully successful. For example,

in Root-me.org's *Format string bug basic 2* challenge, the program checks whether the student modifies the target address at all, even if it is not changed to the correct address. If the target address is modified, the program writes "You're on the right track". However, this type of feedback was very sparse in each of these exercises, with only one or two challenges providing it.

Many organizers said providing this type of tailored feedback was difficult because this feedback had to be specifically tailored for each challenge (O=5). The GirlsGo CyberStart organizer pointed out that for some challenges where exploitation occurs locally, they do not have a way to track student behavior, saying "How would I know what [the student] is staring at?" Instead of providing automated feedback, many organizers opted to provide tailored information in the exercise's forum based on student demand (O=5). Smash the Stack's organizer explained "They'll either email us and say, 'Hey, we're stuck here' and we'll respond or they'll join IRC and ask their questions there. Usually, someone, an admin or just other players will exchange hints." We discuss this forum-based support in more detail in our review of the *Question support* and *Group discussion* dimensions in this section and Section 6.2.5.1, respectively. Finally, three organizers expressed that they had not considered providing automated feedback, but agreed that it would be useful to include in future challenges.

BIBIFI provides exploit examples. In BIBIFI's break phase, other students identify and exploit vulnerabilities in a team's code, demonstrating mistakes made when writing secure code. Because this likely produces multiple variations on the

same exploit, students receive rich feedback on the mistakes made. Unfortunately for students producing exploits, the interface does not provide any significant feedback beyond success or failure. Also, because students only work on a single project, there is no opportunity to practice and receive feedback in varied contexts.

Students can get help when stuck. All but three exercises either allow students to ask the organizers questions when they are stuck or provide static hints to help point students in the right direction (21 do some of both). For example, XSS-Game allows students to reveal hints, which get progressively more informative. In the early challenges in XSS-Game, the final hint provides the solution to help inexperienced students get started. Several organizers prided themselves on the support provided by the community and relied on these informal communications to support struggling students (O=5). Unfortunately, if hints are not well crafted, they can be misleading and cause the student to consider an incorrect path. For example, in CTFlearn's *Prehashbrown* challenge, the student is given the hint that they need to "login as admin" and are shown a login screen. This might make the student think you need to exploit the provided login form. However, the student actually needs to register an account and exploit an SQL injection vulnerability in a search screen that is provided once logged in. We did not observe any relation to the admin account when solving this exercise.

Also, in every exercise except BIBIFI and Mr. Code, students can find textual or video instructions for how to solve some challenges. In several cases, the exercise provided a few directly on their website (N=10). For example, in GirlsGo

CyberStart and picoCTF, the organizers provide videos demonstrating how to solve the first few challenges to get students started. We also found that walkthroughs were produced organically by participants for most exercises (N=26), even if some organizers already provided some walkthroughs themselves (N=7) or tried to discourage their creation to prevent students from copying solutions (O=3). Note, at the outset of our coding process, we planned to consider directly provided and externally produced walkthroughs separately. However, we found that no exercise directly provided solutions for all challenges. Therefore, we chose to mark exercises with any directly provided walkthroughs as implementing the dimension and those with only externally produced solutions as partially implementing.

6.2.4 Encouraging Metacognitive Learning

Metacognitive learning consists of two main components: students' ability to predict learning task outcomes, and their ability to gauge their own grasp of concepts [51, 79]. Guiding students to reflect on which solutions worked and why helps students develop a deeper conceptual understanding, supporting knowledge transfer to new settings and challenges [131, 309, 354, 355]. It also helps students identify gaps in their understanding and target further learning [131].

6.2.4.1 Transfer Learning

To determine whether exercises supported metacognitive learning, we asked whether they taught *how*, *when*, and *why* students should use a particular exploit

or mitigation technique. Answers to these questions are important to help students apply knowledge learned from the challenge to real-world use.

Few exercises guided transfer beyond the challenge context. While the exercises almost all taught *how* to use each concept through hands-on exercises (N=29), very few explicitly explained *when* (N=6) or *why* (N=5) to use the security concept in other settings. In these few cases, the organizers provided authoritative materials (e.g., video lectures or additional reading) around each challenge instructing students on the specific setting details and how approaches should change with new settings. For example, HackEDU and Pwn College provide lecture materials describing progressively more challenging exploit techniques in the face of ever increasing defensive mitigations. While this is a useful tool for learners — and the best method observed across exercises — it falls short of best practice recommendations for metacognition, which suggest active student engagement.

The partial cases were used if it was possible to implicitly determine *when* or *why* a particular concept was needed by comparing similar challenges. However, this is not an optimal solution, as students would likely need a sufficiently strong *a priori* conceptual understanding to identify the nuanced differences.

Interestingly, this was this was the dimension group organizers most often reported not considering (O=9). As an example, when we explained metacognition to the picoCTF organizer, they said "I don't know if I ever heard of metacognition before... that could really guide us in developing problems that can guide our learners even better."

6.2.4.2 Support

Once students understand their relative grasp on concepts and identify points requiring more clarity, they will seek additional information to fill those gaps. Exercises can therefore support students by providing links to additional materials beyond the exercise's scope.

Most exercises provided resources for further investigation. A majority of exercises did provide additional resources to some extent (N=17). These materials often took the form of a "Useful links" page (N=15), though some only provided resources for a subset of covered concepts (N=4). Also, while these lists of resources are useful, it can be difficult for students to identify which resource to follow for a specific question. Some exercises improve on this by providing resource links with each challenge to indicate which are relevant to that problem (N=10). For example, HackEDU included links to blog posts discussing related concepts in the challenge description. Organizers who did not provide these resources generally believed students were provided enough information in the problem descriptions to find resources on their own (O=6). The XSS-Game organizers expressed this sentiment, saying "Either from the description of the challenge or the source code, the user should be able to figure out what to learn about."

6.2.5 Establishing an Environment Conducive to Learning

Finally, we considered the social environment in which the students participated. Social climate (e.g., interactions with other students and educators) has been shown to impact learning, i.e., a negative environment can hamper student progress, while a positive environment can excite and engage students [13, 313]. By participating in a group setting, students are able to receive mentoring from more senior students, brainstorm possible solutions with peers, and get support and encouragement when they get stuck [294]. Additionally, the exercise framing can have a significant impact on whether students feel "good enough" to participate [313]. If the perceived barrier to entry is high, students may choose not to try. This is especially true for commonly underrepresented populations [200, 418, 425].

We characterize the environment along two dimension groups: interactions between students (*Peer Learning*) and between the organizers and students (*Inclusive Setup*).

6.2.5.1 Peer Learning

Peer learning intuitively lightens the student support burden on organizers, as other students can act as a first-line support. More importantly, allowing students to work together gives them the opportunity to collaboratively develop knowledge (as opposed to being given it directly by the organizer), producing more robust understanding [294]. It also has been shown to improve intrinsic motivation—namely social motivation—as students who feel they are part of the learning process and

others depend on their participation are more likely to continue in the face of difficulties [13, 359]. To evaluate whether an exercise provided peer-based learning, we considered whether it explicitly encouraged team formation through a provided team-creation feature (i.e., not just allowing team creation out-of-band) and whether there is an online forum created by the exercise organizers for students to discuss possible challenge solutions.

Exercises help students find community through online forums. Most exercises provided IRC, Slack, or Discord channels or online forums, where students could post questions and share their experiences with other competitors (N=17). For example, picoCTF created a dedicated Q&A forum in Piazza [**?**] with sections for each problem category. Students could post questions when stuck on a problem, or view (and respond to) questions posed by other students. As we stated in our discussion of the *Feedback* and *Question Support* dimensions, several exercise organizers said community participation is important for student success (N=7). The HackTheBox organizers explained that they have "a vocal community that everyone chats... in order to help each other to understand challenges and learn." Similarly, the Crackmes.one organizer said "for a newcomer to the platform, if they don't join the discord, they will not have all the information." Even when organizers did not provide a discussion forum, the students themselves organized one (these were marked as "Partial" implementations, N=3). Because they are not directly linked by the exercise, these partial cases were only identified through organizer interviews, so our review is a lower bound on the number of exercises with a discussion forum.

Almost all the exercises that did not have a forum were in the *Synchronous* category. Their organizers explained they lacked a forum because of their initial competitive design (O=5). For example, the Angstrom organizer explained that during the competition "Everybody's competing against each other." Now that the competition is over, "people are now allowed to collaborate. We should probably add a channel to support that, but we have not." The only other organizer we spoke to who does not provide a discussion forum (Mr. Code) explained, "people join at different times and learn at different rates," so it does not make sense to include a forum.

Team participation was allowed in most exercises, but teams were rarely explicitly supported. While broad forum discussion provides a helpful tool, the typical competition setting disincentivizes the type of close collaboration between students that provides in-depth learning. Therefore, allowing students to participate in teams can act as a middle ground. Unfortunately, few exercises provided support to help students form teams beyond those fortunate enough to be members of clubs or organizations prior to participation (N=9). picoCTF provided the best example of team support, providing a "team recruitment" channel in their online forum to help students create virtual teams. picoCTF also included a built-in feature for "classrooms" where students could register together in groups with a dedicated scoreboard and resources to help teachers provide personalized feedback and support to their class. Similarly to discussion forums, teams were not supported when stu-

dents were expected to move at their own pace (O=2) or because organizers wanted to target individual-level competition (O=4). As an example of the latter case, the Smash the Stack organizer explained, "We bring people on board to help organize that we see progress through the game rapidly," making individual participation necessary to determine potential new organizers.

6.2.5.2 Inclusive Setup

Finally, we consider each exercise's framing with respect to the extraneous load of each challenge and terminology used throughout the exercise. Extraneous load refers to cognitive challenges required to complete tasks that are not directly related to the concepts being taught [402]. Significant extraneous load could cause students to become stuck and quit because of complicated problems unnecessary to the learning goals of the challenge [359]. The terminology used by organizers can affect less experienced students struggling with new concepts. Reassuring terminology lets students know their struggles are expected and that the solution is not beyond them [359]. Conversely, terminology which demeans newcomers reinforces imposter-like feelings. A "Partial" mark for the *Supportive Terminology* dimension indicates that we found the terminology used to be neutral, i.e., no supportive or demeaning phrasing used.

Students' extraneous load varied widely across exercises. In most exercises, extraneous load was introduced, as students had to determine how to run and reverse engineer a binary compiled for an operating system other than their host OS (e.g., by

installing a virtual machine) (N=11). As another example of extraneous load, Cyber Talents used varying flag formats, adding an unnecessary challenge to determine how to correctly submit flags. Many exercises took steps to reduce this extraneous load, such as providing browser-based tool support (e.g., wireshark, command line, disassembler) (N=6) or an ssh server with required tools installed (N=4). Perhaps the best examples were Microcorruption, which allowed students to perform all required tasks in their browser with a browser-based disassembler and debugger, and Pwn College, which included links to binaries pre-loaded in the BinaryNinja cloud service [3] for advanced reverse engineering support.

Extraneous load is not always bad. While reducing extraneous load is helpful to support learning—especially for less experienced students—we do not suggest these extraneous tasks be avoided in every case. Typically, these additional tasks reflect processes students need to understand and perform in a real-world setting. In fact, most of the organizers who did include extraneous load stated that it was to provide a realistic experience (O=6). Perhaps the most extraneous load is introduced by BIBIFI. BIBIFI introduces a significant extraneous load on students as it requires the development of a medium-sized program with several non-security relevant features. This requires students to spend more time working on extraneous tasks, but is intended to be more representative of a real-world programming scenario. The BIBIFI organizer explained "this is just part of the process of building a real system. So that's a trade off. We decided to do it because it gives people real experience." Therefore, introducing extraneous load should be considered

carefully and in the context of the student's academic progression, in conjunction with decisions discussed previously about connecting to student prior knowledge and organizing knowledge to provide context (Sections 6.2.1.1 and 6.2.2.2).

Most exercises used supportive terminology, but a few marginalized beginners. A plurality of exercises included language in their rules or FAQs offering encouragement (N=17). For example, Vulnhub offered several strategies for dealing with "stuck-ness" and Root-me.org suggested a learning path of concepts to help new students work up to more complicated problems. This supportive terminology, along with tailoring exercise difficulty to student experience (as discussed in Section 6.2.1.1), can improve student confidence and exercise engagement [359]. Unfortunately, some exercises chose to use terminology that marginalizes newer students who might struggle with basic concepts (N=5). This included HackthisSite calling their first challenge the "idiot" challenge and saying "if you can't solve it, don't go crying to anyone because they'll just make fun of you" and Pwn College referring to their easiest challenge level as the "baby" level. The Pwn College organizers explained they chose to use the baby notation as it is common in the CTF culture. Their goal was to help give students a point of reference as they worked on other CTFs. However, they agreed "someone might interpret it negatively and we will consider this point."

6.3 Discussion and Recommendations

Through our review of online hacking exercises and interviews with their organizers, we found that no exercise implemented every pedagogical principle, but there were several exemplary approaches taken by organizers across the exercise landscape. Overall, our analysis found there were a few dimensions where exercises showed room for improvement and others where there are clear tradeoffs between principles.

Our findings regarding current exercise weaknesses can be summarized as follows:

- Organizers did not consider metacognition, and there were are limited number of realistic challenges. This may cause problems for students when trying to apply lessons learned to real situations.
- Exercises lacked clear structure to help students establish a robust, structured knowledge base.
- Inherent technical challenges mean that few exercises provided secure development practice or tailored feedback.
- Exercises frequently gave students the autonomy to choose a personalized path based on experience, but rarely activated prior knowledge to explicitly leverage student experience in the learning process.

We also noticed interesting tradeoffs between principles that should be considered carefully in the design of an exercise.

There is a clear tension between providing realistic challenges and minimizing extraneous load. The more realistic a challenge becomes, the more auxiliary tasks are required (e.g., setup, configuration). However, this is not a binary decision: There are a range of good options, which should be intentionally and explicitly selected to fit learning objectives and students' current experience level. Ideally, exercises should move students from one end of the spectrum (i.e., toy challenge with low extraneous load) to the other (i.e., realistic challenge with more extraneous load) as the student develops expertise so they are prepared to leave the exercise ready to perform similar tasks in the real world.

Community participation can have significant benefits, but can be difficult to manage. Many exercises rely on an active community to provide challenges, help assess challenge difficulty, support less experienced students, and participate in engaging discussion. However, this can create a moderation challenge for organizers as they try to provide overarching structure and context for challenges, make sure new students receive necessary feedback, and ensure a supportive culture in discussions. Therefore, organizers should carefully structure community involvement to gain the benefits of broad participation while maintaining educational benefits. Additionally, from a research perspective, this dynamic indicates that it is not simply enough to understand how exercises themselves run, but also how the community operates. Future work should investigate the forums and online conversations that have grown around exercises.

Competition can get in the way of education. Competition offers a useful motivational tool; however, it also limits avenues of support for less experienced students through collaboration and discussion. We observed that in many cases, the balance was weighted toward competition, simply as an artifact of the initial exercise offering (e.g., in a synchronous setting). Organizers should be conscious of this dynamic, especially after an exercise is no longer part of a live competition. Prior work has shown competitive environments in STEM education appear to negatively effect student experience, especially those of underrepresented populations [33, 144, 247, 260, 359]. Instead, exercises should provide more support for team-based learning and helping new students join the community.

6.3.1 Recommendations

With these findings in mind, we suggest recommendations for exercise organizers and directions for future work.

Support active student engagement in metacognition. Because many exercises simply did not consider metacognition, the first step should be to apply best practices from the learning sciences and education literature within the exercise to prompt students to consider their learning state. For example, students could be prompted to predict the outcome of an exploitation attempt prior to its execution and subsequent success or failure. This foregrounds the student's current mental model of system they are attempting to exploit and the exploit itself's function.

This technique has proved effective in other domains at helping students recognize their own incorrect mental models, allowing them to identify gaps in their own understanding and develop deeper conceptual knowledge [93]. Additionally, once the challenge has been solved, the student could be asked to reflect on why their solution worked. Again, this actively engages the student to probe the depth of their knowledge. This process is similar to after-action discussion commonly used in other expert domains to help students consider how any lessons learned might apply to variation in the situation going forward [122].

Use a graphical syllabus to provide concept structure. A graphical syllabus is a visual representation (e.g., flow chart or diagram) of concepts covered in a course and their relationships [289, 290]. These visualizations in have been shown to help students in other domains process and organize information presented in both traditional and online courses. Exercises could adopt this visual presentation to give students a high-level view of relationships between challenges as well as to provide a guide through progressions of related challenges. Using a graphical syllabus is also appealing because it fits the gaming motif common in CTFs: it aligns with roadmaps commonly used to demonstrate player progression between levels.

Incentivize production of educational elements in community submissions. Community-submitted challenges provide a valuable force multiplier, but because of the number of distinct authors, their organization, and the hints and other information provided to the student, can vary widely. This is expected, since

adding these additional elements can be tedious relative to the more interesting problem of developing the challenge. This is similar to the well-documented lack of documentation in APIs and open-source software development [245, 312, 338]. One possible approach is to apply methods from the crowd-documentation literature (popularized by sites like StackOverflow [376]), including curation activities like voting as well as incentives such as reputation scores [259]. Additionally, because there is already a significant amount of community-generated content available in the form of challenge walkthroughs and blog posts about computer security, future work could also consider developing tools to support improved knowledge discovery from these sources.

Exercise	Rank[1]	**Personalization** (6.2.1.1)				**Utilization** (6.2.1.2)	**Organization** (6.2.2.1)		**Context** (6.2.2.2)					**Actionability** (6.2.3.1)	**Feedback** (6.2.3.2)					**Transfer Learning** (6.2.4.1)	
	α		1	1	.8	1	1	1	–	–	–	1		–	1	–	.9	–		.8	.8
			Age	Education	Experience	Subsequent knowledge	Hierarchical	Problem path	Lecture-based	Project-based	Storyline	Real world exercises		Write secure code	Tailored Feedback	Question support	Hints (scaffolding)	Available solutions		When to use	Why to use
Synchronous																					
gCTF† [149]	1.4	○	○	◐		◐	◐	◐	○	○	●	○		○	○	●	●	◐		○	○
Infosec Institute [205]	14.1	○	○	●		○	○	○	○	○	○	○		○	○	○	●	●		○	○
picoCTF† [296]	149.8	●	●	○		●	●	●	○	○	●	○		○	○	●	●	●		◐	◐
CSAW365 [235]	*1228.1	○	○	●		●	●	○	○	○	○	○		○	○	○	●	◐		○	○
HackEDU [161]	*2014.2	○	○	●		●	●	●	●	○	○	●		◐	○	◐	●	●		●	●
Pwnadventure† [4]	*2364.7	○	○	○		●	◐	●	○	○	●	○		○	○	○	○	◐		○	○
PACTF [308]	*9156.0	●	●	●		●	○	○	○	○	○	○		○	◐	●	●	◐		○	○
Angstrom† [14]	*11708.9	○	●	●		●	●	●	○	○	○	○		○	○	●	●	◐		◐	◐
HXP CTF [201]	–	○	○	●		●	○	○	○	○	○	○		○	○	●	◐	◐		○	○
BIBIFI† [69]	–	○	●	○		●	○	○	○	●	○	◐		●	●	●	○	○		●	●
Pwn College† [365]	–	●	●	●		●	●	●	●	○	○	○		○	◐	◐	◐	●		●	●
GirlsGo CyberStart† [206]	–	●	●	●		●	○	●	●	○	●	○		○	○	●	●	●		●	●

[1] Visit rank for the website, in thousands - Alexa if *, otherwise, using Tranco ranking which is less prone to tampering [241].
† An organizer from this exercise was interviewed or responded via email to our review.

Table 6.1: Results of our pedagogical review of 12 synchronous exercises. Each column indicates whether an exercise implemented the pedagogical dimension fully (●), partially (◐), or not at all (○).

Exercise	Rank[1]	**Support** (6.2.4.2)		**Peer Learning** (6.2.5.1)		**Inclusive Setup** (6.2.5.2)	
		Additional materials	Problem-specific materials	Teams	Group discussion	Low extraneous load	Supportive terminology
	α	–	–	–	.8	1	1
Synchronous							
gCTF† [149]	1.4	○	○	○	○	○	●
Infosec Institute [205]	14.1	●	●	○	○	○	○
picoCTF† [296]	149.8	●	●	●	●	●	●
CSAW365 [235]	*1228.1	●	○	●	○	○	●
HackEDU [161]	*2014.2	●	●	○	○	●	●
Pwnadventure† [4]	*2364.7	○	○	○	○	●	◐
PACTF [308]	*9156.0	○	○	●	●	●	●
Angstrom† [14]	*11708.9	○	○	●	○	●	●
HXP CTF [201]	–	◐	◐	●	●	●	◐
BIBIFI† [69]	–	○	○	●	○	○	◐
Pwn College† [365]	–	●	○	○	○	●	◐
GirlsGo CyberStart† [206]	–	●	●	○	◐	●	●

[1] Visit rank for the website, in thousands - Alexa if *, otherwise, using Tranco ranking which is less prone to tampering [241].
† An organizer from this exercise was interviewed or responded via email to our review.

Table 6.2: Results of our pedagogical review of 12 synchronous exercises. Each column indicates whether an exercise implemented the pedagogical dimension fully (●), partially (◐), or not at all (○).

Exercise	Rank[1]	**Personalization** (6.2.1.1) Age	Education	Experience	**Utilization** (6.2.1.2) Subsequent knowledge	**Organization** (6.2.2.1) Hierarchical	Problem path	**Context** (6.2.2.2) Lecture-based	Project-based	Storyline	Real world exercises	**Actionability** (6.2.3.1) Write secure code	**Feedback** (6.2.3.2) Tailored Feedback	Question support	Hints (scaffolding)	Available solutions	**Transfer Learning** (6.2.4.1) When to use	Why to use
Asynchronous																		
HackTheBox† [46]	97.3	○	○	●	●	○	○	○	○	○	●	○	◐	●	◐	◐	○	○
HackthisSite [167]	105.1	○	○	●	●	○	●	○	○	●	○	○	◐	●	●	◐	○	○
OverTheWire [304]	151.3	○	○	◐	●	○	●	○	○	○	○	○	○	●	●	◐	○	○
Root-me.org† [266]	172.7	○	●	●	●	●	◐	○	○	◐	◐	○	◐	●	◐	●	◐	◐
Vulnhub† [142]	175.8	○	○	◐	●	○	◐	○	○	●	◐	○	◐	●	◐	●	◐	○
Hacker101 [162]	330.4	○	○	●	◐	○	○	○	○	○	○	○	○	●	●	◐	○	○
Hellbound Hackers [165]	432.8	○	○	●	●	○	●	○	○	○	●	●	◐	●	●	◐	◐	○
Smash the Stack† [392]	966.1	○	○	○	●	●	●	○	○	○	◐	○	○	●	●	◐	◐	◐
Microcorruption [156]	*378.8	○	○	○	●	●	●	○	○	●	○	○	○	○	◐	◐	◐	○
Pwnable [375]	*515.4	○	○	◐	●	○	○	◐	○	○	○	◐	○	●	●	●	◐	◐
Cyber Talents [98]	*528.0	○	○	●	●	○	○	○	○	○	○	○	◐	○	○	◐	○	○
XSS-Game† [150]	*626.1	○	◐	○	●	●	●	●	○	●	○	○	◐	●	●	◐	◐	◐
Backdoor [357]	*949.1	○	○	●	●	○	○	○	○	○	○	○	○	●	●	◐	◐	◐
Crackmes.one† [350]	*1011.4	○	○	●	●	○	◐	○	○	○	◐	○	○	●	○	●	●	●
CTFlearn [95]	*1267.0	○	○	●	●	○	○	○	○	○	○	○	○	●	◐	●	○	○
HackerTest [166]	*1254.5	○	○	○	●	●	●	○	○	●	○	○	○	●	○	◐	◐	◐
Mr. Code† [217]	*4570.2	●	○	●	●	●	●	●	○	●	○	○	●	○	○	○	○	○
IO Wargame [286]	*7168.8	○	○	○	●	○	●	○	○	○	○	○	◐	●	●	◐	●	◐

[1] Visit rank for the website, in thousands - Alexa if *, otherwise, using Tranco ranking which is less prone to tampering [241].
† An organizer from this exercise was interviewed or responded via email to our review.

Table 6.3: Results of our pedagogical review of 18 asynchronous exercises. Each column indicates whether an exercise implemented the pedagogical dimension fully (●), partially (◐), or not at all (○).

Exercise	Rank[1]	**Support** (6.2.4.2) Additional materials	Problem-specific materials	**Peer Learning** (6.2.5.1) Teams	Group discussion	**Inclusive Setup** (6.2.5.2) Low extraneous load	Supportive terminology
Asynchronous							
HackTheBox† [46]	97.3	◐	○	●	●	○	◐
HackthisSite [167]	105.1	◐	○	○	●	○	◐
OverTheWire [304]	151.3	◐	◐	○	●	●	●
Root-me.org† [266]	172.7	●	●	●	●	○	●
Vulnhub† [142]	175.8	●	○	○	◐	○	●
Hacker101 [162]	330.4	●	○	●	●	○	●
Hellbound Hackers [165]	432.8	●	◐	○	●	○	◐
Smash the Stack† [392]	966.1	●	●	○	●	●	○
Microcorruption [156]	*378.8	○	○	○	○	●	◐
Pwnable [375]	*515.4	○	○	○	●	●	○
Cyber Talents [98]	*528.0	○	○	○	○	○	◐
XSS-Game† [150]	*626.1	○	○	○	◐	●	●
Backdoor [357]	*949.1	●	◐	○	○	○	●
Crackmes.one† [350]	*1011.4	○	○	○	●	○	●
CTFlearn [95]	*1267.0	○	○	○	●	●	●
HackerTest [166]	*1254.5	○	○	○	○	●	◐
Mr. Code† [217]	*4570.2	●	○	○	○	●	●
IO Wargame [286]	*7168.8	○	○	○	●	●	●

[1] Visit rank for the website, in thousands - Alexa if *, otherwise, using Tranco ranking which is less prone to tampering [241].
† An organizer from this exercise was interviewed or responded via email to our review.

Table 6.4: Results of our pedagogical review of 18 asynchronous exercises. Each column indicates whether an exercise implemented the pedagogical dimension fully (●), partially (◐), or not at all (○).

Chapter 7: Building and Validating a Scale for Secure Software Development Self-Efficacy

In addition to many attempts to improve secure development education to expose non-experts to a variety of vulnerabilities [70, 183, 211, 244, 347], it is also necessary to develop a method to measure skills before and after the intervention to test their effectiveness. Additionally, a metric for secure development skill is needed for use as a covariate when evaluating new security tools [209, 231, 351, 368, 372, 393, 419, 428], documentation [8], and APIs [7, 202, 427]. Without such a metric, the experimenter cannot control for participant skill, which may confound results.

Previous studies have sought to measure secure development skills through various methods [7, 44, 76, 77, 83, 110, 112, 135, 208, 242, 282–284, 302, 327, 346, 347, 423]. In most cases, researchers have participants identify and exploit vulnerabilities in sample programs or write small programs with security-critical functionality. For example, while studying security API misuses, Oliveira et al. asked participants to identify common mistakes in security "puzzles" (i.e., code snippets) [302]. In a more intensive evaluation, Ruef et al. asked Build It, Break It, Fix It competitors to write medium size programs with several security requirements during a week-long "build" round [347]. Participants then evaluate each others' submissions through

vulnerability-demonstrating exploits in another week-long "break" round. These assessments provide valuable insights into actual secure-development skill, but are very cumbersome. Other work asked participants to rate their security skill on a single Likert scale or counted the vulnerabilities reported in public code artifacts. While these are both require less participant time, they are also noisy, as participants can have difficulty assessing their abilities and there can be several confounding factors impacting vulnerability counts.

To remedy this shortfall, I propose an alternative approach to measuring secure-development skill: a validated scale. Human behavior researchers regularly develop scales to "measure elusive phenomena that cannot be observed directly" due to cost or complexity [32]. Specifically, I propose measuring developers' secure-development *self-efficacy* — belief in one's ability to successfully perform a task — which correlates with actual skill in other contexts [32]. This scale would measure developers' belief in their ability to complete secure-development tasks, such as identifying security problems during software design or employing secure programming languages.

In this chapter [1], I present the work of my colleagues and I to develop and evaluate such a scale: the Secure Software Development Self-Efficacy Scale (SSD-SES). We followed Netemeyer et al.'s 4-step scale creation process [285]:

1. **Construct Definition and Content Domain:** Clearly identifying the targeted construct's scope (i.e., the underlying idea). We focus on tasks related to secure code production.

[1] Published as [410]

2. **Generating and Judging Measurement Items:** Defining an initial pool of candidate scale questions (*items*) and ensuring they are relevant to the construct and understandable by respondents. We generated items by reviewing five popular secure-development frameworks. We judge questions based on reviews from security experts and developers.

3. **Developing and Refining the Scale:** Using Exploratory Factor Analysis (EFA) to identify an underlying factor structure (i.e., any sub-components of the targeted construct and their associated questions). We also refine the item set to its most efficient form (i.e., only including items with sufficient variance among respondents), while maintaining reliability (i.e., whether the scale consistently measures the construct).

4. **Finalizing the Scale:** We use Confirmatory Factor Analysis (CFA) to confirm the previously identified underlying factor structure holds, maintaining reliability, with a new sample.

All our procedures were approved by the University of Maryland's institutional review board. We evaluated SSD-SES with 311 software developers and 22 security experts. SSD-SES consists of 15 items measuring two underlying factors: vulnerability discovery and mitigation, and security communication.

7.1 Item Generation and Judgment

The first step in scale development is *construct definition*: scoping what the scale will and won't cover. As SSD-SES's goal is to measure software developers'

belief in their ability to perform secure development tasks, we focus only on tasks related to the production of secure code. That is, we do not include tasks from parts of the software development lifecycle such as deployment, maintenance, or monitoring. We also restrict our tasks to those prescribed by widely accepted secure-development frameworks or experienced security experts.

7.1.1 Initial Item Generation

The second step is generating a set of candidate *items* (questions). The goal is to thoroughly survey the construct domain and build an extensive possible item pool [285], to be narrowed in later steps. We chose initial items by analyzing four popular secure-development frameworks: NIST's National Initiative for Cybersecurity Education (NICE) framework [287], the Building Security In Maturity Model (BSIMM) [263], the Open Web Application Security Project (OWASP) Software Assurance Maturity Model (OSAMM) [74], and Microsoft's Security Development Lifecycle (SDL) [273].

A colleague and I independently reviewed each framework, identified a set of prescribed tasks, and selected tasks focused on secure code production. We then met to combine lists. Because best practice recommends a conservative approach to initial item generation (i.e., including any possibly related items) [285], if a task was identified by either myself or the other researcher it was included in the initial set. Finally, we merged tasks from different frameworks considered identical (e.g., phrased differently or using different terminology, but expressing the same idea),

again conservatively.

This process produced 57 unique software-development-specific tasks mentioned in at least one framework. The full task set (with sources) can be found in Appendix B.7, Tables B.10-B.12. These tasks can be divided into six categories: determining security requirements (A1-11), identifying attack vectors (A12-14), identifying vulnerabilities (A15-22), implementing mitigations to prevent or remedy vulnerabilities (A23-37), testing of security requirements (A38-42), and effectively communicating about security with peers, leadership, and security experts (A43-57).

7.1.2 Content Review

To ensure the identified tasks cover the full range of the domain (*content validity*) and that the task wording was understandable to software developers (*face validity*), we surveyed 22 secure-software-development experts and 8 developers.

Expert review. We asked security experts to review our initial 57 tasks (Tables B.10-B.12) and rate them on a 4-point Likert-scale ranging from *Definitely not a secure development task* to *Definitely a secure development task*. Respondents also had an *Unsure* option if the wording was confusing or they could not clearly delineate the task's appropriateness. We also asked respondents whether the task phrasing was unclear or confusing, and to explain any confusion in free text. We concluded by asking experts to list any missing tasks. Our expert review survey text is given in our supplementary material.

We recruited security experts from a convenience sample of the authors' professional contacts (N=7) and members of NIST's Software and Supply Chain Assurance Forum [297] (N=16). To ensure we received expert opinions, we only considered respondents who self-reported 10 or more years of experience. After each response, we added any suggested tasks for review by subsequent experts. We stopped recruiting additional experts when no new tasks were suggested and task appropriateness responses remained stable (e.g., no significant difference in results when adding the last 10 responses), the suggested stopping criteria for free-listing exercises [422]. The 22 secure-development experts had 20 years of experience on average, and 81% also held a graduate degree. Our expert population matched prior security expert studies whose experts had at least 10 years experience [173, 174]. Li et al. used participant titles as an alternate expertise indicator [248]. Our participants also met this condition, reporting senior job titles such as Chief of Development, Cybersecurity Technical Fellow, and Product Security Lead.

We made several changes to our task list based on expert feedback. First, ten tasks were considered inappropriate—*Probably not* or *Definitely not a secure development task*—by 80% of experts. For several tasks, our experts believed it is not the developers' role to determine the balance security and performance costs, but instead the job of customers or leadership (reworded A3 and A7; removed A10, A50, A51, A52, A53). Similarly, our experts stated developers should not be expected to research attacker techniques, but instead get this information from security experts (reworded A5; removed A55). Finally, our experts indicated code signing is not part of secure code production, but instead its deployment, which is out of scope for our

construct definition (removed A11).

Additionally, we revised several tasks' wording. Most significantly, we replaced "program" with "system" to match the modular approach to design common in industry (A1, A2, A4, A12, A13, A20, A29, A35, A45, A46). Other changes included using more common developer terminology (A2), focusing on threats from malicious actors (as opposed to natural disasters) (A6), making security explicit (A4), adding clarifying examples (A15, A29), and rephrasing statements to improve readability (A20, A21, A41, A42).

Finally, we added six tasks. Several experts recommended tasks for identifying and using secure programming languages and libraries (B15, B16, B26, B27, B28; shown in Tables 7.1-7.3). One expert also suggested adding a task for correctly implementing authorization protocols (B34), as it represents a distinct access-control component (compared to authentication).

Multiple experts suggested the Software Assurance Forum for Excellence in Code (SAFECode) Fundamental Practices for Secure Software Development framework [337] as an additional task source, so we repeated the framework review process for SAFECode. While no task list or task categories changes were made, it provided further support for tasks already included.

Developer pilot. Next, to ensure our target population could easily understand each item, we piloted the post-expert-review tasks with eight developers. First, we reframed each task as an *"I can"* statement regarding the developer's confidence in performing the task. We then asked them to indicate their confidence using a 5-point

Likert-scale from *"I am not confident at all"* to *"I am absolutely confident."* We also provided a *"Do not understand"* option if the respondent did not understand the task's meaning. Our full survey text is given in our supplementary material's main survey section.

We recruited a convenience sample of the researchers' professional contacts, chosen to represent varying experience levels. Participants were asked to "think aloud" as they responded to each question. We updated the questions after each pilot, eventually reaching the final set given in Tables 7.1-7.3. Specifically, we made the following changes: we reworded A48 to make it clear we were asking about writing understandable security-error messages, updated A8 to indicate we were asking about code the developer has themself written as opposed to a library function, and replaced the term "boundary cases" with "edge cases" in A38 to use the more common terminology.

7.2 Refining the Scale

To trim our item set and determine the underlying factor structure, we recruited 157 developers and performed EFA. This section describes the methods used and our analysis results.

7.2.1 Recruitment

From September 2018 to July 2019, we recruited participants using several methods to broadly sample the developer population. First, we contacted software-

development-related groups' leadership. This included popular Meetup.com [268] and LinkedIn [251] groups, regional ACM chapters [132], and the researchers' personal contacts at large development companies. We asked each contact to share study details with their organization's members and their colleagues. Prior work has found relative success partnering with organizational leadership in this manner, adding credibility to recruitment messages [413]. We also posted messages on relevant online forums such as Reddit and Slack channels. Dietrich et al. showed this method's usefulness with technology professionals, as participants are reached in a more natural setting and are more likely to be receptive [103]. Finally, we recruited developers directly through the freelancing platform Upwork [400] and the research-participant recruitment site Prolific [315]. Because of our broad recruitment process, we do not include respondents who reported less than one year of development experience.

We also varied the study's compensation method. Prior work suggests using a mix of incentives increases participant diversity [198]. Participants recruited through organizational contacts and online forums were recruited in two waves. In the first wave, we did not advertise or provide any compensation for participation. In the second, participants were entered into a lottery for one of 10 $20 Amazon gift cards. Participants recruited through Upwork and Prolific were paid $8 each.

7.2.2 Survey design

Participants were shown the 58 "I can" statements in Tables 7.1 and **??** and asked to rate each on a 5-point Likert-scale from *"I am not confident at all"* to *"I am absolutely confident"* or indicate they *"Do not understand"*. Tasks were presented in random order to prevent ordering effects. At the survey's conclusion, participants were asked to indicate their software development skill level on a 5-point Likert scale ranging from "Novice (limited experience)" to "Expert (recognized authority)," their years of software development experience, and their average time spent daily performing software development tasks, along with other demographic questions. Our full survey text is given in our supplementary materials' main survey section.

We were concerned some participants might overrate their secure development skill to portray appear more socially acceptable [94]. To avoid social desirability bias, our study utilized deception [285]. That is, during recruitment and throughout the survey, participants were told our goal was to measure *general* software development self-efficacy. At the survey's conclusion, participants were told the study's true nature and were allowed withdraw and have their response deleted.

7.2.3 Demographics

We received 181 responses, but 7 (4%) did not have one year of development experience and 9 withdrew (5%) after learning the study's true purpose. We removed eight responses (4%) considered careless based on abnormally short response times [267]. We set the cutoff at less than five minutes based on an obvious threshold

in the data.

The final 157 participants' development experience ranged from 1 to 45 years ($\mu = 8.68$, $\sigma = 9.46$). Our participants were predominantly male (88%), young (50% below 30 and 83% below 40), educated (77% held a bachelor's degree, 26% held a graduate degree), and white (51%) or Asian (29%). Our participants' demographics are similar to those found by prior large-scale secure development studies [7–9,24,30, 100, 202, 282, 302, 347, 361, 428] and other general software development surveys [31, 57, 87, 188, 229, 238, 377, 407]. These prior studies' developers' mean development experience was between 5 and 16.4 years (mean of means = 9.49 years), and they were mostly male (between 73% and 98%; mean = 89%) and young (mean age between 19.4 and 32.9; mean of means = 28.52), matching our sample. Half of our participants reported incomes between \$15K-\$100K, which matches developer income levels in [302], the only other prior work we found reporting participant income. Note that due to the variance in demographics reported in each paper, these are the only items we could clearly compare to.

Choosing an appropriate sample size can be complicated. Prior work suggests basing sample size on the number of items [28, 49, 67, 152, 169, 233, 250, 261, 293] with a minimum of 100 to 200 participants required [88, 89, 152, 160, 169, 250, 255]. For example, Hair et al. recommend 5 participants per item [169]. However, empirical evaluations of scales' component analysis stability (i.e., whether the factor structure identified varied) with various sample sizes has not found evidence to support the sample size-to-item ratio [19, 34, 406]. Instead, prior work shows factor loading (i.e., the magnitude of correlation between items and their associated factor), the absolute

sample size, and the number of variables associated with each factor have the most significant effect on component analysis stability [157]. Therefore, Guadagnoli and Velicer recommend 150 participants as sufficient if factors are made up of high numbers of items (10 or 12), even with low loadings ($l < 0.40$) [157]. We believed, because of the large number of items tested, we would be likely to meet this standard, so we targeted about 150 qualifying participants. In fact (as will be discussed below), our data met Guadagnoli's and Velicer's more conservative standard: four or more variables per component having loadings over 0.60, which they found sufficient to assess underlying component structures "whatever the sample size used." We therefore conclude our sample size was sufficient.

7.2.4 Issues with initial items

To refine our scale, we first checked for several potential issues in our initial items. First, we observed our scale's internal consistency was very high (Cronbach's $\alpha = 0.98$), indicating our items were closely related. Next, we checked each item's item-total correlation, which indicates how *discriminant* the item is (i.e., participants who score high on the item are more likely to score high on the full scale). Items with low item-total correlation (< 0.2) should be excluded because they do not adequately reflect the scale [120]. We did not observe any questions with low item-total correlation.

We next looked for ceiling or floor effects: tight response groupings at either extremes of the Likert scale (e.g., $4.0 < \mu < 5.0$ and $\sigma < 1$). Since scales are

designed to measure differences between participants, individual questions need to exhibit adequate variance; if everyone responds similarly, the item has limited utility. We found no items with this effect.

Finally, it is important to ensure our target population understands each item. If a respondent if confused by the terminology used or question phrasing, their response is framed by a misconception and not reflective of the underlying construct. We removed 13 items to which $> 5\%$ of participants responded "Do not understand" or simply skipped. In many cases (B12, B19, B28, B30, and B33), the confusion seemed to stem from using less common security terminology such as *fuzzing* or *non-repudiation*. Because the scale should usable by developers of all levels of security knowledge, we removed these items. Similarly, 12.1% of participants found B34 confusing, likely because it asks whether participants can use security services provided by their enterprise. As not all developers work in an enterprise setting which offers these services—including many of the freelancers we recruited—this item also does not meet our goal of producing a measure for all developers.

7.2.5 Factor analysis

With the remaining 45 items, we set out to identify the scale's underlying factor structure. Here, we define factors as our construct's sub-components. A construct can have one component, indicating the scale's items measure it directly, or multiple components, where the scale can be broken into component sub-scales and together their scores reflect the construct. Because these factors are latent, they

can not, by definition, be measured directly. Instead, we must first determine our construct's number of factors and which items best describe each factor, i.e., the underlying factor structure.

Prior to attempting to identify our factor structure, we checked whether our data actually measured common factors and were correlated (prerequisites to establishing the number of factors and their structure). According to Bartlett's test of sphericity ($\chi^2 = 4833.35, p < 0.001$) [373] and the Measure of Sampling Adequacy (0.936) [72], we confirmed our data met these goals.

Next, to identify the factor structure, we performed an exploratory Principal Component Analysis (PCA). PCA is a data summarization method that transforms item responses such that the first dimension (or *component*) explains as much variance in the original data as possible, with each subsequent and orthogonal component explaining as much of the remaining variance as possible. These components represent the scale's underlying factors. To produce an efficient factor structure (i.e., one identifying the most variation with the least set of items), we only retain the top components.

Since there is no standard method for deciding the number of retained components, we relied on several and followed the most common recommendation. This consensus protocol accounts for each method's strengths and weaknesses. First, we calculated each components' eigenvalues and selected those with eigenvalues > 1.0 according to the Kaiser criterion [152]. Next, we determined optimal coordinates by fitting a line to the smallest eigenvalues with a linear regression and identifying where our eigenvalues diverge [330]. We also performed a parallel analysis by generating

random data, calculating the associated eigenvalues, and retaining any eigenvalues whose value was greater than the random data's eigenvalues [194]. Finally, we determined the acceleration factor by looking for the point where our eigenvalues changed dramatically [330]. In these analyses, the Kaiser criterion recommended six components, the optimal coordinates and parallel analysis both recommended two, and the acceleration factor recommended one. Therefore, we retained two components.

To determine which factors each item associated most with (i.e., which it loads on), we rotated responses [153]. There are multiple possible rotation types, which can be divided into orthogonal (e.g., varimax) or oblique (e.g., direct oblimin). Orthogonal rotations are appropriate when the factors are not expected to be correlated and an oblique rotation is appropriate otherwise [153]. Because we did not know whether our factors were correlated, we followed the recommendation of Tabachnick and Fiddell who suggest first using an oblique rotation (in our case a direct oblimin), calculating the correlation of the identified sub-scales associated with each factor, and switching to an orthogonal rotation if correlations do not exceed 0.32, indicating 10% (or more) overlap in variance among factors [388]. We found our factors were correlated (0.64) and maintained the direct oblimin rotation.

After rotating items, we selected which ones to retain, using three inclusion criteria. First, we only considered an item as loading on a factor if its loading exceeded 0.5, indicating significant association with the underlying factor [285]. Next, we applied Saucier's criterion, only considering an item to load on a factor if its loading exceeded twice the loading on any other factor. This ensures variance in item responses maps to changes in the associated factor. Finally, we chose to

remove items where the item variance accounted for by all the retained factors (its communality) was less than 0.4, as recommended by Fabrigar et al. [121], as this tends to indicate low item reliability. This led us to remove 27 more items. We reran PCA on the remaining items and found that the two retained components predicted more than 56.9% of variance. Notably, the first factor accounts for a majority of the scale's variance (47.6%), with the second factor accounting for 9.3% of variance. The rotated factor loadings are given in Table 7.4. Note, because all our factor loadings are above 0.60, this confirms the sufficiency of our 157 developer sample [157].

7.2.6 Reliability

To confirm our remaining items maintained their internal reliability, we first computed Cronbach's α for the full scale ($\alpha = 0.936$) and each sub-scale ($\alpha = 0.907$, 0.876). We found that our data met McKinley et al.'s suggested threshold that a multi-component scale is reliable if α exceeds 0.6 and a majority of sub-scale αs exceed 0.7 [265].

Next, we tested item-total correlation using Pearson correlation between each item and the average of all other items in the same sub-scale. All items exceeded Everritt's 0.2 threshold [120]. Finally, we observed each sub-scales' mean item-total correlation (0.520 and 0.440, respectively) exceeded 0.30, which is considered "exemplary" [341]. Based on these measures, we confirmed our reduced scale had high reliability.

7.3 Finalizing the Scale

We next conducted an additional round of surveys from July to September 2019 with the 18 items remaining after EFA (shown in Table 7.1). In this step, we tested whether the identified two-factor structure was maintained, remaining reliable, with a different participant pool.

7.3.1 Survey Design

Respondents were again asked to respond to the 18 items on a 5-point Likert scale from "I am not at all confident" to "I am absolutely confident." We removed the "Do not understand" option, as we had sufficiently established item face validity.

It is important to confirm our scale measures the targeted construct by testing whether responses match other theoretically relevant measures. To test whether SSD-SES converges with measures we expect it to relate with (*convergent validity*), while being distinct from other, similar scales (*discriminant validity*), we performed Pearson's correlation tests with four related scales. We similarly use Pearson's correlation to compare to two additional, well-established psychometric scales to understand how participant psychological characteristics relate to secure-development self-efficacy. All p-values reported in this section are corrected using a Bonferroni-Holm correction to account for multiple testing [191]. To avoid overburdening participants, we randomly present two of the six additional scales (described in detail below) to each participant.

7.3.2 Recruitment

We used the same recruitment process as the prior step, targeting 120 responses. Prior work suggests 100 participants are sufficient for confirmatory factor analysis [45]. We targeted 20 more participants to have 40 participants complete each additional psychometric scale, giving sufficient power (> 0.80) to identify $>$ 42.5% correlation with SSD-SES [86].

Due to the relative difficulty of recruiting through organizations and online forums, we predominantly relied on Upwork's freelancing service and Prolific in this step. As developers on these platforms tend to be less experienced, respondents from these sites first completed a pre-screening survey, asking them to report years of software development experience and average time spent daily on development tasks. We then invited screened respondents to the full survey using a quota-sampling method. Our quotas were chosen to match developer experience ranges from StackOverflow's most recent developer demographics survey [377]. Specifically, we sought to survey 14 (11.4%) inexperienced developers (1-2 years), 70 (58.4%) moderately experienced developers (3-11 years), and 36 (30.2%) experienced developers ($>$ 11 years). We concluded recruitment after reaching each quota, though we were not able to maintain the desired percentages due to the unpredictable nature of responses from organizational contacts.

7.3.3 Demographics

A total of 162 developers responded in this round, but 2 participants (1%) had less than a year of software-development experience, 6 chose to withdraw (4%), and 8 (5%) responses were considered careless (i.e., completed in less than 5 minutes [267]). The remaining 146 participants' development experience was split between our three quota ranges as follows: 10.3% inexperienced, 65.1% moderately experienced, and 24.6% experienced ($\mu = 9.79$, $\sigma = 10.80$). Our participants were predominantly male (91%), young (48% below 30 and 77% below 40), educated (57% held a bachelor's degree, 26% held a graduate degree), and white (77%) or Asian (9%). Participant median income was between \$1K-\$75K. Again, this matches broader developer demographics [7–9,24,30,31,57,87,100,188,202,229,238,282,302,347,361,377,407,428].

7.3.4 Factor Analysis

To determine whether the latent factor structure identified previously held with our new population, we repeated the PCA procedure using a direct oblimin rotation. We observed three items either no longer sufficiently loaded on their original factor (i.e., B8's loading on F2 reduced to 0.39) or switched factors (i.e., B31 and B32 switched to load on F1). Because these items did not behave reliably across multiple samples, they were removed [285]. The remaining items demonstrated internal consistency with a Cronbach's α of 0.92 (sub-scale αs were 0.90 and 0.88, respectively). The 15 items and their associated latent factors are given in Table 7.5 with their mean responses, factor loadings, and item-total correlations within their sub-scales.

All the remaining items and their sub-scales behaved appropriately according to the variability and reliability metrics given in the prior section.

The remaining sub-scale items represented two distinct themes: vulnerability identification and mitigation tasks and security communication tasks. Again, the first factor accounts for a majority of the scale's variance (48.1%), and the security communication sub-scale accounts for 11.1% of variance.

While EFA is useful for determining a possible underlying factor structure, it is not able to assess that structure's goodness-of-fit onto the data with respect to other possible structures [254]. Therefore, we also performed a Confirmatory Factor Analysis (CFA), which confirms our prior analyses have been conducted thoroughly and appropriately [189]. CFA is a type of structural-equations analysis assessing rival models' goodness-of-fit. Specifically, we compare a null model with all items loading on separate factors, a single common factor model, and our multi-factor model [212].

Our model demonstrated sufficient goodness-of-fit with its χ^2 (176.38) below the conservative limit of twice the its degrees of freedom (DoF = 89) [66]. Using ANOVA comparisons, we found our model fit better than the null ($\chi^2 = 1257.34$, $p < 0.001$) and the single-factor model ($\chi^2 = 317.78$, $p < 0.001$).

We also calculated several other goodness-of-fit metrics. First, we determined the Comparative Fit Index (CFI), which measures the model's fit relative to a more restrictive baseline model [41], and the Tucker-Lewis Index (TLI), a more conservative version of CFI, penalizing overly complex models [42]. Our model performed well in both (CFI = 0.92, TLI = 0.91), with scores over the recommended 0.90

threshold [285]. Next, we calculated the Standardized Root Mean Square Residual (SRMR), an absolute measure of fit calculating the difference between observed and predicted correlation [399]. Our model demonstrated a sufficient SRMR of 0.054—a value below 0.08 is considered good fit [399]. Finally, we calculated the Root Mean Square Error of Approximation (RMSEA), which measures how well the model reproduces item covariances, instead of a baseline model comparison. Our model demonstrated a "moderate" fit with a RMSEA of 0.082 [256].

7.3.5 Reliability

To measure the sub-scales' internal consistency, we calculate their composite reliability [134]. Composite reliability offers a more accurate view of reliability over Cronbach's α, which makes several potentially inaccurate assumptions, such as considering factor loadings and error variances equal [334]. Instead, composite reliability considers factor loadings from CFA, measuring the latent factors' reliability instead of their individual items. We found both sub-scales' composite reliability (0.90 and 0.87, respectively) exceeded the recommended threshold of 0.60, indicating they are internally consistent [29].

7.3.6 Convergent Validity

Prior work suggests the ability to identify vulnerabilities is influenced by participants' understanding of the development environment (e.g., the programming language and libraries used) and level of security experience [413]. We next consider

how well our scale correlates with each of these concepts.

Secure-development self-efficacy is related to general development self-efficacy. To measure respondents' software development skill, we utilized the Computer Programing Self-Efficacy Scale (CPSES), a measure of respondents' belief in their ability to produce working programs meeting functionality requirements [398]. CPSES scores were statistically significantly correlated with both SSD-SES sub-scales ($\rho = 0.528$, $p = .001$ and $\rho = 0.627$, $p < 0.001$, respectively).

To measure security experience, we asked if participants had received security training, if they had found a vulnerability or had one found in their code, and how often they communicate with security experts. We estimated the relationship of security experience and secure-development self-efficacy with a poisson regression (appropriate for count data [63]). For each sub-scale, our initial regression model included each security experience response and the participants' software development experience as independent variables. To avoid overfitting, we tested all combinations of the independent variables, selecting the model with minimum Bayesian Information Criteria [329], as is standard.

Table 7.7a shows the vulnerability identification and mitigation regression model and Table 7.7b gives the security communication regression model. Base cases were selected to represent the medium experience level to allow clearer comparisons. Variable levels are presented in decreasing experience order. The log estimate (LE) column gives the variable's observed effect. For categorical variables, the LE is the expected relative change in SSD-SES sub-scale score when moving to

the given variable level from the base case. The LE for each base case is definitionally 1.0. We also give the 95% confidence interval for the log estimate (CI) and the associated p-value.

Secure-development self-efficacy increases with security experience. As expected, both SSD-SES sub-scale scores increase with security experience. Specifically, participants who had never found a vulnerability (LE = 0.82, p $<$ 0.001) or never worked with a security expert (OR = 0.81, p $<$ 0.001) had less belief in their ability to find and mitigate security vulnerabilities. Further, those who have found multiple vulnerabilities (LE = 1.08, p = 0.49) or worked with multiple security experts (LE = 1.09, p = 0.024) had even higher self-efficacy.

We observed nearly the same significant trends in the security communication sub-scale. However, more than one experience finding a vulnerability did not show a significant increase over a single vulnerability identification experience (LE = 1.08, p = 0.112). Additionally, we found experienced developers ($>$ 11 years experience) were more likely to believe they could effectively discuss security (LE = 1.11, p = 0.017).

7.3.7 Discriminant Validity

To confirm that SSD-SES in fact measures a new construct, we compared it to two end-user security behavior scales and a general self-efficacy scale. First, we tested whether either sub-scale correlated with the Security Intention Behavior Scale (SeBIS) [115], which measures end-user intention to perform a variety of security

behaviors, and the End-User Security Attitudes (SA-6) scale [124], which measures end-user attitudes toward common security behaviors. Correlations between each of these scales and SSD-SES's two sub-scales are given in Table 7.6. We did not observe any significant correlation between either scale and SSD-SES.

Next, we tested the correlation between SSD-SES and the General Self-Efficacy Scale (GSE), which assesses the "belief that one can perform a novel or difficult tasks, or cope with adversity—in various domains of human functioning [356]." This comparison tested whether SSD-SES simply measured respondents' belief in themselves as opposed to a domain-specific belief. Again, we did not observe any significant correlation between GSE and SSD-SES.

The lack of significant correlations with SeBIS, SA-6, or GSE indicates SSD-SES measures a distinct underlying construct.

7.3.8 Relationship with Psychological Constructs

Finally, we included two well-established psychometric measures to understand how participants psychological characteristic relate to SSD-SES scores: the Need for Cognition (NFC) scale and the General Decision-Making Scale (GDMS). Correlations between NFC and GDMS and SSD-SES's two sub-scales are given in Table 7.6.

Curious developers believed more in their ability to identify and mitigate vulnerabilities. NFC measures intellectual curiosity (i.e., the tendency to engage in and enjoy thinking) [60]. We found that NFC significantly correlated with the

vulnerability identification and mitigation sub-scale ($\rho = 0.464$, $p = 0.007$), but not the security communication sub-scale ($\rho = 0.183$, $p = 0.337$). This suggests developers who are more curious are more likely to feel confident in their ability to search for, identify, and mitigate vulnerabilities. This finding corroborates prior work, which observed developers who were more open or curious were more likely to find vulnerabilities in secure-development puzzles [302].

We also compared SSD-SES to GDMS, which assesses how individuals approach decisions with respect to five decision styles. As secure development requires complex planning and decision-making, our goal was to test whether any style correlated with better secure-development self-efficacy. However, we did not observe any statistically significant correlation.

7.4 Discussion and Limitations

Our final 15-item scale measures two distinct underlying factors: vulnerability identification and mitigation as well as security communication. Through our scale development process we observed SSD-SES demonstrate construct validity, internal consistency and reliability, goodness-of-fit, and convergent and discriminant validity. We found SSD-SES correlated with general programming self-efficacy, security experience, and intellectual curiosity, as expected. In this section, we discuss our study's limitations and the need for future work as well as how SSD-SES can be employed by researchers and practitioners.

7.4.1 Limitations

Software developer recruitment is challenging [282], requiring significant effort to reach the sample used in this study. While we sought to recruit a diverse sample with respect to development experience and we observed similar demographics to the global developer population, it is possible our recruiting methods do not fully represent the broader population. Future work should consider methods to recruit participants not active in development organizations, online forums, or freelancing and research recruitment platforms.

In the final round of recruitment, we performed quota sampling based on participants' years of experience. We also randomly assigned two of six external scales to each participant, without consideration for quota sampling. As a result, the experience distributions for these external scales were not entirely consistent: fewer than expected participants in the inexperienced group were assigned CPSES (2% as opposed to 10% of the total sample) and fewer of the experienced group were assigned GES (14% as opposed to 25%). The supplementary materials give further details. We do not expect this fairly small inconsistency to have a large effect on the results, particularly as we observed almost no significant effects related to experience.

While we believe our approach to item generation and question trimming thoroughly reviews secure development concepts, we do not argue our scale covers all possible factors. Rather, our items and factors produce meaningful and reliable results and pass relevant validity checks. It is possible some factors are not included.

We leave future work to investigate additional factors, expanding on our current findings.

Though we found SSD-SES reliable and valid according to several measures, additional testing is necessary. First, as we have only shown correlation between factors and other psychometric measures, we cannot make assessments of causal relationships; as such, we cannot yet create a predictive model to target interventions at specific self-efficacy components.

Most importantly, further work is necessary to determine if SSD-SES measures *actual* secure development skill improvement. In future work, we plan to administer SSD-SES as a pre- and post-test for a hands-on security training course, allowing us to assess improvements and correlate with course grades.

Finally, technology (and accordingly, secure development practice) changes over time. We designed SSD-SES to describe general principles we believe should remain relatively static, but as with any scale SSD-SES should be occasionally revisited and refreshed as needed.

7.4.2 Using SSD-SES

Our creation and validation of a lightweight secure software development self-efficacy scale presents a valuable resource for a variety of purposes. Below, we suggest possible uses.

SSD-SES for testing educational interventions. The initial impetus for creating SSD-SES was to measure the value of an educational intervention, such as a

capture-the-flag exercise [77]. By applying SSD-SES before and after administering training, a researcher can observe changes in secure-development self-efficacy, providing feedback for the improvement and comparison of interventions.

To emphasize this benefit, we note that we did not observe a significant relationship between SSD-SES and participants' self-reported security training in our regression. While we would expect training to improve self-efficacy, our results suggest in practice, it does not consistently do so. Therefore, future work in security education is necessary to identify the right training to produce improved outcomes.

SSD-SES as a covariate. SSD-SES also provides a useful option for software-developer studies where the researcher may want to control for participants' secure-development skill. For example, in a usability study of security APIs, it would be beneficial to include SSD-SES as a covariate to ensure differences in outcomes between participants occur because of changes to the API and not differences in security skill.

SSD-SES for measuring security culture. Finally, because SSD-SES is a lightweight measure, it can be administered broadly to capture an organization's "security culture". This would be useful both for researchers comparing different organizations or measuring change over time, but also for practitioners looking to diagnose whether and what actions are necessary for organizational improvement.

#	Secure Development Statement	Do Not Und.	μ	σ
F1				
B3	*I can perform a threat risk analysis (e.g., likelihood of vulnerability, impact of exploitation, etc.)*	0%	3.15	1.29
B4	*I can identify potential security threats to the system*	0%	3.61	1.10
B6	*I can identify the common attack techniques used by attackers*	0%	3.45	1.15
B11	*I can identify potential attack vectors in the environment the system interacts with (e.g., hardware, libraries, etc.)*	1.72%	2.97	1.25
B15	*I can identify common vulnerabilities of a programming language*	0%	3.44	1.15
B39	*I can design software to quarantine an attacker if a vulnerability is exploited*	1.72%	2.65	1.34
B44	*I can mimic potential threats to the system*	1.72%	3.19	1.19
B47	*I can evaluate security controls on the system's interfaces/interactions with other software systems*	1.72%	3.28	1.23
B48	*I can evaluate security controls on the system's interfaces/interactions with hardware systems*	3.45%	2.91	1.30
F2				
B8	*I can identify code that handles sensitive data (e.g., Personally Identifiable Information)*	0%	4.09	1.10
B31	*I can correctly implement authentication protocols*	0%	3.76	1.16
B32	*I can correctly implement authorization protocols*	3.45%	3.78	1.11
B50	*I can communicate security assumptions and requirements to other developers on the team to ensure vulnerabilities are not introduced due to misunderstandings*	0%	3.71	1.10
B51	*I can communicate system details with other developers to ensure a thorough security review of the code*	0%	3.85	1.16
B53	*I can discuss lessons learned from internal and external security incidents to ensure all development team members are aware of potential threats*	0%	3.88	1.11
B55	*I can effectively communicate to company leadership identified security issues and the cost/risk trade-off associated with deciding whether or not to fix the problem*	0%	3.78	1.13
B57	*I can communicate functionality needs to security experts to get recommendations for secure solutions (e.g., secure libraries, languages, design patterns, and platforms)*	3.45%	3.81	1.11
B58	*I know the appropriate point of contact/response team in my organization to contact if a vulnerability in production code is identified*	1.72%	4.04	1.25

Table 7.1: Set of secure development tasks identified after both the expert review and developer pilot transformed into *"I can"* statements. Each task was evaluated on a 5-point Likert-scale (from "I am not at all confident" to "I am absolutely confident") by 157 developers. For each statement, we give the rate of "Do not understand" responses, the average response, and standard deviations. These are the final items retained based on EFA grouped according to their associated sub-scale.

#	Secure Development Statement	Do Not Und.	μ	σ
B1	*I can determine security controls which are necessary to implement in the system*	6.90%	3.50	1.13
B2	*I can determine security requirements for the system*	1.72%	3.64	1.06
B5	*I can identify access points into the system (i.e., attack surface) which could be used by an attacker*	1.72%	3.34	1.24
B7	*I can identify critical operational requirements which must continue to function or recover quickly after an attack*	6.90%	3.44	1.15
B9	*I can identify usage patterns that should be disallowed by the system's design*	0%	3.66	1.12
B10	*I can identify potential attack vectors associated with the system under development*	6.90%	3.24	1.26
B12	*I can identify potential vulnerabilities in the operationalization of software (e.g., human errors)*	5.17%	3.77	1.10
B13	*I can identify security vulnerabilities in others? code (e.g., peer review or third party components)*	0%	3.38	1.20
B14	*I can identify common coding mistakes that create security vulnerabilities*	0%	3.80	1.07
B16	*I can understand security limitations of a programming language*	0%	3.68	1.04
B17	*I can identify potential vulnerabilities as I write code*	0%	3.73	1.01
B18	*I can use automated code analysis tools to identify vulnerabilities*	1.72%	3.31	1.27
B19	*I can use software fuzzing tools to identify vulnerabilities*	13.79%	2.92	1.33
B20	*I can review system design to identify areas where potential security risks exist*	0%	3.44	1.19
B21	*I can identify sections of code that are most likely to include security vulnerabilities*	0%	3.54	1.14
B22	*I can understand security issues and concerns associated with reused code (e.g., code samples, shared code)*	1.72%	3.79	1.07
B23	*I can apply applicable secure coding and testing standards*	3.45%	3.63	1.19
B24	*I can use provably secure programming languages*	22.41%	3.72	1.26
B25	*I can identify secure implementations of common libraries*	3.45%	3.34	1.19
B26	*I can use a secure implementation of a common library that is recommended by a security expert*	6.90%	4.01	1.14
B27	*I can apply security principles (e.g., least privilege) into the design of the system*	3.45%	3.49	1.25
B28	*I can utilize protocols that provide confidentiality of user data*	5.17%	3.83	1.18
B29	*I can utilize protocols that provide integrity of user data*	3.45%	3.81	1.13
B30	*I can utilize protocols that provide availability in the face of an attack*	10.34%	3.07	1.27
B33	*I can utilize protocols that provide non-repudiation*	39.66%	3.21	1.31
B34	*I can leverage enterprise security services to mitigate vulnerabilities (e.g., enterprise PKI)*	12.06%	3.16	1.32
B35	*I can leverage enterprise security teams for help to fix vulnerable code*	3.45%	3.93	1.23

Table 7.2: Set of secure development tasks removed based on EFA.

#	Secure Development Statement	Do Not Und.	μ	σ
B36	*I can leverage external security review (e.g., penetration testing, bug bounties) to find vulnerable code*	3.45%	3.41	1.26
B37	*I can design software to prevent potential vulnerabilities*	1.72%	3.41	1.24
B38	*I can rewrite software to remove vulnerabilities*	0%	3.66	1.18
B40	*I can write code to monitor and log system execution for later review*	1.72%	3.95	1.23
B41	*I can write error handling code to alert for possible malicious behavior*	0%	3.77	1.13
B42	*I can design software so that it fails gracefully in the face of attack*	3.45%	3.19	1.30
B43	*I can enumerate edge cases of the system's use*	13.79%	3.43	1.15
B45	*I can assess that security requirements are met (e.g., through security design and code reviews)*	0%	3.65	1.13
B46	*I can demonstrate the effectiveness of implemented security mitigations*	12.07%	3.35	1.20
B49	*I can document a system's security implications and assumptions so they are readable and actionable by others*	1.72%	3.68	1.18
B52	*I can communicate with other internal teams to understand how to securely interact with their systems*	0%	3.95	1.10
B54	*I can write understandable security and privacy error messages to draw the required user/operator attention*	0%	3.92	1.13
B56	*I can maintain awareness of hardware and software technologies' security issues and their potential implications*	0%	3.40	1.30

Table 7.3: Set of secure development tasks removed based on EFA.

	F1	ITC
α	0.907	–
IIC	0.520	–
B3	0.81	0.78
B4	0.69	0.80
B6	0.83	0.78
B11	0.79	0.81
B15	0.65	0.66
B39	0.74	0.73
B44	0.64	0.72
B47	0.64	0.77
B48	0.82	0.78

	F2	ITC
α	0.876	–
IIC	0.440	–
B8	0.64	0.66
B31	0.61	0.70
B32	0.67	0.67
B50	0.73	0.79
B51	0.68	0.74
B53	0.76	0.73
B55	0.64	0.72
B57	0.61	0.70
B58	0.80	0.67

Table 7.4: Remaining items and factor structure after initial EFA. The first two rows show reliability measures (Cronbach's α and average inter-item correlation) for each sub-scale. The remaining rows show the retained items, their loadings, and item-total correlations within the sub-scale.

#	Secure Development Statement	μ	σ	l	ITC
Vulnerability Identification and Mitigation (48.2% of variance explained, CR = 0.90)					
C3	*I can perform a threat risk analysis (e.g., likelihood of vulnerability, impact of exploitation, etc.)*	2.99	1.15	0.74	0.79
C4	*I can identify potential security threats to the system*	3.34	0.99	0.76	0.79
C6	*I can identify the common attack techniques used by attackers*	3.30	1.15	0.66	0.77
C11	*I can identify potential attack vectors in the environment the system interacts with (e.g., hardware, libraries, etc.)*	2.91	1.10	0.67	0.75
C15	*I can identify common vulnerabilities of a programming language*	3.34	1.16	0.56	0.72
C39	*I can design software to quarantine an attacker if a vulnerability is exploited*	2.41	1.19	0.77	0.72
C44	*I can mimic potential threats to the system*	2.97	1.06	0.78	0.71
C47	*I can evaluate security controls on the system's interfaces/interactions with other software systems*	3.08	1.02	0.72	0.72
C48	*I can evaluate security controls on the system's interfaces/interactions with hardware systems*	2.88	1.17	0.78	0.73
Security Communication (11.1% of variance explained, CR = 0.87)					
C50	*I can communicate security assumptions and requirements to other developers on the team to ensure vulnerabilities are not introduced due to misunderstandings*	3.46	1.03	0.73	0.86
C51	*I can communicate system details with other developers to ensure a thorough security review of the code*	3.50	1.15	0.78	0.83
C53	*I can discuss lessons learned from internal and external security incidents to ensure all development team members are aware of potential threats*	3.64	1.11	0.65	0.72
C55	*I can effectively communicate to company leadership identified security issues and the cost/risk trade-off associated with deciding whether or not to fix the problem*	3.51	1.18	0.60	0.75
C57	*I can communicate functionality needs to security experts to get recommendations for secure solutions (e.g., secure libraries, languages, design patterns, and platforms)*	3.60	1.12	0.73	0.83
C58	*I know the appropriate point of contact/response team in my organization to contact if a vulnerability in production code is identified*	3.90	1.14	0.91	0.73

Table 7.5: SSD-SES's final questions and associated sub-scales. Responses were reported on the following scale: *I am not confident at all* (1), *I am slightly confident* (2), *I am somewhat confident* (3), *I am moderately confident* (4), and *I am absolutely confident* (5).

	Vulnerability Identification and Mitigation	**Security Communication**
Convergent Validity		
CPSES	$\boldsymbol{\rho = 0.528}$, $\mathbf{p = 0.001}$	$\boldsymbol{\rho = 0.627}$, $\mathbf{p < 0.001}$
Discriminant Validity		
SA-6	$\rho = 0.249$, p = 0.191	$\rho = 0.300$, p = 0.111
$SeBIS_1$	$\rho = 0.077$, p = 0.622	$\rho = 0.202$, p = 0.298
$SeBIS_2$	$\rho = 0.370$, p = 0.053	$\rho = 0.176$, p = 0.337
$SeBIS_3$	$\rho = 0.308$, p = 0.096	$\rho = 0.227$, p = 0.224
$SeBIS_4$	$\rho = 0.145$, p = 0.412	$\rho = 0.122$, p = 0.470
GES	$\rho = 0.363$, p = 0.067	$\rho = 0.375$, p = 0.063
Other Psychometric Scales		
NFC	$\boldsymbol{\rho = 0.464}$, $\mathbf{p = 0.007}$	$\rho = 0.183$, p = 0.337
$GDMS_1$	$\rho = 0.013$, p = 0.928	$\rho = 0.129$, p = 0.450
$GDMS_2$	$\rho = 0.276$, p = 0.121	$\rho = 0.178$, p = 0.337
$GDMS_3$	$\rho = 0.148$, p = 0.403	$\rho =$-0.075, p = 0.622
$GDMS_4$	$\rho =$ -0.075, p = 0.622	$\rho = 0.139$, p = 0.403
$GDMS_5$	$\rho = 0.271$, p = 0.121	$\rho = 0.307$, p = 0.096

Table 7.6: Correlations between sub-scales, related scales, and other psychometrics.

Variable	Value	Log Estimate	CI	*p*-value
Found Vuln	**Multiple**	**1.08**	**[1, 1.17]**	**0.049**
	Once	–	–	–
	Never	**0.82**	**[0.73, 0.92]**	**< 0.001***
Expert	**Multiple**	**1.09**	**[1.01, 1.17]**	**0.024***
Coworker	One	–	–	–
	Never	**0.81**	**[0.74, 0.88]**	**< 0.001***

***Significant effect** – Base case (Log Estimate = 1, by definition)

(a) Vulnerability Identification and Mitigation

Variable	Value	Log Estimate	CI	*p*-value
Found Vuln	Multiple	1.08	[0.98, 1.18]	0.112
	Once	–	–	–
	Never	**0.86**	**[0.76, 0.98]**	**0.026***
Expert	**Multiple**	**1.06**	**[0.97, 1.15]**	**0.191***
Coworker	One	–	–	–
	Never	**0.87**	**[0.79, 0.95]**	**0.002***
Dev	**> 11 years**	**1.11**	**[1.02, 1.2]**	**0.017***
Experience	3-11 years	–	–	–
	0-2 years	0.91	[0.79, 1.03]	0.143

***Significant effect** – Base case (Log Estimate = 1, by definition)

(b) Security Communication

Table 7.7: Summary of regressions estimating relationship between each sub-scale and security experiences.

Chapter 8: Discussion

The work in this thesis provides insight into the types of vulnerabilities developers introduce, differences in experts' and non-experts' vulnerability discovery processes, the current state of security exercises, and measurement of security education outcomes.

The results from Chapter 3 showed that misunderstanding of security concepts were most common. Chapter 4 demonstrated that the biggest difference between experts and non-experts when trying to find these issues came from the variety of their different vulnerabilities to which they were exposed. These results motivated the need for building better tools to effectively leverage the experiences of security experts and improve education for non-experts to help them develop the variety of experiences necessary to find vulnerabilities.

Chapter 5 took a first step toward improving tool support, producing a revers engineering process three-phase model along with five guidelines for usable reverse engineering tool development. This chapter provides theoretical grounding for creating tools to seamlessly fit the users' process, allowing them to focus on their central task, i.e., vulnerability discovery, instead of trying to understand the interface.

In Chapters 6 and 7, I considered the education of non-experts. The results of

Chapter 6 presented a generally positive picture regarding pedagogy currently used by online security exercises, with several exercises considering many pedagogical recommendations of the learning science and education literature. However, this work did identify some avenues for educational improvement to reduce barriers for new students and engage a broader audience in vulnerability discovery. Finally, Chapter 7 produced a light-weight metric for secure development self-efficacy, useful for future comparisons of educational interventions and expertise development.

8.1 Vulnerability Discovery Experience is Essential

One consistent theme throughout this thesis was the central role of experience in vulnerability discovery. The main finding of Chapter 4 was that vulnerability discovery experience was the key differentiator between experts and non-experts. In Chapter 3, we observed that education itself did not have a significant impact on the number of vulnerabilities developers introduced, even when they were specifically taught via lecture the specific vulnerabilities to avoid. This indicates experience observing — at a minimum — vulnerabilities in a program is necessary. In Chapter 7 we found higher belief in secure development skill was correlated with vulnerability discovery experience. Finally, Chapter 5 showed that experience played an important role in every phase of reverse engineering, guiding decisions and helping reverse engineers recognize common program patterns.

Together, these results point to the need for improved and more accessible security education. In Section 6.3, we highlight several improvements to that could

be implemented in online security exercises based on the recommendations of learning science. However, while security exercises are commonly reported as the most common method for vulnerability discovery expertise development, this is only one educational method. Future work must consider other educational methods, such as certifications, on-the-job training, and direct interactions between community members. Also, particular emphasis should be placed on understanding the development of practitioners from underrepresented populations to increase the currently limited diversity of the community [55, 56, 127, 163]. This includes barriers into and within the community that limit participation and retention.

Additionally, experience is not binary. Instead, it is multi-dimensional covering several vulnerability types developed incrementally as practitioners are exposed to new issues. A practitioner may have significant experience with a particular vulnerability type, but still miss related issues because the program they are reviewing includes a problem sufficiently unique from anything they have seen before. They may also have significant experience in one area and no experience in others, e.g., knowledgable in cryptography, but not web vulnerabilities. It is unlikely that the security education community, even with the best of interventions, will be able to develop a sufficient number of experts with experience in all vulnerability types. Therefore, it is necessary that we consider this variation in our processes for finding vulnerabilities. One way this is currently done is through the growing use of bug bounties. By allowing anyone to search for bugs in a program, the program does not rely on the experience of a single practitioner, but in theory leverages the experience of the entire community. Unfortunately, this simple approach is limited as

it does not allow for coordination between practitioners, does not optimize effort to match the most relevant experience to a particular program, and is limited by the community's diversity [55, 56, 127, 163]. Future work should further investigate the effect of experience on practitioner abilities and consider developing processes and tools for vulnerabilities discovery aimed at leveraging practitioners at every level of development. For example, Nosco et al.'s assembly line approach presents a promising approach for coordination among practitioners where each performs tasks appropriate to their experience level [291].

8.2 Moving Forward with Human-Centric Vulnerability Discovery

While this thesis presents useful insights for human-centric vulnerability discovery improvements, its most significant contribution is the theoretical foundation provided for future work. For example, Chapter 5 establishes a model for reverse engineering and guidelines for usable tool design. This provides a framework for future work more precisely investigating the reverse engineering process and evaluating current and future tool designs. Similarly, Chapter 6 presents a set of pedagogy which should be considered in security exercise development and tradeoffs organizers must consider. Future work should build on these results to test how different pedagogy impact actual student behaviors.

Additionally, this thesis maintained a narrow scope on the individual's processes around vulnerability discovery. In practice, many practitioners reported working with others [413] and in most cases, they must also report their findings to

someone else to fix the vulnerability. This means that the process of vulnerability discovery has many social factors to consider. In Chapter 4, I highlight some of the issues practitioners face in reporting, but further work should consider other social dynamics such as team organization and coordination, organizational pressures on decisions and processes, and the broader community relationships (e.g., collective documentation, script development, Q&A support).

8.3 User Studies are Important for Security Professionals

Finally, my research offers a case study in the value of human factors research for professional security work. Like in vulnerability discovery, many other areas of professional security work — incident response, network defense, privacy work, etc. — rely on ad-hoc tool and process development by individuals or small groups of practitioners. This approach tends to lead to poor interface designs and incorrect assumptions about general users. Conversely, studying the actual user base allows the identification of blindspots in current approaches. As an example, in Chapter 3, we observe mismatches between vulnerabilities introduced and the types of problems many vulnerability analysis tools are designed to identify. Similarly, Chapter 6 demonstrates common challenges in current security exercises. These user studies also provide the theoretical grounding to support more usable interface design. This can be seen in the general reverse engineering process model and guidelines established in Chapter 5. Taken together, these results offer a strong argument for further work considering the human factors of security professional work.

Chapter 9: Conclusion

Software vulnerability discovery is a complex task which will require significant human intelligence for the foreseeable future. In this thesis, I identify the processes humans use to find vulnerabilities, the challenges they face, and take the first steps toward providing human-centric support.

The results of Chapter 3 show that even when incentivized to produced secure code, developers struggle with security concepts. In Chapter 4, I found that experience is the key differentiator between experts and non-experts in vulnerability discovery. Chapters 5, 6, and 7 build off these results, providing insights for both tool and educational improvements. Specifically, Chapter 5 provides a detailed model of the reverse engineering process and guidelines for usable tool development. Chapter 6 outlines the state of current online hacking exercises and provides recommendations to improve learning in this setting. Finally, Chapter 7 develops a metric for secure development self-efficacy, a needed tool for security education evaluation as well as studies of practitioner behaviors. Together, this work provides a solid foundation for dramatic improvements in human-centric vulnerability discovery.

Appendix A: Study Instruments

A.1 Chapter 4: Screening Survey

Vulnerability-discovery experience.

1. On a scale from 1-5, how would you assess your vulnerability discovery skill (1 being a beginner and 5 being an expert)?

2. Please select the range which most closely matches the number of software vulnerabilities you have discovered. (Choices: 0-3, 4-6, 7-10, 11-25, 26-50, 51-100, 101-50, > 500)

3. How many total years of experience do you have with vulnerability discovery?

4. Please select the range that most closely matches the number of hours you typically spend performing software vulnerability discover tasks per week. (Choices: < 5, 5-10, 10-20, 20-30, 30-40, > 40)

5. Please specify the range that most closely matches the number of hours you typically spend on non-vulnerability discovery, technical tasks per week (e.g. software or hardware programming, systems administration, network analysis,

etc.). (Choices: < 5, 5-10, 10-20, 20-30, 30-40, > 40)

6. What percentage of the bugs you have discovered were found in the following contexts?

   (a) Bug Bounty programs (i.e., sold specific bug to vendor)

   (b) General Software Testing

   (c) Penetration Testing

   (d) Vulnerability finding exercise (e.g., Capture-the-Flag competition, security course)

   (e) Unrelated to a specific program (i.e., for fun or curiosity)

   (f) Other

7. What percentage of the bugs you have discovered were in software of the following types?

   (a) Host (e.g., Server or PC)

   (b) Web Application

   (c) Network Infrastructure

   (d) Mobile Application

   (e) API

   (f) iOT or Firmware

   (g) Other

8. What percentage of the bugs you have discovered were of the following types?

   (a) Memory Corruption (e.g., buffer overflow)

   (b) Input Validation (e.g., XSS, SQL injection, format string misuse)

   (c) Cryptographic Error (e.g., weak key use, bad random number generator)

   (d) Configuration Error (e.g., unpatched network device)

   (e) Incorrect Calculation (e.g., integer overflow, off-by-one error)

   (f) Protection Mechanism Error (e.g., incorrect access control, improper certificate check)

   (g) Poor Security Practices (e.g., insecure data storage or transmission)

Note: In the previous three questions, we did not provide our own definitions for software type or program type and only provided examples for vulnerability types. We selected terms used by multiple popular bug bounty platforms (i.e., SynAck, HackerOne, and BugCrowd) and allowed participants to select options based on their own definitions. During the interview, we included follow-up questions to understand their definitions for the top ranked item in each category.

Technical skills.

1. On a scale from 1 to 5, how would you assess your proficiency in each of the following technical skills (1 being a beginner or having no experience and 5 being an expert)?

   (a) Networking

(b) Database Management

(c) Object-Oriented Programming (e.g., Java, C++)

(d) Functional Programming (e.g., OCaml, Haskell)

(e) Procedural Programming (e.g., C, Go)

(f) Web Development

(g) Mobile Development

(h) Distributed/Parallel Computing

(i) System Administration

(j) Reverse Engineering

(k) Cryptanalysis

(l) Software Testing

(m) Test Automation

(n) Hardware/Firmware Development

(o) Other

Demographics.

1. Please specify the gender with which you most closely identify. (Choices: Male, Female, Other, Prefer not to answer)

2. Please specify your age. (Choices: 18-29, 30-39, 40-49, 50-59, 60-69, $>$ 70, Prefer not to answer)

3. Please specify your ethnicity (Choices: White, Hispanic or Latino, Black or African American, American Indian or Alaska Native, Asian, Native Hawaiian, or Pacific Islander, Other)

4. Please specify which country/state/province you live in.

5. Please specify the highest degree or level of school you have completed (Choices: Some high school credit, no diploma or equivalent; High school graduate, diploma or the equivalent; Some college credit, no degree; Trade/technical/vocational training; Associate degree; Bachelor's degree; Master's degree; Professional degree; Doctorate degree)

6. If you are currently a student or have completed a college degree, please specify your field(s) of study (e.g., Biology, Computer Science, etc).

7. Please select the response option that best describes you current employment status. (Choices: Working for payment or profit, Unemployed, Looking after home/family, Student, Retired, Unable to work due to permanent sickness or disability, Other)

8. If you are working for payment, please specify your current job title.

9. If you are currently working for payment, please specify the business sector which best describes you job (Choices: Technology, Government or government contracting, Healthcare and social assistance, Retail, Construction, Educational services, Finance, Arts/Entertainment/Recreation, Other)

10. Please specify the range which most closely matches your total, pre-tax, household income in 2016. (Choices: <$29,999, $30,000-$49,999, $50,000-$74,999, $75,000-$99,999, $100,000-$124,999, $125,000-$149,999, $150,000-$199,999, >$200,000)

11. Please specify the range which most closely matches your total, pre-tax, household income specifically from vulnerability discovery and software testing in 2016. (Choices: <$999, $1,000-$4,999, $5,000-$14,999, $15,000-$29,999, $30,000-$49,999, $50,000-$74,999, $75,000-$99,999, $100,000-$124,999, $125,000-$145,999, $150,000-$199,999, >$200,000)

A.2 Chapter 4: Interview Protocol

A.2.1 General experience

1. What was the motivation/thought process behind performing vulnerability discovery in the different contexts you listed in the survey?

2. If most of the bugs are in a particular software or bug type:

   (a) Why do you focus on a specific area? Why this area?

   (b) Have you ever developed software in this area?

   (c) Have you worked outside of this area of expertise in the past? What was the reason for the change?

3. If the types of bugs are generally split across software or bug type:

   (a) What do you see as the importance of staying general?

(b) Are there any unique trends you've noticed that encourage you to stay general with your skills?

A.2.2 Task analysis

Program selection.

1. In general, how do you decide which software [for testers: part of the program] to investigate for vulnerabilities and which not? What factors do you consider when making this decision?

2. Which of these factors do you consider to be the most important? Why?

3. Are there any factors that you consider non-starters (i.e. reason not to try looking for bugs)?

4. Is there a specific process you use when determining where to look for vulnerabilities and which of these characteristics different software exhibit?

   (a) Why did you choose this particular process?

   (b) How did you develop/learn this process?

   (c) Are there any tools that you use that assist you in this process?

      i. What were the benefits of these tools? Weaknesses?

      ii. Have you ever used anything else for this purpose? What led you to switch?

Vulnerability-discovery process.

1. Once you've selected a software target, what steps do you take when looking for vulnerabilities?

    (a) For each step:

        i. What are your goals for this step? What information are you trying to collect?

        ii. What actions do you take to complete this step? Have you every tried anything else? What are the advantages/disadvantages of this set of actions?

        iii. Are there any tools that you use to complete this step? What were the benefits of these tools? Weaknesses?

        iv. What skills do you use to complete this task? Why do you think these skills are important? How did you develop/learn these skills?

        v. How do you know when you have successfully completed this step?

        vi. How do you decide when to take this step? Is it something you repeat multiple times? Do you always do this step?

        vii. How did you learn to take this step? Why did you find this source of information helpful?

2. Is there anything else you haven't mentioned that you've done to try to find vulnerabilities and stopped? What are the main differences between your

current process and what you did in the past? What led you to switch?

Reporting.

1. What kinds of information do you include in the report? Do you always report the same information? What factors do you consider when deciding which information to include in the report?

2. What information do you think is the most important in vulnerability reports?

3. Have you ever included/excluded anything that you didn't feel was important, but just included/excluded because you felt it was traditional/expected?

4. What bug report information is the most difficult/time consuming to get?

5. Do you ever look at anyone else's bug reports to learn from them? Why do you think these are helpful?

6. Do you use any special tool for reporting? What were the benefits of these tools? Weaknesses?

7. Do the organizations you submit to reach out to you with questions about the bugs? If so, what do they ask about?

8. Can you give me an example of a bad/good experience you've had with reporting? In your opinion, what factors are the most important for a good reporting experience?

A.2.3 Skill development

Learning.

1. Imagine you were asked for advice by an enthusiastic young person interested in learning about vulnerability discovery. How would you recommend they get started? What steps would you suggest they take?

Community participation.

1. Do you have regular communication with other hackers or software testers?
2. How do you typically communicate with others?
3. How important is each community you participate in to your/others development?
4. What community that you belong to do you find most useful? Why?
5. What information do you typically share?
6. How often do you communicate (specifically regarding technical information)?
7. How close are the relationships you have with others? How many other hackers/testers do you communicate with?

A.3 Chapter 5: Interview protocol

A.3.1 App Background

To begin our discussion, I want you to think of a program that you recently reverse engineered.

1. What was the name of the program? [If they're not comfortable telling the name, there are a few additional cues below]
   (a) What type of functionality did the app provide? [Exs: Banking, Messaging, Social Media, Productivity]
   (b) Approximately, how many lines of code or number of classes did the app have?
2. Why were you investigating this program?
3. Approximately, how long did you spend reverse engineering this app?
4. What tools did you use for your reverse engineering process? [Exs: IDAPro, debugger, fuzzer]
5. Did you reverse engineer this app with other people?
   (a) (If yes) how did you divide up the work?

A.3.2 Reverse Engineering Process

Next, we'll talk about this app in more detail. If possible, I would like you to open the program you searched the same way you did when you first started investigating it. If you would like to share your screen with me, that would be helpful for providing context, however, this is not necessary. Primarily, I want you to open everything on your computer to help you remember the exact steps you took when you searched the program.

[If they do share their screen] Also, if you are comfortable, I would like to record this screen sharing session, so that we have a later reference.

Please walk me through how you searched the program. As you go through your process, please explain every step you took, even if it was not helpful toward your eventual goal. For example, if you decided to reverse engineer a specific class, but realized it was not relevant to your search after reading the code, we would still like to know that you performed this step. [a few cueing questions are provided below to guide the conversation]

1. Where did you start?

2. What questions did you ask? How did you answer these questions?

A.3.3 Items of Interest

Decision Points. [Every time the participant had to decide between one or more

actions during their process. Ex: Where to start? What test cases to try? Which path to go down first? When to inspect a function?]

1. Record the decision that was made
2. How did you make this decisions? Explain your thought process

Hypotheses. [Every time the participant states a question they have to answer or makes a conjecture about what they think the program (or component) does. Ex: X class performs Y function. X data is transmitted off device, it's using Y encryption]

1. Record the hypothesis or question asked
2. Why did you think this could be the case?
3. How did they (in)validate this hypothesis?

Beacons. [Every time the participant states recognizing the functionality of some code without actually stepping through it. That is, they are able to notice some pattern in the code and make some deductions about functionality based on this]

1. Record the beacon that was noticed
2. Why did this stand out to you? How were you able to recognize it?
3. How did you know that it was X instead of something else?

Simulation. [Every time the participant discusses looking at the code to determine how it works]

1. Record how they investigate the code.

    (a) (If Automation) Do you use a custom tool or something open source/purchased?

        i. (If not custom) What tool do you use?

            A. Does this tool provide the results you would want or does it fall short in some way? [Ex: I actually want output X, but I get Y, so I need to do these steps to get to X]

    (b) Is this generally the approach you use?

        i. (If no) Why here and not in other cases?

        ii. (If yes) What advantage do you think this approach has over other manual/automated investigation?

2. Please describe what's going on in your head or the automation?

    (a) What are the inputs and outputs?

    (b) When do you know when to stop?

Resources. [Every time the participant discusses referencing some documentation or information source external to the code]

1. Record what resource they used

2. Do you regularly consult this resource for information?

3. What do you think the benefit of this resource is over other sources of information? [Exs: Language documentation, Stack Overflow, internal documentation]

A.4 Chapter 6: Interview Protocol

Over the past few months, we have been looking at several online capture-the-flag competitions, wargames, and other security exercises with an educational focus. For each exercise, we completed several challenges reviewing the content and the way in which this information was presented to students.

Our goal was to see how each exercise is organized, focusing on features commonly recommended in the learning science and education literature. This includes features related to connecting to the learner's prior knowledge, the organization of knowledge, providing active practice and feedback, encouraging metacognitive learning, and establishing a supportive environment.

In this interview, I want to talk about each of these categories, the specific dimensions of each, and how we viewed these as being applied in your exercise. We have a few goals with this interview. The first is to make sure we have a clear picture of your exercise. You know it much better than we ever will, so we want to tap into your knowledge to make sure we're not missing something. Second, we want to understand more about why your exercise is organized the way it is. The dimensions we're looking at are recommendations, not requirements. As you'll see, they are many good things and exercise might have. We would expect an exercise

to have some, but not all. So, in this interview, we want to get at why you decided to include some, but not others.

A.4.1 General Organizational Questions

Before we begin going through our review, I have a few general questions about your organization to help us better understand the context in which the exercise was developed.

1. What lead you to create your exercise? That is, what was your main motivation in providing this educational platform?

2. Who is your target audience(s)?

A.4.2 Analysis Review

Moving on to our review of your exercise, now I'll go through each core pedagogical principle and its dimensions. For each, I will give you our definition and our decision and then get your response.

For each core principle and related dimension, we provided the interviewee with the textual definitions given throughout Section 6.2

For each dimension coded No or Partial, we explained the reasoning behind our decision and asked the following question progression:

1. Do you agree?

   (a) *If no:* Why not?

(b) *If yes:* Do you agree that this pedagogical dimension would be helpful to students if implemented?

i. *If no:* Why not?

ii. *If yes:* Why did you choose not to implement it?

A. *If they said it was too difficult:* What are the challenges?

A.5 Chapter 7: Expert Review Survey

A.5.1 Instructions

Thank you for participating as an expert reviewer. Throughout this survey, you will be shown a list of secure software development tasks we have collected from several secure development guidelines and frameworks including NIST's NICE framework, Microsoft's SDL, BSIMMs, and OpenSAMM.

For each secure development task, you will be asked to indicate whether you believe the task is or is not actually a component of the secure development process. Specifically, we care about tasks that have an effect on the end product software. These tasks could be carried out by any member of the organization (e.g., programmer, product manager, etc) at any phase of the development process (e.g., requirements, design, implementation, etc), but must have an effect on the final code that is produced.

Through your review, we hope to remove any items from our list that may have been incorrectly included. Additionally, we will ask if the task wording is clear. If you find the text to be unclear or confusing, please briefly explain your confusion

so that we can improve the statement's clarity.

If there are any secure development tasks that you do not believe were covered, please provide these at the conclusion of the survey so that they can be added to our list.

Note, for this survey we ask that you consider each task listed independently. For example, multiple tasks may reflect similar concepts. Please consider the adequateness of each of these tasks separately. Additionally, the given order the tasks is arbitrary. They are not meant to represent any sequential order of operations, but instead are tasks that should be completed at some point in the secure development process (in some cases multiple times).

A.5.2 Tasks

[Task description from Tables B.10-B.12]

1. Select the option below that best reflects the above task.

   (a) Definitely a secure development task

   (b) Probably a secure development task

   (c) Probably not a secure development task

   (d) Definitely not a secure development task

   (e) Unsure

2. Did you find the text for the above task unclear or confusing?

   (a) Yes. Please briefly explain what is confusing or unclear.

(b) No

A.5.3 Final Thoughts

1. Are there any secure software development tasks that we did not include? Please list all such tasks below, each on its own line.

2. Do you have any other questions, comments, concerns, or suggestions for our research team that were not addressed throughout the survey?

A.5.4 Demographics

1. Please specify your current (or most recent) job title.

2. How many total years of experience do you have with secure software development?

3. Please specify the highest degree or level of school you have completed

   (a) Some high school credit, no diploma

   (b) High school graduate, diploma or equivalent (e.g., GED)

   (c) Some college credit, no degree

   (d) Associate's degree

   (e) Bachelor's degree

   (f) Master's degree

   (g) Doctoral degree

4. If you are currently a student or have completed a college degree, please specify your field(s) of study (e.g. Biology, Computer Science, etc).

A.6 Chapter 7: Main Survey

A.6.1 Instructions

Throughout this survey, you will be shown a list of software development tasks. For each software development task, you will be asked to indicate your confidence in your ability to complete the task.

Note that throughout we refer to your ability to perform each task with respect to the system under development. Here, when we refer to the "system", we specifically mean the features you are involved in the design and implementation of. This could be a single program or collection of programs. For example, If you work on a specific set of features or an application that integrates with a large environment of programs, the "system" would only be those features or applications you work on.

After indicating your confidence in completing each task, you will be asked to answer a few additional questions regarding your decision-making.

Finally, you will be asked to answer a few demographic questions.

(Note: Only participants in the fourth step of the scale development process were asked to answer additional scale questions)

A.6.2 Tasks

Please rate your level of confidence in completing the software development tasks given below.

["I can" statements from Tables 7.1 and **??**]

1. I am not confident at all
2. I am slightly confident
3. I am somewhat confident
4. I am moderately confident
5. I am absolutely confident
6. I do not understand the question

A.6.3 Demographics

1. How many total years of experience do you have with software development?
2. Have you ever identified a security error in yours or someone else's code?
   (a) Never
   (b) Once
   (c) Multiple times
3. Has anyone ever identified a security error in code you have written?

(a) Never

(b) Once

(c) Multiple times

4. Have you ever worked with someone you would consider a secure development expert?

(a) Never

(b) I have worked with one security expert

(c) I have worked with multiple security experts

5. How often do you communicate with security experts?

(a) Yearly

(b) Quarterly

(c) Monthly

(d) Weekly

(e) Daily

6. Have you ever discussed potential security problems with a secure development expert?

(a) Never

(b) Once

(c) A few times

(d) On a regular basis

7. Please select the range that most closely matches the amount of time you typically spend performing software development tasks per week.

   (a) ¡ 5 hours

   (b) 5-10 hours

   (c) 10-20 hours

   (d) 20-30 hours

   (e) 30-40 hours

   (f) ¿ 40 hours

8. Please indicate the response below that most closely matches your software development skill level:

   (a) Expert (recognized authority)

   (b) Advanced (applied theory)

   (c) Intermediate (practical application)

   (d) Novice (limited experience)

   (e) Fundamental awareness (basic knowledge)

   (f) Not applicable

9. Have you ever participated in any training related to software security (e.g., coursework, employer training, online exercises, etc)?

(a) Yes

(b) No

(c) Not sure

10. Please select from the list of training types below, all the types of prior software security training you have participated in:

(a) Academic coursework

(b) Certification

(c) Security conference

(d) Online exercises

(e) Employer training

11. Please specify the gender with which you most closely identify.

(a) Male

(b) Female

(c) Other

(d) Prefer not to answer

12. Please specify your age

(a) 18-29

(b) 30-39

(c) 40-49

(d) 50-59

(e) 60-69

(f) Over 70

13. Please specify your ethnicity. Select all that apply.

(a) White

(b) Hispanic or Latino

(c) Black or African American

(d) American Indian or Alaska Native

(e) Asian, Native Hawaiian, or Pacific Islander

(f) Native Hawaiian or Pacific Islander

(g) Other

(h) Prefer not to answer

14. Please specify the highest degree or level of school you have completed

(a) Some high school credit, no diploma

(b) High school graduate, diploma or equivalent (e.g., GED)

(c) Some college credit, no degree

(d) Associate's degree

(e) Bachelor's degree

(f) Master's degree

(g) Doctoral degree

15. If you are currently a student or have completed a college degree, please specify your field(s) of study (e.g. Biology, Computer Science, etc).

16. Please select the response option that best describes your current employment status.

    (a) Working for payment or profit

    (b) Unemployed

    (c) Looking after home/family

    (d) Student

    (e) Retired

    (f) Unable to work due to permanent sickness or disability

    (g) Other

    (h) Prefer not to answer

17. Please specify the range which most closely matches your total, pre-tax, personal income specifically from software development last year.

    (a) < $999

    (b) $1,000-$4,999

    (c) $5,000-$14,999

    (d) $15,000-$29,999

(e) $30,000-$49,999

(f) $50,000-$74,999

(g) $75,000-$99,999

(h) $100,000-$124,999

(i) $125,000-$149,999

(j) $150,000-$199,999

(k) > $200,000

(l) Prefer not to asnwer

Appendix B: Additional Data

B.1 Chapter 3: Additional Contest Details

To provide additional context for our results, this appendix includes a more thorough breakdown of the sampled population along with the number of breaks and vulnerabilities for each competition. Table B.1 presents statistics for sampled teams, participant demographics, and counts of break submissions and unique vulnerabilities introduced divided by competition. Figure B.1 shows the variation in team sizes across competitions.

B.2 Chapter 3: Additional Coding

We coded several variables in addition to those found to have significant effect on vulnerability types introduced. This appendix describes the full set of variables coded. Table B.2 provides a summary of all variables.

Hard to read code is a potential reason for vulnerability introduction. If team members cannot comprehend the code, then resulting misunderstandings could cause more vulnerabilities. To determine whether this occurred, we coded each project according to several readability measures. These included whether the

Contest	Fall 14 (SL)	Spring 15 (SL)	Fall 15 (SC)	Fall 16 (MD)	Total
# Teams	10	42	27	15	94
# Contestants	26	100	86	35	247
% Male	46 %	92 %	87 %	80 %	84 %
% Female	12 %	4 %	8 %	3 %	6 %
Age	22.9/18/30	35.3/20/58	32.9/17/56	24.5/18/40	30.1/17/58
% with CS degrees	85 %	39 %	35 %	57 %	45 %
Yrs. programming	2.9/1/4	9.7/0/30	9.6/2/37	9.6/3/21	8.9/0/37
Team size	2.6/1/6	2.4/1/5	3.2/1/5	2.3/1/8	2.7/1/8
# PLs known per team	6.4/3/14	6.9/1/22	8.0/2/17	7.9/1/17	7.4/1/22
% MOOC	0%	100 %	91 %	53 %	76 %
# Breaks	30	334	242	260	866
# Vulns.	12	41	64	65	182

Table B.1: Participants demographics from sampled teams with the number of breaks submitted and vulnerabilities introduced per competition. Some participants declined to specify gender. Slashed values represent mean/min/max

project was broken into several single-function sub-components (Modularity), whether the team used variable and function names representative of their semantic roles (Variable Naming), whether whitespace was used support visualization of control-flow and variable scope (Whitespace), and whether comments were included to summarize relevant details (Comments).

Additionally, we identified whether projects followed secure development best practices [58, pg. 32-36], specifically *Economy of Mechanism* and *Minimal Trusted Code*.

When coding *Economy of Mechanism*, if the reviewer judged the project only included necessary steps to provide the intended security properties, then the project's security was economical. For example, one project submitted to the secure log problem added a constant string to the end of each access log event before

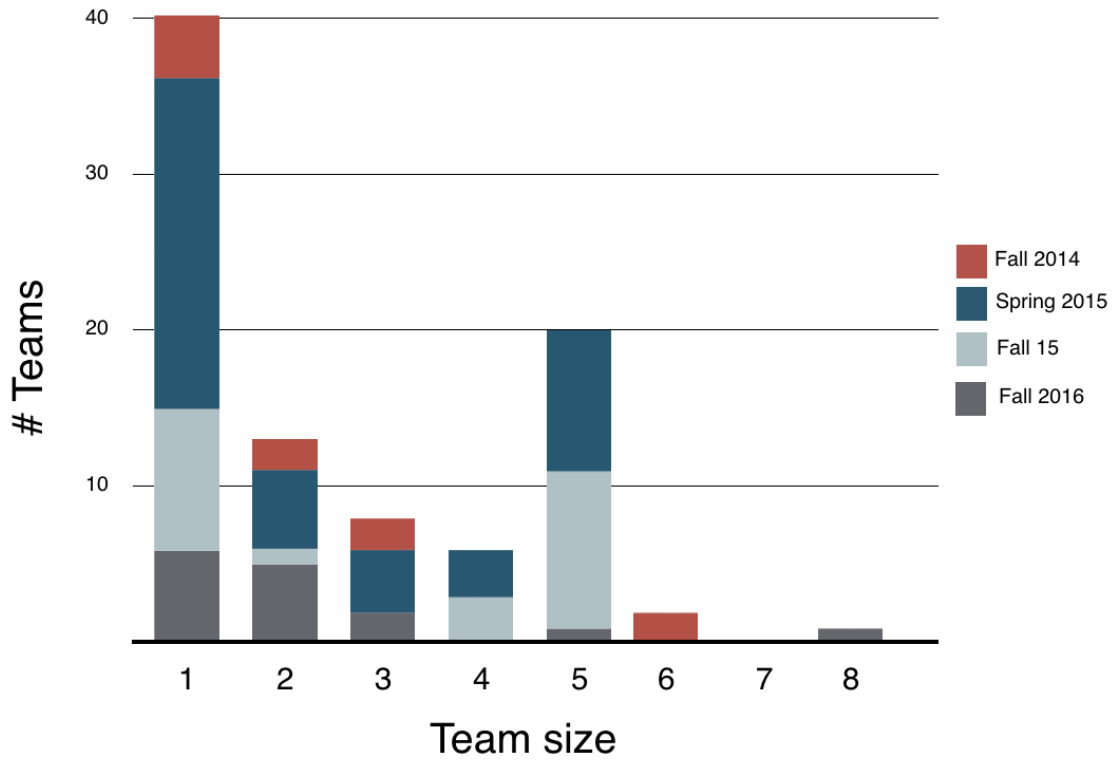

Figure B.1: Histogram of team size by competition.

encrypting. In addition to using a message authentication code to ensure integrity, they checked that this hardcoded string was unchanged as part of their integrity check. Because removing this unnecessary step would not sacrifice security, we coded this project as not economical.

Minimal Trusted Code was measured by checking whether the security-relevant functionality was implemented in multiple locations. Projects passed if they created a single function for each security requirement (e.g., encryption, access control checks, etc.) and called it throughout. The alternative—copying and pasting code wherever security functionality was needed—is likely to lead to mistakes if each code segment is not updated whenever changes are necessary.

Variable	Levels	Description	Alpha
Modular	T / F	Whether the project is segmented into a set of functions and classes each performing small subcomponents of the project	1
Variable Naming	T / F	Whether the author used variable names indicating the purpose of the variable	1
Whitespace	T / F	Whether the author used whitespace (i.e., indentation and new lines) to allow the reader to easily infer control-flow and variable scope	1
Comments	T / F	Whether the author included comments to explain blocks of the project	0.89
Economy of Mechanism	T / F	How complicated are the implementations of security relevant functions	0.84
Minimal Trusted Code	T / F	Whether security relevant functions are implemented once or multiple times	0.84

Table B.2: Summary of the project codebook.

B.3 Chapter 3: Regression Analysis

For each vulnerability type subclass, we performed a poisson regression [63, 67-106] to understand whether the team's characteristics or their programming decisions influenced the vulnerabilities introduced. In this appendix, we provide an extended analysis discussion, focusing on the full set of covariates in each initial model, our model selection process, and the results omitted from the main paper due to their lack of significant results or poor model fit.

B.3.1 Initial Covariates

As a baseline, all initial regression models included factors for the language used (*Type Safety* and *Popularity*), team characteristics (development experience and security education), and the associated problem. These base covariates were used to understand the effect of a team's intrinsic characteristics, their development environment, and the problem specification. The *Type Safety* variable identified whether each project was statically typed (e.g., Java or Go, but not C or C++), dynamically typed (e.g., Python, Ruby), or C/C++ (*Type Safety*).

For *Misunderstanding* regressions, the *Bad Choice* regression only included the baseline covariates and the *Conceptual Error* regression added the library type (*Library Type*). The project's *Library Type* was one of three categories based on the libraries used (*Library Type*): no library used (*None*), a standard language library (e.g., PyCrypto for Python) (*Language*), or a non-standard library (*3rd Party*).

The *No Implementation* regressions only included the baseline covariates. Additionally, since the *Some Intuitive* vulnerabilities only occurred in the MD problem, we did not include problem as a covariate in the *Some Intuitive* regression.

In addition to the baseline covariates, the *Mistake* regression added the *Minimal Trusted Code* and *Economy of Mechanism* variables, whether the team used test cases during the build phase, and the project's number of lines of code. These additional covariates were chosen as we expect smaller, simpler, and more rigorously tested code to include less mistakes.

B.3.2 Model Selection

We calculated the Bayseian Information Criterion (BIC)—a standard metric for model fit [329]—for all possible combinations of the initial factors. To determine the optimal model and avoid overfitting, we selected the minimum BIC model.

As our study is semi-controlled, there are a large number of covariates which must be accounted for in each regression. Therefore, our regressions were only able to identify large effects [85]. Note, for this reason, we also did not include any interaction variables. Including interaction variables would have reduced the power of each model significantly and precluded finding even very large effects. Further, due to the sparse nature of our data (e.g., many languages and libraries were used, in many cases only by one team), some covariates could only be included in an aggregated form, limiting the analysis specificity. Future work should consider these interactions and more detailed questions.

B.3.3 Results

Tables B.3–B.7 provide the results of each regression not included in the main text.

Variable	Value	Log Estimate	CI	*p*-value
Popularity	C (91.5)	1.03	[0.98, 1.09]	0.23

*****Significant effect – Base case (Estimate=1, by definition)

Table B.3: Summary of regression over *Bad Choice* vulnerabilities. Pseudo R^2 measures for this model were 0.02 (McFadden) and 0.03 (Nagelkerke).

Variable	Value	Log Estimate	CI	*p*-value
MOOC	False	–	–	–
	True	1.76	[0.70, 4.34]	0.23

*Significant effect – Base case (Estimate=1, by definition)

Table B.4: Summary of regression over *Conceptual Error* vulnerabilities. Pseudo R^2 measures for this model were 0.01 (McFadden) and 0.02 (Nagelkerke).

Variable	Value	Log Estimate	CI	*p*-value
Yrs. Experience	8.9	1.12	[0.82, 1.55]	0.47

*Significant effect – Base case (Estimate=1, by definition)

Table B.5: Summary of regression over *All Intuitive* vulnerabilities. Pseudo R^2 measures for this model were 0.06 (McFadden) and 0.06 (Nagelkerke).

B.4 Chapter 5: Codebook

In this appendix, we list the final codebook used to analyze the content of each interview. Our codebook was divided into six parts, reflecting our items of interest discussed in Section 5.2.1. For each code, we give a short description where necessary. Some codes were further divided into sub-codes to provide additional specificity to our analysis. We indicate this hierarchical relationship by presenting sub-codes in an indented bulleted list under their parent's code.

B.4.1 Hypotheses

For each hypothesis, we coded both the justification or observation that led to the formulation of a particular hypothesis (*reason*) and type of hypothesis the formed (*type*).

Variable	Value	Log Estimate	CI	*p*-value
Problem	SC	–	–	–
	SL	1.02	[0.98, 1.07]	0.373

*Significant effect – Base case (Estimate=1, by definition)

Table B.6: Summary of regression over *Some Intuitive* vulnerabilities. Pseudo R^2 measures for this model were 0.02 (McFadden) and 0.07 (Nagelkerke).

Variable	Value	Log Estimate	CI	*p*-value
Problem	SC	–	–	–
	MD	0.58	[0.25, 1.35]	0.21
	SL	**0.31**	**[0.15, 0.60]**	**¡ 0.001***

*Significant effect – Base case (Estimate=1, by definition)

Table B.7: Summary of regression over *Unintuitive* vulnerabilities. Pseudo R^2 measures for this model were 0.07 (McFadden) and 0.16 (Nagelkerke).

B.4.1.1 Reason

- Structure - The RE made their inference based on the structure of the data reviewed. For example, ten digits separated by three dashes is probably a phone number.

- Observed Behavior - The RE made a determination about program functionality after a full evaluation of the code or execution. That is, they did not rely on outside information to determine the code functionality.

- Prior Experience - The RE made an inference about program behavior without fully evaluating the code by drawing on similar past experiences.

B.4.1.2 Type

- Vulnerability - The RE hypothesized that a particular code segment was vulnerability to exploitation.
- Function - The RE hypothesized what the behavior of a particular code segment was.
- Data Type or Purpose - The RE hypothesized what the type or purpose of a variable or register was.

B.4.2 Question

- What is the observable behavior of the program? - The RE asked what information could be observed when running the program without using any introspection tools (e.g., debugger, packet capture, etc.).
- What does the program do for input X? - The RE checks how the program responds when provided with a specific input of interest.
- How is variable/register/constant X used? - The RE seeks to determine how a specific value of interest is used by the program.
- What security controls are being used? - The RE asks what mitigations are in place around the program to prevent exploitation (e.g., ASLR, DEP, etc.)
- What is the output of function/code X? - The RE seeks to determine the possible output of a function or block of code of interest.

- What is the possible value of variable/register X at point Y? - The RE seeks to determine all possible values of a specific variable or register at a point of interest in the program.

- What is the concrete value of variable/register X at point Y? - The RE seeks to determine the value of a variable or register of interest at a specific point in the program given a concrete trace of the program's execution.

- Where are the strings/names related to X? - The RE seeks to find semantically similar strings and variable or function names in the program related to a concept of interest (e.g., encryption).

- What input leads to point X being reached? - The RE seeks to determine the specific input that will cause a segment of code of interested to be executed.

- Can code at point X be reached? - The RE seeks to determine whether it is possible for a segment of code of interest to be executed.

- How has the program changed over time? - The RE asks what changes to the program's code have been made between the current program version and previous versions.

- What is the control flow path for input X? - The RE seeks to determine what control flow path through the program is followed when an input of interest is provided.

- What is the type of variable/register X? - The RE seeks to determine the type

of data (e.g., string, integer, pointer, etc.) stored in a variable or register of interest.

- What is the possible input to function X? - The RE seeks to determine all possible inputs to a function of interest.
- What is the output of function X used for? - The RE seeks to determine how a functions output is used. For example, is the data transmitted to another device over the Internet and what is the reason for this transmission?
- What call/uses X (function, string, offset, register)? - The RE asks what other functions call or use a particular item of interest.
- What does function/code X do? - The RE seeks to determine what the overall behavior of a segment of code or function is.
- What does function X call? - The RE asks what other functions a function of interest calls.

B.4.3 Beacon

- String - In the usual sense, meaning the primitive datatype indicating a series of null terminated characters.
- API calls - In the usual sense, meaning function calls from external libraries.
- Low Level Operations - Individual assembly code operations and their parameters (e.g., mov or add).

- Constants - In the usual sense, meaning primitives in the code that hold a value that does not change.
- Variable Name - The name of a variable, which can provide hints about the behavior of the program or the purpose of the variable.
- Operation Sequence -A specific order or sequence of some operations (e.g., API calls, assembly instructions, constants, etc.) that the RE that indicate a particular behavior to the RE on first glance.
- Comments - Any comments left in the code by the developers. These may be available if the RE has access to source code.
- Program Metadata - Meta information about the program itself such as the number of lines of code.
- Function Prototype - In the usual sense, meaning the type that the function returns, its name, and its parameters.
- UI element - Any element of in the UI of the program.
- Control Flow - A specific path through the program that is dictated by some decided sequence of conditional branches.
- Program Flow - The position of a particular function or code segment in relation to the broader order of behaviors. That is, the RE can make inferences about a function or code segment's behavior knowing that it comes before, after, or in concert with other behaviors.

B.4.4 Simulation Method

The simulation methods we observed were divided into three groups: dynamic analysis, static analysis, and metadata review. In addition to coding these methods, we also coded interactions of their use.

B.4.4.1 Dynamic Analysis

- Execute with specific input - Executing the program with input chosen with some specific purpose or idea behind it.
- Execute in debugger to a certain point - Setting a breakpoint in the debugger and executing.
- Manipulate environment - Running the program itself and altering outside parameters (e.g., network state, files on disc, etc.) while observing how these affect the program.
- Monitor dynamic behavior - Run the program with additional tooling (e.g., packet capture, file monitoring) to see how it interacts with its environment.
- Edit, recompile, and run - When an RE changes the source code, then runs it to see what happens.
- Fuzzing - Providing a series of varied inputs to the program and observing their effect on program behavior. These inputs can be selected manually or using automation.

- Compare to known implementation - Running a known implementation of an algorithm used by the program (such as an encryption algorithm) and comparing the known implementation's results to the program's results.

B.4.4.2 Static Analysis

- Read code line-by-line - The RE simply processes the state of the program in their head by running through the code line-by-line.
- Scan beacons - Scanning through the code quickly and without much detail in the interest of identifying important beacons.
- List function imports/strings - Listing out the functions imported or the strings used in the program.
- Search for specific string - The RE checks to see if a specific string is used. This is performed either manually (i.e., scrolling through and scanning the code) or with the help of a search tool.
- List file metadata - Listing out the metadata of a specific file such as its size or type.
- Review differences from prior releases - The RE looks at how the program has changed from version to version.
- Reconstruct a data structure - The RE writes out a variable's data structure in psuedo-code by making inferences from the binary.

- Control flow analysis - Analyzing some path through the code on specific control flow inputs.
- Data flow analysis - Analyzing some path through the code by data as it is passed between variables and through memory.
- Symbolic execution - The RE determines the set of symbols and expressions representing possible values of data at a specific point in the program.
- Function call cross-referencing - The RE determines where a particular function is called in the code.
- Compare to known implementation - The RE compares the code they believe is performing a particular algorithm or function to code from an outside source that is known to perform that algorithm or function.
- Reimplementation - The RE writes a program to perform the behaviors they believe the program under investigation is performing. They then compare their implementation to the program under inspection to determine whether they are the same or identify differences.

B.4.4.3 Metadata Review

- check security mitigations - Checking the security restrictions or mitigations.

B.4.4.4 Method Interactions

- View static and dynamic representation together - When the RE views both static and dynamic code representations on their screen at the same time. For example, if they have a disassembler open reading code line-by-line side-by-side with a debugger.

- Static to Dynamic - When the RE first uses a static method, then uses information from this to inform the use of a dynamic method.

- Dynamic to Static - When the RE first uses a dynamic method, then uses information from this to inform the use of a static method.

- Combined - When the RE uses dynamic and static methods in concert, constantly passing information back and forth between static and dynamic methods.

B.4.5 Decision

For each decision point, we coded both the reasoning behind the RE's decision (*reason*) and type of decision the RE made (*type*).

B.4.5.1 Reason

- State of investigation - The RE makes a decision based on where they are within the investigation process. For example, the RE may choose to list

APIs called because that is always their first step when reverse engineering a new program.

- Strategy - The decision is dictated by an overarching RE strategy. For example, the RE may choose to look at functions in descending order according to their size.

- Function prototype - The RE made a decision based on a function's prototype (input and output types, number of arguments, etc). For example, if a function is passed a large number of function pointers as arguments, then it might be starting many threads and would be interesting to investigate.

- Specific sub-goal - The decision was made because it was necessary to complete some other task.

- Control flow path - The decision was dictated by the control flow path that was currently being followed.

- Data flow path - The decision was dictated by the data flow path that was currently being followed.

- Proximity to interesting information - The decision about some element was made because it is physically nearby some interesting information in the code.

- Prior experience - The RE made their decision based on prior reverse engineering experience.

- Program metadata - The RE made their decision based only on program metadata.
- Observed behavior - The decision was made based on an understanding of what the program was actually doing.

B.4.5.2 Type

- Function/code to analyze - Which code segments or functions to attempt to analyze.
- Analysis inputs - Which inputs to use when performing a simulation method.
- Order of functions/code to analyze - The order in which the reverse engineer analyzes portions of code or functions.
- Simulation method to use - Which simulation method to employ at that particular moment.

B.5 Chapter 6: Codebook

In this appendix, we provide the codebook used to categorize organizer decisions for not fully implementing reviewed pedagogical principles.

- **Does not fit our goals**: The organizer considered implementing the dimension, but chose not to because it did not fit within the structure and intended goals of the exercise.

- **Challenging**: The organizers considered implementing the dimension, but decided that implementation would be too difficult.
- **Not considered**: The organizers did not consider the dimension when designing the exercise.

B.6 Chapter 6: Additional Dimensions

B.7 Chapter 7: Additional Data

In this appendix, we provide additional data that was not presented in the main paper for space considerations. Specifically, this includes the distribution of software development experience across participants assigned each additional scale (Table B.9) and the set of initial items (with sources) generated through our review of relevant frameworks (Tables B.10-B.12).

	Content					**Active Practice**	**Metacognition**
	Reverse Engineering	Cryptography	Web Vulnerabilities	Binary Exploitation	Forensics and Networking	Exploit Code	How to use
Synchronous							
gCTF† [149]	●	●	●	●	●	●	●
Infosec Institute [205]	○	●	●	○	●	◐	●
picoCTF† [296]	●	●	●	●	●	●	●
CSAW365 [235]	●	●	●	●	●	●	●
HackEDU [161]	●	●	●	●	○	●	●
Pwnadventure† [4]	●	●	◐	●	○	●	●
PACTF [308]	●	●	○	○	●	●	●
Angstrom† [14]	●	●	●	●	●	●	●
HXP CTF [201]	●	●	●	●	●	●	●
BIBIFI† [69]	●	●	○	●	●	●	●
Pwn College† [365]	●	○	○	●	●	●	●
GirlsGo CyberStart† [206]	●	●	●	○	●	●	●

	Content					**Active Practice**	**Metacognition**
	Reverse Engineering	Cryptography	Web Vulnerabilities	Binary Exploitation	Forensics and Networking	Exploit Code	How to use
Asynchronous							
HackTheBox† [46]	●	●	●	●	●	●	●
HackthisSite [167]	●	●	●	○	●	●	●
OverTheWire [304]	●	●	●	●	●	●	●
Root-me.org† [266]	●	●	●	●	●	●	●
Vulnhub† [142]	●	●	●	●	●	●	●
Hacker101 [162]	●	●	●	●	●	●	●
Hellbound Hackers [165]	●	●	●	●	●	●	●
Smash the Stack† [392]	●	●	●	●	○	●	●
Microcorruption [156]	●	○	○	●	○	●	●
Pwnable [375]	●	●	○	●	○	●	●
Cyber Talents [98]	●	●	●	●	●	●	●
XSS-Game† [150]	●	○	●	○	○	●	●
Backdoor [357]	●	●	●	●	●	●	●
Crackmes.one† [350]	●	●	○	●	○	●	●
CTFlearn [95]	●	●	●	●	●	●	●
HackerTest [166]	○	○	●	○	○	●	●
Mr. Code† [217]	●	●	●	○	●	●	●
IO Wargame [286]	●	○	○	●	○	●	●

† An organizer from this exercise was interviewed or responded via email to our review.

Table B.8: Dimensions of our pedagogical analysis not covered in Table 7.5. Each column indicates whether an exercise implemented the pedagogical dimension fully (●), partially (◐), or not at all (○).

Scale	1-2 years	3-11 years	> 11 years
CPSES	1	34	14
SA-6	6	30	12
SeBIS	4	34	12
GES	5	33	6
NFC	6	30	13
GDMS	8	29	15
Total	15	95	36

Table B.9: The distribution of participants' software development experience who were shown each additional scale.

#	Secure Development Task	N[1]	B[2]	O[3]	S[4]	F[5]
A1	*Determine security controls required for the program*	✓		✓	✓	
A2	*Determine security objectives required for the program*	✓		✓	✓	
A3	*Perform risk analysis (e.g., probability of successful attack occurrence)*	✓	✓	✓	✓	
A4	*Identify potential threats to the program*	✓	✓	✓	✓	✓
A5	*Identify access points into the system (i.e., attack surface) which could be used by an attacker*	✓	✓	✓	✓	✓
A6	*Identify the common attack techniques used by potential threats to the application*	✓	✓	✓	✓	✓
A7	*Identify critical operational requirements which must continue to function in the face of an attack or recover quickly afterwards*	✓	✓	✓		✓
A8	*Identify functions that handle sensitive data (e.g., Personally Identifiable Information)*	✓	✓	✓	✓	✓
A9	*Identify usage patterns (e.g., misuse, abuse of functionality) that should be disallowed by the software's design*			✓		
A10	*Analyze tradeoffs between cost and protection provided by security controls*	✓				
A11	*Perform code signing (e.g., integrity checks on software) for use in cryptographically verifying the authenticity of a module or release*			✓		
A12	*Identify potential attack vectors associated with the program under development*	✓		✓	✓	✓
A13	*Identify potential attack vectors in environment the program interacts with (e.g., hardware, libraries, etc)*	✓		✓	✓	✓
A14	*Identify potential vulnerabilities in the operationalization of software (e.g., human errors, cultural or political issues)*	✓		✓		✓
A15	*Identify security vulnerabilities in others' code*	✓	✓			✓
A16	*Identify common coding mistakes that create security vulnerabilities*	✓	✓	✓		
A17	*Identify potential vulnerabilities as you write code*	✓	✓			
A18	*Use automated code analysis tools to identify vulnerabilities*			✓		✓
A19	*Use software fuzzing tools to identify vulnerabilities*			✓		✓
A20	*Identify areas in program design where security risks might be possible*			✓		✓
A21	*Identify sections of code that might include security vulnerabilities*			✓		✓
A22	*Understand security issues and concerns associated with reused code (e.g., third-party libraries, shared code)*			✓		✓

[1] NICE, [2] BSIMM, [3] OSAMM, [4] SDL, [5] SAFECode

Table B.10: Initial secure development tasks identified through framework review. The SAFECode framework was not included in the initial review, but added after multiple expert recommendations.

#	Secure Development Task	N[1]	B[2]	O[3]	S[4]	F[5]
A23	*Apply applicable secure coding and testing standards*	✓	✓		✓	✓
A24	*Apply security principles (e.g., least privilege, layered defense) into design of the program*			✓	✓	✓
A25	*Utilize protocols that provide confidentiality*	✓		✓		✓
A26	*Utilize protocols that provide integrity*	✓		✓		✓
A27	*Utilize protocols that provide availability*	✓		✓		✓
A28	*Correctly implement authentication protocols*	✓		✓		✓
A29	*Utilize protocols that provide non-repudiation*	✓		✓		
A30	*Leverage enterprise security services to mitigate vulnerabilities (e.g., enterprise PKI)*	✓		✓		
A31	*Leverage enterprise security experts for help to fix vulnerable code*	✓				
A32	*Design software to prevent potential vulnerabilities*	✓		✓	✓	✓
A33	*Rewrite software to remove software vulnerabilities*	✓	✓			✓
A34	*Design software to quarantine attacker if a vulnerability is exploited*	✓		✓		
A35	*Write code to monitor and log program execution for later review*	✓	✓			✓
A36	*Write error handling code to alert for possible malicious behavior*	✓	✓			✓
A37	*Design software so that it fails gracefully in the face of attack*	✓	✓			
A38	*Enumerate boundary conditions and mimic potential threats*	✓	✓	✓	✓	
A39	*Assess that security requirements are met (e.g., through security design and code reviews)*	✓	✓	✓		
A40	*Demonstrate the effectiveness of implemented security mitigations*	✓	✓	✓		✓
A41	*Evaluate security controls on interfaces to other software systems the program interfaces with*	✓		✓		
A42	*Evaluate security controls on interfaces to other hardware systems the program interfaces with*	✓		✓		

[1] NICE, [2] BSIMM, [3] OSAMM, [4] SDL, [5] SAFECode

Table B.11: Additional initial secure development tasks identified through framework review. The SAFECode framework was not included in the initial review, but added after multiple expert recommendations.

#	Secure Development Task	N[1]	B[2]	O[3]	S[4]	F[5]
A43	*Document security implications and assumptions of software so that they are readable and actionable by future developers*	✓		✓		
A44	*Communicate security assumptions and requirements to other developers to avoid vulnerabilities caused by misunderstandings*	✓	✓	✓		
A45	*Communicate program details with other developers to ensure a thorough security review of the code*			✓		
A46	*Communicate with other internal teams the program integrates with to understand how to securely interact with their systems*	✓		✓		
A47	*Discuss lessons learned from internal and external security incidents so all team members are aware of potential threats*		✓			
A48	*Document the most important security and privacy error and alert messages requiring user/operator attention*			✓		
A49	*Effectively communicate to company leadership the security risks and/or security relevant information assumed when using the program*	✓	✓	✓		✓
A50	*Effectively educate customers on the security risks and/or security relevant information assumed when using the program*	✓	✓	✓		✓
A51	*Effectively communicate to company leadership the cost/risk trade-off associated with deciding whether to fix identified problems*	✓	✓	✓		
A52	*Effectively communicate to customers the cost/risk trade-off associated with deciding whether patch their software*	✓	✓	✓		
A53	*Balance user functionality needs with security requirements*	✓		✓		
A54	*Maintain awareness of security issues with new hardware and software technologies and their potential implications*	✓	✓			
A55	*Maintain awareness of attack techniques used by a variety of malicious actors (e.g., insiders, nation states, cyber criminals)*		✓			
A56	*Communicate functionality needs to security experts to get recommendations for secure solutions (e.g., secure libraries and languages)*			✓		
A57	*Know the appropriate point of contact/response team in my organization to contact if a vulnerability in production code is identified*			✓		✓

[1] NICE, [2] BSIMM, [3] OSAMM, [4] SDL, [5] SAFECode

Table B.12: Additional initial secure development tasks identified through framework review. The SAFECode framework was not included in the initial review, but added after multiple expert recommendations.

Bibliography

[1] 2017. Association of Software Testing — Software Testing Professional Association. (2017). https://www.associationforsoftwaretesting.org/ (Accessed 06-08-2017).

[2] 2017. Ministry of Testing - Co-Creating Smart Testers. (2017). https://www.ministryoftesting.com/ (Accessed 06-08-2017).

[3] 2020. Binary Ninja Cloud. (2020). https://cloud.binary.ninja/ (Accessed 06-03-2020).

[4] Vector 35. 2020. Pwnadventure Sourcery. (2020). https://sourcery.pwnadventure.com/ (Accessed 05-27-2020).

[5] R. Abu-Salma, M. A. Sasse, J. Bonneau, A. Danilova, A. Naiakshina, and M. Smith. 2017. Obstacles to the Adoption of Secure Communication Tools. In *2017 IEEE Symposium on Security and Privacy (SP)*. 137–153. DOI:http://dx.doi.org/10.1109/SP.2017.65

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

[7] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. 2017b. Comparing the Usability of Cryptographic APIs. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (IEEE S&P)*. 154–171. DOI:http://dx.doi.org/10.1109/SP.2017.52

[8] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. 2016. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (IEEE S&P)*. 289–305. DOI:http://dx.doi.org/10.1109/SP.2016.25

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

[10] S. Akbari and M. B. Menhaj. 2000. A new framework of a decision support system for air to air combat tasks. In *ICSMCCCS '00*, Vol. 3. 2019–2022 vol.3. DOI:http://dx.doi.org/10.1109/ICSMC.2000.886411

[11] Mortada Al-Banna, Boualem Benatallah, and Moshe Chai Barukh. 2016. Software Security Professionals: Expertise Indicators. In *Proceedings of the 2nd IEEE International Conference on Collaboration and Internet Computing (CIC '16)*. Pittsburgh, PA, USA, 139–148. DOI:http://dx.doi.org/10.1109/CIC.2016.030

[12] A Algarni and Y Malaiya. 2014. Software vulnerability markets: Discoverers and buyers. *International Journal of Computer, Information Science and Engineering* 8, 3 (2014), 71–81.

[13] Susan A Ambrose, Michael W Bridges, Michele DiPietro, Marsha C Lovett, and Marie K Norman. 2010. *How learning works: Seven research-based principles for smart teaching*. John Wiley & Sons.

[14] angstromCTF. 2019. angstromCTF. (2019). https://angstromctf.com/ (Accessed 05-27-2020).

[15] John Annett. 2003. Hierarchical task analysis. *Handbook of cognitive task design* 2 (2003), 17–35.

[16] Nuno Antunes and Marco Vieira. 2009. Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services. In *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '09)*. IEEE Computer Society, Washington, DC, USA, 301–306. DOI:http://dx.doi.org/10.1109/PRDC.2009.54

[17] Jorge Aranda and Gina Venolia. 2009. The Secret Life of Bugs: Going Past the Errors and Omissions in Software Repositories. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 298–308. DOI:http://dx.doi.org/10.1109/ICSE.2009.5070530

[18] M. A. J. Arne Worm. 2000. Information-centered human-machine systems analysis for tactical command and control systems modeling and development. In *ICSMCCCS '00*, Vol. 3. 2240–2246 vol.3. DOI:http://dx.doi.org/10.1109/ICSMC.2000.886449

[19] Willem A. Arrindell and Jan van der Ende. 1985. An Empirical Test of the Utility of the Observations-To-Variables Ratio in Factor and Components Analysis. *Applied Psychological Measurement* 9, 2 (1985), 165–178. DOI:http://dx.doi.org/10.1177/014662168500900205

[20] Philippe Arteau, Andrey Loskutov, Juan Dodero, and Kengo Toda. 2019. SpotBugs. https://spotbugs.github.io/. (2019).

[21] Vairam Arunachalam and William Sasso. 1996. Cognitive Processes in Program Comprehension: An Empirical Analysis in the Context of Software Reengineering. *Journal on System Software* 34, 3 (Sept. 1996), 177–189. DOI:http://dx.doi.org/10.1016/0164-1212(95)00074-7

[22] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[23] Hala Assal and Sonia Chiasson. 2018. Security in the Software Development Lifecycle. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*. USENIX Association, Baltimore, MD, 281–296. https://www.usenix.org/conference/soups2018/presentation/assal

[24] Hala Assal and Sonia Chiasson. 2019. 'Think Secure from the Beginning': A Survey with Software Developers. In *Proceedings of the 37th CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Article 289, 13 pages. DOI:http://dx.doi.org/10.1145/3290605.3300519

[25] Andrew Austin and Laurie Williams. 2011. One Technique is Not Enough: A Comparison of Vulnerability Discovery Techniques. In *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement (ESEM '11)*. IEEE Computer Society, Washington, DC, USA, 97–106. DOI:http://dx.doi.org/10.1109/ESEM.2011.18

[26] Dejan Baca, Bengt Carlsson, Kai Petersen, and Lars Lundberg. 2013. Improving software security with static automated code analysis in an industry setting. *Software: Practice and Experience* 43, 3 (2013), 259–279. http://dblp.uni-trier.de/db/journals/spe/spe43.html#BacaCPL13

[27] Nathan Backman. 2016. Facilitating a Battle Between Hackers: Computer Security Outside of the Classroom. In *In Proc. of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE ?16)*. ACM, New York, NY, USA, 603?608. DOI:http://dx.doi.org/10.1145/2839509.2844648

[28] Andrew R Baggaley. 1983. Deciding on the ratio of number of subjects to number of variables in factor analysis. *Multivariate Experimental Clinical Research* (1983).

[29] Richard P. Bagozzi and Youjae Yi. 1988. On the evaluation of structural equation models. *Journal of the Academy of Marketing Science* 16, 1 (01 Mar 1988), 74–94. DOI:http://dx.doi.org/10.1007/BF02723327

[30] Rebecca Balebako, Abigail Marsh, Jialiu Lin, Jason I. Hong, and Lorrie Cranor. 2014. The Privacy and Security Behaviors of Smartphone App Developers. (2 2014). DOI:http://dx.doi.org/10.1184/R1/6470528.v1

[31] Sebastian Baltes and Stephan Diehl. 2018. Towards a Theory of Software Development Expertise. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 187–200. DOI:http://dx.doi.org/10.1145/3236024.3236061

[32] Albert Bandura. 1993. Perceived Self-Efficacy in Cognitive Development and Functioning. *Educational Psychologist* 28, 2 (1993), 117–148. DOI:http://dx.doi.org/10.1207/s15326985ep2802_3

[33] Lecia Jane Barker, Kathy Garvin-Doxas, and Michele Jackson. 2002. Defensive Climate in the Computer Science Classroom. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE ?02)*. Association for Computing Machinery, New York, NY, USA, 43?47. DOI:http://dx.doi.org/10.1145/563340.563354

[34] Paul T Barrett and Paul Kline. 1981. The observation to variable ratio in factor analysis. *Personality Study & Group Behaviour* (1981).

[35] Frederic Bartlett. 1932. C.(1932). Remembering. *Cambridge: CambridgeUniversityPress* (1932).

[36] Leilani Battle, Remco Chang, and Michael Stonebraker. 2016. Dynamic Prefetching of Data Tiles for Interactive Visualization. In *SIGMOD '16*. ACM, New York, NY, USA, 1363–1375. DOI:http://dx.doi.org/10.1145/2882903.2882919

[37] Leilani Battle and Jeffrey Heer. 2019. Characterizing Exploratory Visual Analysis: A Literature Review and Evaluation of Analytic Provenance in Tableau. *Computer Graphics Forum* (2019). http://idl.cs.washington.edu/papers/exploratory-visual-analysis

[38] Walter Baziuk. 1995. BNR/NORTEL: path to improve product quality, reliability and customer satisfaction. In *Sixth International Symposium on Software Reliability Engineering, ISSRE 1995, Toulouse, France, October 24-27, 1995*. 256–262. DOI:http://dx.doi.org/10.1109/ISSRE.1995.497665

[39] Steffen Becker, Carina Wiesen, Nils Albartus, Nikol Rummel, and Christof Paar. 2020. An Exploratory Study of Hardware Reverse Engineering — Technical and Cognitive Processes. In *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*. USENIX Association, 285–300. https://www.usenix.org/conference/soups2020/presentation/becker

[40] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)* 57, 1 (1995), 289–300. DOI:http://dx.doi.org/10.2307/2346101

[41] Peter M Bentler. 1990. Comparative fit indexes in structural models. *Psychological bulletin* 107, 2 (1990), 238.

[42] Peter M Bentler and Douglas G Bonett. 1980. Significance tests and goodness of fit in the analysis of covariance structures. *Psychological bulletin* 88, 3 (1980), 588.

[43] Jorge Blasco and Elizabeth A. Quaglia. 2018. InfoSec Cinema: Using Films for Information Security Teaching. In *2018 USENIX Workshop on Advances in Security Education (ASE 18)*. USENIX Association, Baltimore, MD. https://www.usenix.org/conference/ase18/presentation/blasco

[44] Kevin Bock, George Hughey, and Dave Levin. 2018. King of the Hill: A Novel Cybersecurity Competition for Teaching Penetration Testing. In *2018 USENIX Workshop on Advances in Security Education (ASE 18)*. USENIX Association, Baltimore, MD. https://www.usenix.org/conference/ase18/presentation/bock

[45] Kenneth A Bollen. 2014. *Structural equations with latent variables*. Vol. 210. John Wiley & Sons.

[46] Hack The Box. 2020. Hack The Box. (2020). https://www.hackthebox.eu (Accessed 05-27-2020).

[47] John D. Bransford, Ann L. Brown, and Rodney R. Cocking. 2000. *How people learn: Brain, mind, experience, and school: Expanded edition*. National Academies Press.

[48] S. Bratus. 2007. What Hackers Learn that the Rest of Us Don't: Notes on Hacker Curriculum. *IEEE Security Privacy* 5, 4 (2007), 72–75.

[49] Richard W Brislin. 1980. Cross-cultural research methods. In *Environment and culture*. Springer, 47–82.

[50] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* 18, 6 (1983), 543 – 554. DOI:http://dx.doi.org/https://doi.org/10.1016/S0020-7373(83)80031-5

[51] Ann L. Brown. 1975. The Development of Memory: Knowing, Knowing About Knowing, and Knowing How to Know. Advances in Child Development and Behavior, Vol. 10. JAI, 103 – 152. DOI:http://dx.doi.org/https://doi.org/10.1016/S0065-2407(08)60009-9

[52] Adam Bryant. 2012. *Understanding How Reverse Engineers Make Sense of Programs from Assembly Language Representations*. Ph.D. Dissertation. US Air Force Institute of Technology.

[53] Bugcrowd. 2015. *Defensive Vulnerability Pricing Model*. Technical Report. Bugcrowd. https://pages.bugcrowd.com/whats-a-bug-worth-2015-survey

[54] Bugcrowd. 2016a. Home - Bugcrowd. (2016). http://bugcrowd.com (Accessed 02-18-2017).

[55] Bugcrowd. 2016b. *The State of Bug Bounty*. Technical Report. Bugcrowd. https://pages.bugcrowd.com/2016-state-of-bug-bounty-report

[56] BugCrowd. 2020. Inside the mind of a hacker. (2020). https://itmoah.bugcrowd.com/. (Accessed 10-19-2017).

[57] Jean-Marie Burkhardt, Françoise Détienne, and Susan Wiedenbeck. 2002. Object-Oriented Program Comprehension: Effect of Expertise, Task and Phase. *Empirical Software Engineering* 7, 2 (01 Jun 2002), 115–156. DOI:http://dx.doi.org/10.1023/A:1015297914742

[58] 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. 2017. *Curriculum Guidelines for Post-Secondary Degree Programs in Cybersecurity*. Technical Report. ACM, IEEE, AIS, and IFIP. 32–36 pages. https://cybered.hosting.acm.org/wp-content/uploads/2018/02/newcover_csec2017.pdf

[59] Tanner J. Burns, Samuel C. Rios, Thomas K. Jordan, Qijun Gu, and Trevor Underwood. 2017. Analysis and Exercises for Engaging Beginners in Online CTF Competitions for Security Education. In *2017 USENIX Workshop on Advances in Security Education (ASE 17)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/ase17/workshop-program/presentation/burns

[60] John T. Cacioppo, Richard E. Petty, and Chuan Feng Kao. 1984. The Efficient Assessment of Need for Cognition. *Journal of Personality Assessment* 48, 3 (1984), 306–307. DOI:http://dx.doi.org/10.1207/s15327752jpa4803_13 PMID: 16367530.

[61] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, 209–224.

[62] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, 459–465.

[63] A Colin Cameron and Pravin K Trivedi. 2013. *Regression analysis of count data*. Vol. 53. Cambridge university press.

[64] Susan G. Campbell, Polly OâRourke, and Michael F. Bunting. 2015. Identifying Dimensions of Cyber Aptitude: The Design of the Cyber Aptitude and Talent Assessment. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 59, 1 (2015), 721–725. DOI:http://dx.doi.org/10.1177/1541931215591170

[65] Janis A Cannon-Bowers and Eduardo Ed Salas. 1998. *Making decisions under stress: Implications for individual and team training*. American psychological association.

[66] Edward G Carmines and John McIver. 1981. Analyzing models with unobserved variables. *Social measurement: Current issues* 80 (1981).

[67] Raymond B. Cattell. 1952. *Factor analysis: An introduction and manual for the psychologist and social scientist*. Harper & Row, New York.

[68] M. Ceccato, P. Tonella, C. Basile, B. Coppens, B. De Sutter, P. Falcarin, and M. Torchiano. 2017. How Professional Hackers Understand Protected Code While Performing Attack Tasks. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*. IEEE Press, Piscataway, NJ, USA, 154–164. DOI:http://dx.doi.org/10.1109/ICPC.2017.2

[69] Maryland Cybersecurity Center. 2020. Build it Break it Fix it. (2020). https://builditbreakit.org/ (Accessed 05-27-2020).

[70] Center for Cyber Safety and Education. 2017. *Global Information Security Workforce Study*. Technical Report. Center for Cyber Safety and Education, Clearwater, FL. https://iamcybersafe.org/wp-content/uploads/2017/07/N-America-GISWS-Report.pdf

[71] National Cyber Security Centre. 2017. NCSC certified degrees. (2017). https://www.ncsc.gov.uk/information/ncsc-certified-degrees (Accessed 04-01-2019).

[72] Barbara A. Cerny and Henry F. Kaiser. 1977. A Study Of A Measure Of Sampling Adequacy For Factor-Analytic Correlation Matrices. *Multivariate Behavioral Research* 12, 1 (1977), 43–47. DOI: http://dx.doi.org/10.1207/s15327906mbr1201_3 PMID: 26804143.

[73] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, Washington, DC, USA, 380–394. http://dx.doi.org/10.1109/SP.2012.31

[74] Pravir Chandra. 2017. *Software Assurance Maturity Model*. Technical Report. Open Web Application Security Project.

[75] Yung-Yu Chang, Pavol Zavarsky, Ron Ruhl, and Dale Lindskog. 2011. Trend analysis of the CVE for software vulnerability management. In *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on*. IEEE, 1290–1293.

[76] Wu chang Feng, Robert Liebman, Lois Delcambre, Michael Lupro, Tim Sheard, Scott Britell, and Gerald Recktenwald. 2017. CyberPDX: A Camp for Broadening Participation in Cybersecurity. In *2017 USENIX Workshop on Advances in Security Education (ASE 17)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/ase17/workshop-program/presentation/feng

[77] Peter Chapman, Jonathan Burket, and David Brumley. 2014. PicoCTF: A Game-Based Computer Security Competition for High School Students. In *Proc. of the 1st USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE '14)*. USENIX Association, San Diego, CA. https://www.usenix.org/conference/3gse14/summit-program/presentation/chapman

[78] Kathy Charmaz. 2006. *Constructing Grounded Theory: A Practical Guide Through Qualitative Analysis*. SagePublication Ltd, London. http://www.amazon.com/Constructing-Grounded-Theory-Qualitative-Introducing/dp/0761973532

[79] William G Chase and Herbert A Simon. 1973. Perception in chess. *Cognitive psychology* 4, 1 (1973), 55–81.

[80] Ronald S. Cheung, Joseph P. Cohen, Henry Z. Lo, Fabio Elia, and Veronica Carrillo-Marquez. 2012. Effectiveness of Cybersecurity Competitions. In *Proceedings of the International Conference on Security and Management (SAM '12)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 1–5.

[81] E. J. Chikofsky and J. H. Cross. 1990. Reverse engineering and design recovery: a taxonomy. *IEEE Software* 7, 1 (Jan 1990), 13–17. DOI:http://dx.doi.org/10.1109/52.43044

[82] Steve Christey and Robert A Martin. 2007. Vulnerability type distributions in CVE. (2007).

[83] Kevin Chung and Julian Cohen. 2014. Learning Obstacles in the Capture The Flag Model. In *Proceedings of the 1st USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE '14)*. USENIX Association, San Diego, CA. https://www.usenix.org/conference/3gse14/summit-program/presentation/chung

[84] Paul Cobb, Erna Yackel, and Terry Wood. 1992. A constructivist alternative to the representational view of mind in mathematics education. *Journal for research in mathematics education* (1992), 2–33.

[85] J. Cohen. 1988. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates.

[86] Jacob Cohen. 2013. *Statistical power analysis for the behavioral sciences*. Routledge.

[87] CollabNet. 2019. 13th Annual State of Agile Report. https://www.stateofagile.com/#ufh-i-521251909-13th-annual-state-of-agile-report/473508. (2019).

[88] Andrew L Comrey. 1973. *A first course in factor analysis*. Academic Press, New York.

[89] Andrew L Comrey. 1978. Common methodological problems in factor analytic studies. *Journal of consulting and clinical psychology* 46, 4 (1978), 648.

[90] G. Conti, T. Babbitt, and J. Nelson. 2011. Hacking Competitions and Their Untapped Potential for Security Education. *IEEE Security Privacy* 9, 3 (2011), 56–59.

[91] Jennifer Cowley. 2014. *Job Analysis Results for Malicious-Code Reverse Engineers: A Case Study*. Technical Report CMU/SEI-2014-TR-002. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=91548

[92] Harald Cramér. 2016. *Mathematical methods of statistics (PMS-9)*. Vol. 9. Princeton university press.

[93] Catherine Crouch, Adam P. Fagen, J. Paul Callan, and Eric Mazur. 2004. Classroom demonstrations: Learning tools or entertainment? *American Journal of Physics* 72, 6 (2004), 835–838. DOI:http://dx.doi.org/10.1119/1.1707018

[94] Douglas P Crowne and David Marlowe. 1960. A new scale of social desirability independent of psychopathology. *Journal of consulting psychology* 24, 4 (1960), 349.

[95] CTFLearn. 2020. CTFLearn. (2020). https://ctflearn.com/ (Accessed 05-27-2020).

[96] CTFTime. 2017a. CTF? WTF? (2017). https://ctftime.org/ctf-wtf/ (Accessed 06-08-2017).

[97] CTFTime. 2017b. CTFTime.org / All About CTF (Capture-the-flag). (2017). https://ctftime.org (Accessed 06-08-2017).

[98] CyberTalents. 2020. Cyber Talents Practice. (2020). https://cybertalents.com/challenges (Accessed 05-27-2020).

[99] Tamara Denning, Adam Shostack, and Tadayoshi Kohno. 2014. Practical Lessons from Creating the Control-Alt-Hack Card Game and Research Challenges for Games In Education and Research. In *2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14)*. USENIX Association, San Diego, CA. https://www.usenix.org/conference/3gse14/summit-program/presentation/denning

[100] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep Me Updated: An Empirical Study of Third-Party Library Updatability on Android. In *Proceedings of the 24th Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 2187–2200. DOI:http://dx.doi.org/10.1145/3133956.3134059

[101] Pranita Deshpande, Cynthia B. Lee, and Irfan Ahmed. 2019. Evaluation of Peer Instruction for Cybersecurity Education. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, New York, NY, USA, 720–725. DOI:http://dx.doi.org/10.1145/3287324.3287403

[102] Francoise Detienne. 1990. Chapter 3.1 - Expert Programming Knowledge: A Schema-based Approach. In *Psychology of Programming*, J.-M. Hoc, T.R.G. Green, R. Samurçay, and D.J. Gilmore (Eds.). Academic Press, London, 205 – 222. DOI:http://dx.doi.org/https://doi.org/10.1016/B978-0-12-350772-3.50018-5

[103] Constanze Dietrich, Katharina Krombholz, Kevin Borgolte, and Tobias Fiebig. 2018. Investigating System Operators' Perspective on Security Misconfigurations. In *Proc. of the 25th ACM Conference on Computer and Communications Security (CCS '18)*. ACM.

[104] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. 2014. Explore-by-example: An Automatic Query Steering Framework for Interactive Data Exploration. In *SIGMOD '14*. ACM, New York, NY, USA, 517–528. DOI:http://dx.doi.org/10.1145/2588555.2610523

[105] Felix Dörre and Vladimir Klebanov. 2016. Practical Detection of Entropy Loss in Pseudo-Random Number Generators. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. 678–689.

[106] Adam Doupé, Marco Cova, and Giovanni Vigna. 2010. Why Johnny Can'T Pentest: An Analysis of Black-box Web Vulnerability Scanners. In *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'10)*. Springer-Verlag, Berlin, Heidelberg, 111–131. http://dl.acm.org/citation.cfm?id=1884848.1884858

[107] Adam Doupé, Manuel Egele, Benjamin Caillat, Gianluca Stringhini, Gorkem Yakin, Ali Zand, Ludovico Cavedon, and Giovanni Vigna. 2011. Hit 'em Where It Hurts: A Live Security Exercise on Cyber Situational Awareness. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC ?11)*. Association for Computing Machinery, New York, NY, USA, 51?61. DOI:http://dx.doi.org/10.1145/2076732.2076740

[108] Jason Doyle. 2017. Google-Nest-Cam-Bug-Disclosures. (2017). https://github.com/jasondoyle/Google-Nest-Cam-Bug-Disclosures (Accessed 08-1-2017).

[109] Will Drewry and Tavis Ormandy. 2007. Flayer: Exposing Application Internals. In *WOOT '07*.

[110] Wenliang Du. 2011. SEED: hands-on lab exercises for computer security education. *IEEE Security & Privacy* 9, 5 (2011), 70–73.

[111] John E Dunn. 2019. Medtronic cardiac implants can be hacked, FDA issues alert. (2019). https://nakedsecurity.sophos.com/2019/03/25/medtronic-cardiac-implants-can-be-hacked-fda-issues-alert/

[112] C. Eagle. 2013. Computer Security Competitions: Expanding Educational Outcomes. *IEEE Security Privacy* 11, 4 (July 2013), 69–71. DOI:http://dx.doi.org/10.1109/MSP.2013.83

[113] Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Matthew Finifter, Adrian Mettler, and David Wagner. 2013. An Empirical Study on the Effectiveness of Security Code Review. In *Proceedings of the 5th International Conference on Engineering Secure Software and Systems (ESSoS'13)*. Springer-Verlag, Berlin, Heidelberg, 197–212. DOI:http://dx.doi.org/10.1007/978-3-642-36563-8_14

[114] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 73–84.

[115] Serge Egelman and Eyal Peer. 2015. Scaling the Security Wall: Developing a Security Behavior Intentions Scale (SeBIS). In *Proceedings of the 33rd CHI Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 2873–2882. DOI:http://dx.doi.org/10.1145/2702123.2702249

[116] Eldad Eilam. 2011. *Reversing: secrets of reverse engineering*. John Wiley & Sons.

[117] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.* 32, 2, Article 5 (2014), 5:1–5:29 pages.

[118] K Anders Ericsson and Neil Charness. 1994. Expert performance: Its structure and acquisition. *American psychologist* 49, 8 (1994), 725.

[119] K Anders Ericsson, Ralf T Krampe, and Clemens Tesch-Römer. 1993. The role of deliberate practice in the acquisition of expert performance. *Psychological review* 100, 3 (1993), 363.

[120] Brian S Everitt and Anders Skrondal. 2010. *The Cambridge dictionary of statistics*. New York University.

[121] Leandre R Fabrigar, Duane T Wegener, Robert C MacCallum, and Erin J Strahan. 1999. Evaluating the use of exploratory factor analysis in psychological research. *Psychological methods* 4, 3 (1999), 272.

[122] Peter J. Fadde and Gary Klein. 2012. Accelerating Expertise Using Action Learning Activities. *Cognitive Technology* 17, 1 (2012), 11–18.

[123] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 50–61.

[124] Cori Faklaris, Laura A. Dabbish, and Jason I. Hong. 2019. A Self-Report Measure of End-User Security Attitudes (SA-6). In *Proceedings of the 5th Symposium on Usable Privacy and Security (SOUPS '19)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/soups2019/presentation/faklaris

[125] Ming Fang and Munawar Hafiz. 2014. Discovering Buffer Overflow Vulnerabilities in the Wild: An Empirical Study. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '14)*. ACM, New York, NY, USA, Article 23, 10 pages. DOI:http://dx.doi.org/10.1145/2652524.2652533

[126] Matthew Field. 2018. WannaCry cyber attack cost the NHS £92m as 19,000 appointments cancelled. (2018). https://www.telegraph.co.uk/technology/2018/10/11/wannacry-cyber-attack-cost-nhs-92m-19000-appointments-cancelled/

[127] Matthew Finifter, Devdatta Akhawe, and David Wagner. 2013. An empirical study of vulnerability rewards programs. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security '13)*. 273–288.

[128] Matthew Finifter and David Wagner. 2011. Exploring the relationship between web application development tools and security. In *USENIX conference on Web application development*.

[129] FIRST.org. 2016. Common Vulnerability Scoring System. (2016). https://www.first.org/cvss/calculator/3.0 (Accessed 12-19-2016).

[130] Ronald A Fisher. 1922. On the interpretation of χ 2 from contingency tables, and the calculation of P. *Journal of the Royal Statistical Society* 85, 1 (1922), 87–94.

[131] John H. Flavell. 1976. Metacognitive aspects of problem solving. In *The nature of intelligence*, Lauren B. Resnick (Ed.). Lawrence Erlbaum Associates.

[132] The Association for Computing Machinery. 2019. Chapters. `https://acm.org/chapters`. (2019).

[133] ForAllSecure. 2019. ForAllSecure. (2019). `https://forallsecure.com/` (Accessed 05-30-2019).

[134] Claes Fornell and David F. Larcker. 1981. Evaluating Structural Equation Models with Unobservable Variables and Measurement Error. *Journal of Marketing Research* 18, 1 (1981), 39–50. `DOI:http://dx.doi.org/10.1177/002224378101800104`

[135] Gordon Fraser, Alessio Gambi, Marvin Kreis, and José Miguel Rojas. 2019a. Gamifying a Software Testing Course with Code Defenders. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, New York, NY, USA, 571–577. `DOI:http://dx.doi.org/10.1145/3287324.3287471`

[136] Gordon Fraser, Alessio Gambi, Marvin Kreis, and José Miguel Rojas. 2019b. Gamifying a Software Testing Course with Code Defenders. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE ?19)*. Association for Computing Machinery, New York, NY, USA, 571?577. `DOI:http://dx.doi.org/10.1145/3287324.3287471`

[137] Dustin Fraze. 2017. Computer and Humans Exploring Software Security (CHESS). (2017). `https://www.darpa.mil/program/computers-and-humans-exploring-software-security` (Accessed 05-31-2019).

[138] Deen G Freelon. 2010. ReCal: Intercoder reliability calculation as a web service. *International Journal of Internet Science* 5, 1 (2010), 20–33.

[139] Sylvain Frey, Awais Rashid, Pauline Anthonysamy, Maria Pinto-Albuquerque, and Syed Asad Naqvi. 2019. The Good, the Bad and the Ugly: A Study of Security Decisions in a Cyber-Physical Systems Game. *IEEE Transactions on Software Engineering* 45, 5 (2019), 521–536.

[140] Karl Pearson F.R.S. 1900. X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *Philos. Mag.* 50, 302 (1900), 157–175.

[141] Fyodor. 2017. Full Disclosure Mailing List. (2017). `http://seclists.org/fulldisclosure/` (Accessed 08-01-2017).

[142] g0tmi1k. 2020. Vulnhub. (2020). `https://www.vulnhub.com/` (Accessed 05-27-2020).

[143] Markus Gaasedelen. 2018. Lighthouse — Code Coverage Explorer for IDA Pro & Binary Ninja. (2018). `https://github.com/gaasedelen/lighthouse` (Accessed 08-21-2019).

[144] Kathy Garvin-Doxas and Lecia J. Barker. 2004. Communication in Computer Science Classrooms: Understanding Defensive Climates as a Means of Creating Supportive Behaviors. *J. Educ. Resour. Comput.* 4, 1 (March 2004), 2?es. `DOI:http://dx.doi.org/10.1145/1060071.1060073`

[145] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 38–49. `DOI:http://dx.doi.org/10.1145/2382196.2382204`

[146] Thomas Gilray, Michael D. Adams, and Matthew Might. 2016. Allocation Characterizes Polyvariance: A Unified Methodology for Polyvariant Control-flow Analysis. *SIGPLAN Notes* 51, 9 (Sept. 2016), 407–420. `DOI:http://dx.doi.org/10.1145/3022670.2951936`

[147] Jennifer Goldbeck, Jonathan Katz, Michael Hicks, and Gang Qu. 2019. Coursera Cybersecurity Specialization. `https://www.coursera.org/specializations/cyber-security`. (2019).

[148] Google. 2016. Issues - chromium. (2016). `https://bugs.chromium.org/p/chromium/issues/list` (Accessed 02-18-2017).

[149] Google. 2019. Google CTF 2019. (2019). `https://g.co/ctf` (Accessed 05-27-2020).

[150] Google. 2020. XSS Game. (2020). `https://xss-game.appspot.com/` (Accessed 05-27-2020).

[151] Adam Gordon. 2015. *Official (ISC) 2 guide to the CISSP CBK*. Auerbach Publications.

[152] Richard L. Gorsuch. 1978. *Factor Analysis* (2nd ed.). Erlbaum, Hillsdale, NJ.

[153] Richard L. Gorsuch. 1988. *Exploratory Factor Analysis*. Springer US, Boston, MA, 231–258. DOI: http://dx.doi.org/10.1007/978-1-4613-0893-5_6

[154] David Gotz and Zhen Wen. 2009. Behavior-driven Visualization Recommendation. In *IUI '09*. ACM, New York, NY, USA, 315–324. DOI:http://dx.doi.org/10.1145/1502650.1502695

[155] Matthew Green and Matthew Smith. 2016. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy* 14, 5 (2016), 40–46.

[156] NCC Group. 2014. Embedded Security CTF. (2014). https://microcorruption.com/ (Accessed 05-27-2020).

[157] Edward Guadagnoli and Wayne F Velicer. 1988. Relation of sample size to the stability of component patterns. *Psychological bulletin* 103, 2 (1988), 265.

[158] Greg Guest, Arwen Bunce, and Laura Johnson. 2006. How many interviews are enough? An experiment with data saturation and variability. *Field methods* 18, 1 (2006), 59–82.

[159] L. Gugerty and G. Olson. 1986. Debugging by Skilled and Novice Programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '86)*. ACM, New York, NY, USA, 171–174. DOI:http://dx.doi.org/10.1145/22627.22367

[160] Joy Paul Guilford. 1954. Psychometric methods. (1954).

[161] HackEDU. 2020. HackEDU. (2020). https://www.hackedu.com/ (Accessed 05-27-2020).

[162] HackerOne. Home — Hacker 101. (????). https://www.hacker101.com/ (Accessed 05-21-2020).

[163] Hackerone. 2016. *2016 Bug Bounty Hacker Report*. Technical Report. Hackerone, San Francisco, California. https://hackerone.com/blog/bug-bounty-hacker-report-2016

[164] HackerOne. 2016. HackerOne: Vulnerability Coordination and Bug Bounty Platform. (2016). http://hackerone.com (Accessed 02-18-2017).

[165] HellBound Hackers. 2020. HellBound Hackers. (2020). https://www.hellboundhackers.org/ (Accessed 05-27-2020).

[166] HackerTest. 2020. Hacker Test. (2020). http://www.hackertest.net/ (Accessed 05-27-2020).

[167] HackThisSite. 2016. HackThisSite. (2016). https://www.hackthissite.org/ (Accessed 05-27-2020).

[168] Munawar Hafiz and Ming Fang. 2016. Game of detections: how are security vulnerabilities discovered in the wild? *Empirical Software Engineering* 21, 5 (2016), 1920–1959. DOI:http://dx.doi.org/10.1007/s10664-015-9403-7

[169] Joseph F Hair, William C Black, Barry J Babin, Rolph E Anderson, Ronald L Tatham, and others. 2006. Multivariate data analysis. (2006).

[170] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *USENIX Security '13*. USENIX, Washington, D.C., 49–64.

[171] Joseph Hallett, Robert Larson, and Awais Rashid. 2018. Mirror, Mirror, On the Wall: What are we Teaching Them All? Characterising the Focus of Cybersecurity Curricular Frameworks. In *2018 USENIX Workshop on Advances in Security Education (ASE 18)*. USENIX Association, Baltimore, MD. https://www.usenix.org/conference/ase18/presentation/hallett

[172] Julie Haney and Wayne Lutters. 2017. Skills and Characteristics of Successful Cybersecurity Advocates. In *Proceedings of the 3rd Workshop on Security Information Workers (SOUPS '17)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/soups2017/workshop-program/wsiw2017/haney

[173] Julie M. Haney and Wayne G. Lutters. 2018. "It's Scary... It's Confusing... It's Dull": How Cybersecurity Advocates Overcome Negative Perceptions of Security. In *Proceedings of the 14th Symposium on Usable Privacy and Security (SOUPS '18)*. USENIX Association, Baltimore, MD, 411–425. https://www.usenix.org/conference/soups2018/presentation/haney-perceptions

[174] Julie M. Haney, Mary Theofanos, Yasemin Acar, and Sandra Spickard Prettyman. 2018. "We make it a big deal in the company": Security Mindsets in Organizations that Develop Cryptographic Products. In *Proceedings of the 14th Symposium on Usable Privacy and Security (SOUPS '18)*. USENIX Association, Baltimore, MD, 357–373. https://www.usenix.org/conference/soups2018/presentation/haney-mindsets

[175] Allen Harper, Shon Harris, Jonathan Ness, Chris Eagle, Gideon Lenkey, and Terron Williams. 2018. *Gray hat hacking: the ethical hacker's handbook* (3rd ed.). McGraw-Hill Education.

[176] Margaret C Harrell and Melissa A Bradley. 2009. *Data collection methods. Semi-structured interviews and focus groups.* Technical Report. Rand National Defense Research Institute.

[177] William R. Harris, Somesh Jha, Thomas W. Reps, and Sanjit A. Seshia. 2017. Program Synthesis for Interactive-security Systems. *Form. Methods Syst. Des.* 51, 2 (Nov. 2017), 362–394. DOI:http://dx.doi.org/10.1007/s10703-017-0296-5

[178] Hope J. Hartman. 1998. Metacognition in teaching and learning: An introduction. *Instructional Science* 26, 1/2 (1998), 1–3. http://www.jstor.org/stable/23371261

[179] Andrew F Hayes and Klaus Krippendorff. 2007. Answering the call for a standard reliability measure for coding data. *Communication methods and measures* 1, 1 (2007), 77–89. http://dx.doi.org/10.1080/19312450709336664

[180] J. Heer, J. Mackinlay, C. Stolte, and M. Agrawala. 2008. Graphical Histories for Visualization: Supporting Analysis, Communication, and Evaluation. *IEEE Transactions on Visualization and Computer Graphics* 14, 6 (Nov 2008), 1189–1196. DOI:http://dx.doi.org/10.1109/TVCG.2008.137

[181] Jeffrey Heer and Ben Shneiderman. 2012. Interactive Dynamics for Visual Analysis. *Commun. ACM* 55, 4 (April 2012), 45–54. DOI:http://dx.doi.org/10.1145/2133806.2133821

[182] Julien Henry, David Monniaux, and Matthieu Moy. 2012. PAGAI: A Path Sensitive Static Analyser. *Electronic Notes in Theoretical Computer Science* 289 (Dec. 2012), 15–25. DOI:http://dx.doi.org/10.1016/j.entcs.2012.11.003

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

[184] Hex-Rays. 2017. IDA: Lumina Server. (2017). https://www.hex-rays.com/products/ida/lumina/index.shtml (Accessed 01-06-2019).

[185] Hex-Rays. 2019a. Hex-Rays Decompiler: Overview. (2019). https://www.hex-rays.com/products/decompiler/ (Accessed 11-11-2019).

[186] Hex-Rays. 2019b. IDA: About. (2019). https://www.hex-rays.com/products/ida/ (Accessed 05-30-2019).

[187] Hex-Rays. 2019c. Plug-in Contest 2018: Hall of Fame. (2019). https://www.hex-rays.com/contests/2018/index.shtml (Accessed 05-30-2019).

[188] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in Continuous Integration: Assurance, Security, and Flexibility. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*. ACM, New York, NY, USA, 197–207. DOI:http://dx.doi.org/10.1145/3106237.3106270

[189] Timothy R. Hinkin, J. Bruce Tracey, and Cathy A. Enz. 1997. Scale Construction: Developing Reliable and Valid Measurement Instruments. *Journal of Hospitality & Tourism Research* 21, 1 (1997), 100–120. DOI:http://dx.doi.org/10.1177/109634809702100108

[190] Allyson L Holbrook, Melanie C Green, and Jon A Krosnick. 2003. Telephone versus face-to-face interviewing of national probability samples with long questionnaires: Comparisons of respondent satisficing and social desirability response bias. *Public opinion quarterly* 67, 1 (2003), 79–125.

[191] Sture Holm. 1979. A Simple Sequentially Rejective Multiple Test Procedure. *Scandinavian Journal of Statistics* 6, 2 (1979), 65–70. http://www.jstor.org/stable/4615733

[192] Thomas J Holt. 2009a. The attack dynamics of political and religiously motivated hackers. *Cyber Infrastructure Protection* (2009), 161–182.

[193] Thomas J Holt. 2009b. Lone hacks or group cracks: Examining the social organization of computer hackers. *Crimes of the Internet* (2009), 336–355.

[194] John L. Horn. 1965. A rationale and test for the number of factors in factor analysis. *Psychometrika* 30, 2 (01 Jun 1965), 179–185. DOI:http://dx.doi.org/10.1007/BF02289447

[195] Sam Houston. 2016. Researcher Resources - How to become a Bug Bounty Hunter. (2016). https://forum.bugcrowd.com/t/researcher-resources-how-to-become-a-bug-bounty-hunter/1102 (Accessed 05-21-2020).

[196] David Hovemeyer and William Pugh. 2004. Finding Bugs is Easy. *SIGPLAN Not.* 39, 12 (Dec. 2004), 92–106. DOI:http://dx.doi.org/10.1145/1052883.1052895

[197] David Hovemeyer and William Pugh. 2007. Finding More Null Pointer Bugs, but Not Too Many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '07)*. ACM, New York, NY, USA, 9–14. DOI:http://dx.doi.org/10.1145/1251535.1251537

[198] Gary Hsieh and Rafal Kocielnik. 2016. You Get Who You Pay for: The Impact of Incentives on Participation Bias. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing (CSCW '16)*. ACM, New York, NY, USA, 823–835. DOI:http://dx.doi.org/10.1145/2818048.2819936

[199] K. Huang, M. Siegel, S. Madnick, X. Li, and Z. Feng. 2016. Poster: Diversity or concentration? Hackers' strategy for working across multiple bug bounty programs. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (IEEE S&P '16)*.

[200] Sylvia Hurtado, Jeffrey Milem, Alma Clayton-Pedersen, and Walter Allen. 1999. Enacting Diverse Learning Environments: Improving the Climate for Racial/Ethnic Diversity in Higher Education. *ASHE-ERIC Higher Education Report* 26, 8 (1999).

[201] HXP. 2020. HXP CTF. (2020). https://2018.ctf.link/ (Accessed 05-27-2020).

[202] Luigi Lo Iacono and Peter Leo Gorski. 2017. I Do and I Understand. Not Yet True for Security APIs. So Sad. In *Proceedings of the 2nd European Workshop on Usable Security (EuroUSEC '17)*. Internet Society. https://doi.org/10.14722/eurousec

[203] IEEE. 2018. IEEE Spectrum: The Top Programming Languages 2018. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018. (2018).

[204] IISP. 2017. *IISP Knowledge Framework*. Technical Report. IISP. https://www.iisp.org/imis15/iisp/About_Us/Our_Knowledge_Framework/iisp/About_Us/Our_Knowledge_Framework.aspx?hkey=6e8644f9-fc2f-4f53-9784-b0fb2dba5e8b

[205] InfoSec Institute. 2020a. n00bs CTF Labs. (2020). http://ctf.infosecinstitute.com/index.php (Accessed 05-27-2020).

[206] SANS Cybersecurity Institute. 2020b. Girls Go Cyberstart. (2020). https://girlsgocyberstart.org/ (Accessed 05-27-2020).

[207] T. j. Jankun-Kelly, K. Ma, and M. Gertz. 2007. A Model and Framework for Visualization Exploration. *IEEE Transactions on Visualization and Computer Graphics* 13, 2 (March 2007), 357–369.

[208] Ge Jin, Manghui Tu, Tae-Hoon Kim, Justin Heffron, and Jonathan White. 2018. Game Based Cybersecurity Training for High School Students. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, NY, USA, 68–73. DOI:http://dx.doi.org/10.1145/3159450.3159591

[209] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don'T Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 672–681. http://dl.acm.org/citation.cfm?id=2486788.2486877

[210] William E. Johnson, Allison Luzader, Irfan Ahmed, Vassil Roussev, Golden G. Richard III, and Cynthia B. Lee. 2016. Development of Peer Instruction Questions for Cybersecurity Education. In *2016 USENIX Workshop on Advances in Security Education (ASE 16)*. USENIX Association, Austin, TX. https://www.usenix.org/conference/ase16/workshop-program/presentation/johnson

[211] Melanie Jones. 2019. Why cybersecurity education matters. (2019). https://www.itproportal.com/features/why-cybersecurity-education-matters/

[212] Karl G Jöreskog and Dag Sörbom. 1993. *LISREL 8: Structural equation modeling with the SIMPLIS command language*. Scientific Software International.

[213] Alexander Kalinin, Ugur Cetintemel, and Stan Zdonik. 2014. Interactive Data Exploration Using Semantic Windows. In *SIGMOD '14*. ACM, New York, NY, USA, 505–516. DOI:http://dx.doi.org/10.1145/2588555.2593666

[214] Karthik Kannan and Rahul Telang. 2005. Market for Software Vulnerabilities? Think Again. *Manage. Sci.* 51, 5 (May 2005), 726–740. DOI:http://dx.doi.org/10.1287/mnsc.1040.0357

[215] Stylianos Karagiannis, Elpidoforos Maragkos-Belmpas, and Emmanouil Magkos. 2020. An Analysis and Evaluation of Open Source Capture the Flag Platforms as Cybersecurity e-Learning Tools. In *Information Security Education. Information Security in Action*, Lynette Drevin, Suné Von Solms, and Marianthi Theocharidou (Eds.). Springer International Publishing, Cham, 61–77.

[216] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-sensitivity for Points-to Analysis. *SIGPLAN Notes* 48, 6 (June 2013), 423–434. DOI:http://dx.doi.org/10.1145/2499370.2462191

[217] David Keller. 2020. Mr Code's Wild Ride. (2020). https://www.mrcodeswildride.com/hacking (Accessed 05-27-2020).

[218] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. 2012. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE '12)*. 121–132.

[219] Jason Kick. 2014. *Cyber exercise playbook*. Technical Report. MITRE CORP BEDFORD MA.

[220] Ada S. Kim and Andrew J. Ko. 2017. A Pedagogical Analysis of Online Coding Tutorials. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE ?17)*. Association for Computing Machinery, New York, NY, USA, 321?326. DOI: http://dx.doi.org/10.1145/3017680.3017728

[221] G. Klein, D. Klinger, and T. Miller. 1997. Using decision requirements to guide the design process. In *ICSMCCCS '97*, Vol. 1. 238–244 vol.1. DOI:http://dx.doi.org/10.1109/ICSMC.1997.625756

[222] Gary A Klein. 1989. Recognition-primed decisions. *Advances in man-machine systems research* 5 (1989), 47–92.

[223] Gary A Klein. 2017. *Sources of power: How people make decisions*. MIT press.

[224] Gary A. Klein and Christopher P. Brezovic. 1986. Design Engineers and the Design Process: Decision Strategies and Human Factors Literature. *Proceedings of the Human Factors Society Annual Meeting* 30, 8 (1986), 771–775. DOI:http://dx.doi.org/10.1177/154193128603000809

[225] Gary A Klein, Roberta Calderwood, and Anne Clinton-Cirocco. 1986. Rapid decision making on the fire ground. In *Proceedings of the 30th Human Factors Society Annual Meeting (HFES '86)*, Vol. 30. Sage Publications Sage CA: Los Angeles, CA, 576–580.

[226] Gary A Klein, Roberta Calderwood, and Donald Macgregor. 1989. Critical decision method for eliciting knowledge. *IEEE Transactions on Systems, Man, and Cybernetics* 19, 3 (1989), 462–472.

[227] D. W. Klinger, R. Stottler, and S. R. LeClair. 1992. Manufacturing application of case-based reasoning. In *NAECON '92*. 855–859 vol.3.

[228] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32, 12 (Dec. 2006), 971–987. DOI:http://dx.doi.org/10.1109/TSE.2006.116

[229] Lindsay Kolowich. 2017. The Demographics of Developers Around the World. https://blog.hubspot.com/marketing/developers-demographic-survey. (2017).

[230] Klaus Krippendorff. 2004. Reliability in Content Analysis. *Human Communication Research* 30, 3 (2004), 411–433. DOI:http://dx.doi.org/10.1111/j.1468-2958.2004.tb00738.x

[231] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. 2017. CrySL: Validating Correct Usage of Cryptographic APIs. *CoRR* abs/1710.00564 (2017). http://arxiv.org/abs/1710.00564

[232] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. 2018. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Todd Millstein (Ed.), Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 10:1–10:27. DOI:http://dx.doi.org/10.4230/LIPIcs.ECOOP.2018.10

[233] Joseph T. Kunce, Daniel W. Cook, and Douglas E. Miller. 1975. Random Variables and Correlational Overkill. *Educational and Psychological Measurement* 35, 3 (1975), 529–534. DOI:http://dx.doi.org/10.1177/001316447503500301

[234] NYU OSIRIS Lab. 2003. Capture the Flag — CSAW. (2003). https://www.csaw.io/ctf (Accessed 05-27-2020).

[235] NYU OSIRIS Lab. 2020. CSAW365. (2020). https://365.csaw.io/ (Accessed 05-27-2020).

[236] Sjoerd Langkemper. 2018. Practice your hacking skills with these CTFs. (December 2018). https://www.sjoerdlangkemper.nl/2018/12/19/practice-hacking-with-vulnerable-systems/ (Accessed 05-22-2020).

[237] Thomas D. LaToza, David Garlan, James D. Herbsleb, and Brad A. Myers. 2007. Program Comprehension As Fact Finding. In *Proc. of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 361–370. DOI:http://dx.doi.org/10.1145/1287624.1287675

[238] Thomas D. LaToza and Brad A. Myers. 2010. Developers Ask Reachability Questions. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE '10)*. ACM, New York, NY, USA, 185–194. DOI:http://dx.doi.org/10.1145/1806799.1806829

[239] Lucas Layman, Laurie Williams, and Robert St. Amant. 2007. Toward Reducing Fault Fix Time: Understanding Developer Behavior for the Design of Automated Fault Detection Tools. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement (ESEM '07)*. IEEE Computer Society, Washington, DC, USA, 176–185. DOI:http://dx.doi.org/10.1109/ESEM.2007.82

[240] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. 2014. Why does cryptographic software fail?: a case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems*. ACM, 7.

[241] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS 2019)*. DOI:http://dx.doi.org/10.14722/ndss.2019.23386

[242] Ákos Lédeczi, Miklós Maróti, Hamid Zare, Bernard Yett, Nicole Hutchins, Brian Broll, Péter Völgyesi, Michael B. Smith, Timothy Darrah, Mary Metelko, Xenofon Koutsoukos, and Gautam Biswas. 2019. Teaching Cybersecurity with Networked Robots. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, New York, NY, USA, 885–891. DOI:http://dx.doi.org/10.1145/3287324.3287450

[243] M. M. Lehman. 1980. Programs, life cycles, and laws of software evolution. *Proc. of the IEEE* 68, 9 (Sept 1980), 1060–1076. DOI:http://dx.doi.org/10.1109/PROC.1980.11805

[244] Timothy C Lethbridge, Jorge Diaz-Herrera, Richard Jr J LeBlanc, and J Barrie Thompson. 2007. Improving software practice through education: Challenges and future trends. In *Future of Software Engineering*. IEEE Computer Society, 12–28.

[245] T. C. Lethbridge, J. Singer, and A. Forward. 2003. How software engineers use documentation: the state of the practice. *IEEE Software* 20, 6 (2003), 35–39.

[246] Stanley Letovsky. 1986. Cognitive Processes in Program Comprehension. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. Ablex Publishing Corp., Norwood, NJ, USA, 58–79. http://dl.acm.org/citation.cfm?id=21842.28886

[247] Colleen M. Lewis, Ken Yasuhara, and Ruth E. Anderson. 2011. Deciding to Major in Computer Science: A Grounded Theory of Students? Self-Assessment of Ability. In *Proceedings of the Seventh International Workshop on Computing Education Research (ICER ?11)*. Association for Computing Machinery, New York, NY, USA, 3?10. DOI:http://dx.doi.org/10.1145/2016911.2016915

[248] Paul Luo Li, Andrew J. Ko, and Jiamin Zhu. 2015. What Makes a Great Software Engineer?. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 700–710. http://dl.acm.org/citation.cfm?id=2818754.2818839

[249] Michael Ligh, Steven Adair, Blake Hartstein, and Matthew Richard. 2010. *Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code*. John Wiley & Sons.

[250] Richard Harold Lindeman. 1980. *Introduction to bivariate and multivariate analysis*. Technical Report.

[251] LinkedIn. 2019. LinkedIn. https://linkedin.com/. (2019).

[252] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. 1986. Mental Models and Software Maintenance. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. Ablex Publishing Corp., Norwood, NJ, USA, 80–98. http://dl.acm.org/citation.cfm?id=21842.28887

[253] Matthew Lombard, Jennifer Snyder-Duch, and Cheryl Campanella Bracken. 2002. Content analysis in mass communication: Assessment and reporting of intercoder reliability. *Human communication research* 28, 4 (2002), 587–604.

[254] J Scott Long. 1983. *Confirmatory factor analysis: A preface to LISREL*. Vol. 33. Sage Publications.

[255] Robert Loo. 1983. Caveat on Sample Sizes in Factor Analysis. *Perceptual and Motor Skills* 56, 2 (1983), 371–374. DOI:http://dx.doi.org/10.2466/pms.1983.56.2.371

[256] Robert C MacCallum, Michael W Browne, and Hazuki M Sugawara. 1996. Power analysis and determination of sample size for covariance structure modeling. *Psychological methods* 1, 2 (1996), 130.

[257] Kaie Maennel, Rain Ottis, and Olaf Maennel. 2017. Improving and Measuring Learning Effectiveness at Cyber Defense Exercises. In *Secure IT Systems*, Helger Lipmaa, Aikaterini Mitrokotsa, and Raimundas Matulevičius (Eds.). Springer International Publishing, Cham, 123–138.

[258] Thomas Maillart, Mingyi Zhao, Jens Grossklags, and John Chuang. 2016. Given Enough Eyeballs, All Bugs Are Shallow? Revisiting Eric Raymond with Bug Bounty Programs. In *Proceedings of the 15th Workshop on the Economics of Information Security (WEIS '16)*. http://weis2016.econinfosec.org/wp-content/uploads/sites/2/2015/08/WEIS_2016_paper_76.pdf

[259] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. 2011. Design Lessons from the Fastest Q&A Site in the West. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. Association for Computing Machinery, New York, NY, USA, 2857–2866. DOI:http://dx.doi.org/10.1145/1978942.1979366

[260] Jane Margolis and Allan Fisher. 2002. *Unlocking the clubhouse: Women in computing*. MIT press, Cambridge, MA.

[261] LA Maruscuilo and JR Levin. 1983. Multivariate statistics in the social sciences. *Books/Cole, Monterrey, California* (1983).

[262] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. 2019. Reliability and Inter-Rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice. *Proc. ACM Hum.-Comput. Interact.* 3, CSCW, Article 72 (Nov. 2019), 23 pages. DOI:http://dx.doi.org/10.1145/3359174

[263] Gary McGraw, Sammy Migues, and Brian Chess. 2009. Software Security Framework — BSIMM. (2009). https://www.bsimm.com/framework.html (Accessed 05-22-2018).

[264] Gary McGraw and John Steven. 2011. Software [In]security: Comparing Apples, Oranges, and Aardvarks (or, All Static Analysis Tools Are Not Created Equal. (2011). http://www.informit.com/articles/article.aspx?p=1680863 (Accessed 02-26-2017).

[265] Robert K McKinley, Terjinder Manku-Scott, Adrian M Hastings, David P French, and Richard Baker. 1997. Reliability and validity of a new measure of patient satisfaction with out of hours primary medical care in the united kingdom: development of a patient questionnaire. *BMJ* 314, 7075 (1997), 193. DOI:http://dx.doi.org/10.1136/bmj.314.7075.193

[266] Root Me. 2010. Root Me. (2010). https://www.root-me.org/ (Accessed 05-27-2020).

[267] Adam W Meade and S Bartholomew Craig. 2012. Identifying careless responses in survey data. *Psychological methods* 17, 3 (2012), 437.

[268] Meetup. 2017. We Are What We Do — Meetup. (2017). https://www.meetup.com (Accessed 06-08-2017).

[269] Adam Mein and Chris Evans. 2011. Dosh4Vulns: Google's Vulnerability Reward Programs. (2011). https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwil6IGama_SAhXL7YMKHVuODtsQFggcMAA&url=https%3A%2F%2Fsoftware-security.sans.org%2Fdownloads%2Fappsec-2011-files%2Fvrp-presentation.pdf&usg=AFQjCNE3KBH-YVZLREoQtDEFC4LNtrcrug&sig2=GkTVmNFxUNkSOCeTTZXnIQ&bvm=bv.148073327,d.eWE (Accessed 02-26-2017).

[270] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates. 2013. When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 65–74. DOI: http://dx.doi.org/10.1109/ESEM.2013.19

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

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

[273] Microsoft. 2019. Microsoft Security Development Lifecycle Practices. `https://www.microsoft.com/en-us/securityengineering/sdl/practices`. (2019).

[274] Thomas E Miller, Steve P Wolf, Marvin L Thordsen, and Gary Klein. 1992. A decision-centered approach to storyboarding anti-air warfare interfaces. *Fairborn, OH: Klein Associates Inc. Prepared under contract* 66001 (1992).

[275] Jelena Mirkovic and Peter A. H. Peterson. 2014. Class Capture-the-Flag Exercises. In *2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14)*. USENIX Association, San Diego, CA. `https://www.usenix.org/conference/3gse14/summit-program/presentation/mirkovic`

[276] Jelena Mirkovic, Aimee Tabor, Simon Woo, and Portia Pusey. 2015. Engaging Novices in Cybersecurity Competitions: A Vision and Lessons Learned at ACM Tapia 2015. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)*. USENIX Association, Washington, D.C. `https://www.usenix.org/conference/3gse15/summit-program/presentation/mirkovic`

[277] Mitre. 2019. CVE. (2019). `https://cve.mitre.org/`

[278] MITRE. 2019. CWE: Common Weakness Enumeration. `https://cwe.mitre.org/data/definitions/1000.html/`. (2019).

[279] Mozilla. 2016. Components for Firefox. (2016). `https://bugzilla.mozilla.org/describecomponents.cgi?product=Firefox` (Accessed 02-18-2017).

[280] Sarah Nadi, Stefan Kruger, Mira Mezini, and Eric Bodden. 2016. "Jumping Through Hoops": Why do Java Developers Struggle with Cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. 935–946. `DOI:http://dx.doi.org/10.1145/2884781.2884790`

[281] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, and Matthew Smith. 2020. 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 (CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–13. `DOI:http://dx.doi.org/10.1145/3313831.3376791`

[282] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, Emanuel von Zezschwitz, and Matthew Smith. 2019. "If you want, I can store the encrypted password." A Password-Storage Field Study with Freelance Developers. (2019).

[283] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. 2017. Why Do Developers Get Password Storage Wrong?: A Qualitative Usability Study. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 311–328.

[284] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, and Matthew Smith. 2018. Deception Task Design in Developer Password Studies: Exploring a Student Sample. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*. USENIX Association, Baltimore, MD, 297–313. `https://www.usenix.org/conference/soups2018/presentation/naiakshina`

[285] Richard G Netemeyer, William O Bearden, and Subhash Sharma. 2003. *Scaling procedures: Issues and applications*. Sage Publications.

[286] Netgarage. 2020. IO Wargame. (2020). `https://io.netgarage.org/` (Accessed 05-27-2020).

[287] William Newhouse, Stephanie Keith, Benjamin Scribner, and Greg Witte. 2017. *NIST Special Publication 800-181, The NICE Cybersecurity Workforce Framework*. Technical Report. National Institute of Standards and Technology.

[288] Duc Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. 2017. A Stitch in Time: Supporting Android Developers in WritingSecure Code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 1065–1077. `DOI:http://dx.doi.org/10.1145/3133956.3133977`

[289] Linda B Nilson. 2000. The graphic syllabus: A demonstration workshop on how to visually represent a course. In *Proceedings of the Professional and Organizational Development Network in Higher Education (POD '00)*.

[290] Linda B. Nilson. 2002. The Graphic Syllabus: Shedding a Visual Light on Course Organization. *To Improve the Academy* 20, 1 (2002), 238–259. DOI:http://dx.doi.org/10.1002/j.2334-4822.2002.tb00585.x

[291] Timothy Nosco, Jared Ziegler, Zechariah Clark, Davy Marrero, Todd Finkler, Andrew Barbarello, and W. Michael Petullo. 2020. The Industrial Age of Hacking. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1129–1146. https://www.usenix.org/conference/usenixsecurity20/presentation/nosco

[292] J.D. Novak, D.B. Gowin, Cambridge University Press, and J.B. Kahle. 1984. *Learning How to Learn*. Cambridge University Press. https://books.google.com/books?id=8jkBcSDQPXcC

[293] Jum C Nunnally. 1994. *Psychometric theory* (3rd ed.). Tata McGraw-Hill Education.

[294] Angela M. O'Donnell and King Alison. 1999. *Cognitive Perspectives on Peer Learning* (1st ed.). Routledge. https://doi.org/10.4324/9781410603715

[295] Trail of Bits. 2017. Find a CTF - CTF Field Guide. (2017). https://trailofbits.github.io/ctf/intro/find.html (Accessed 06-08-2017).

[296] Plaid Parliament of Pwning. 2013. picoCTF. (2013). https://picoctf.com/ (Accessed 05-27-2020).

[297] National Institute of Standards and Technology. 2019. Software and Supply Chain Assurance Forum - Cyber Supply Chain Risk Management — CSRC. https://csrc.nist.gov/Projects/Supply-Chain-Risk-Management/SSCA. (2019).

[298] Offensive Security. 2017. Offensive Security's Exploit Database Archive. (2017). https://www.exploit-db.com/ (Accessed 08-1-2017).

[299] H. Ohno. 2000. Analysis and modeling of human driving behaviors using adaptive cruise control. In *IECON '00*, Vol. 4. 2803–2808 vol.4. DOI:http://dx.doi.org/10.1109/IECON.2000.972442

[300] K. Ohtsuka. 1997. "Scheduling tracing", a technique of knowledge elicitation for production scheduling. In *ICSMCCCS '97*, Vol. 2. 1033–1038 vol.2. DOI:http://dx.doi.org/10.1109/ICSMC.1997.638084

[301] P. OKane, S. Sezer, and K. McLaughlin. 2011. Obfuscation: The Hidden Malware. *IEEE Security and Privacy* 9, 5 (Sep. 2011), 41–47. DOI:http://dx.doi.org/10.1109/MSP.2011.98

[302] Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A. DeLong, Justin Cappos, and Yuriy Brun. 2018. API Blindspots: Why Experienced Developers Write Vulnerable Code. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*. USENIX Association, Baltimore, MD, 315–328. https://www.usenix.org/conference/soups2018/presentation/oliveira

[303] Kenneth Olmstead and Aaron Smith. 2017. Americans and Cybersecurity. (2017). http://www.pewinternet.org/2017/01/26/americans-and-cybersecurity/ (Accessed 07-15-2017).

[304] OverTheWire. 2020. OverTheWire. (2020). https://overthewire.org/wargames/ (Accessed 05-27-2020).

[305] OWASP. 2017. Top 10-2017 Top 10. https://www.owasp.org/index.php/Top_10-2017_Top_10. (2017).

[306] Kentrell Owens, Alexander Fulton, Luke Jones, and Martin Carlisle. 2019. pico-Boo!: How to avoid scaring students away in a CTF competition. (2019). https://cisse.info/pdf/download.php?file=CISSE_v07_i01_p21_pre.pdf

[307] Andy Ozment. 2004. Bug Auctions: Vulnerability Markets Reconsidered. In *Third Workshop on the Economics of Information Security*.

[308] PACTF. 2016. PACTF. (2016). https://2019.pactf.com/ (Accessed 05-27-2020).

[309] Annemarie Sullivan Palincsar and Ann L Brown. 1983. Reciprocal teaching of comprehension-monitoring activities. *Center for the Study of Reading Technical Report; no. 269* (1983).

[310] Hernan Palombo, Armin Ziaie Tabari, Daniel Lende, Jay Ligatti, and Xinming Ou. 2020. 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)*. USENIX Association, 205–220. https://www.usenix.org/conference/soups2020/presentation/palombo

[311] G. Parekh, D. DeLatte, G. L. Herman, L. Oliva, D. Phatak, T. Scheponik, and A. T. Sherman. 2018. Identifying Core Concepts of Cybersecurity: Results of Two Delphi Processes. *IEEE Transactions on Education* 61, 1 (Feb 2018), 11–20. DOI:http://dx.doi.org/10.1109/TE.2017.2715174

[312] D. L. Parnas and P. C. Clements. 1986. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering* SE-12, 2 (1986), 251–257.

[313] Ernest T Pascarella and Patrick T Terenzini. 1991. *How college affects students: Findings and insights from twenty years of research.* ERIC.

[314] Victor-Valeriu Patriciu and Adrian Constantin Furtuna. 2009. Guide for Designing Cyber Security Exercises. In *Proceedings of the 8th WSEAS International Conference on E-Activities and Information Security and Privacy (E-ACTIVITIES/ISP 09)*. World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 172–177.

[315] Eyal Peer, Laura Brandimarte, Sonam Samat, and Alessandro Acquisti. 2017. Beyond the Turk: Alternative platforms for crowdsourcing behavioral research. *Journal of Experimental Social Psychology* 70 (2017), 153 – 163. DOI:http://dx.doi.org/https://doi.org/10.1016/j.jesp.2017.01.006

[316] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19, 3 (1987), 295 – 341. DOI:http://dx.doi.org/https://doi.org/10.1016/0010-0285(87)90007-7

[317] Adam Perer and Ben Shneiderman. 2008. Systematic Yet Flexible Discovery: Guiding Domain Experts Through Exploratory Data Analysis. In *IUI '08*. ACM, New York, NY, USA, 109–118. DOI:http://dx.doi.org/10.1145/1378773.1378788

[318] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 426–437. DOI:http://dx.doi.org/10.1145/2810103.2813604

[319] Nicole Perlroth. 2017. All 3 Billion Yahoo Accounts Were Affected by 2013 Attack. (2017). https://www.nytimes.com/2017/10/03/technology/yahoo-hack-3-billion-users.html

[320] Jean Piaget. 1978. *Success and understanding.* Routledge.

[321] Jean Piaget and Margaret Cook. 1952. *The origins of intelligence in children.* Vol. 8. International Universities Press New York.

[322] Olgierd Pieczul, Simon Foley, and Mary Ellen Zurko. 2017. Developer-centered Security and the Symmetry of Ignorance. In *Proceedings of the 2017 New Security Paradigms Workshop (NSPW 2017)*. ACM, New York, NY, USA, 46–56. DOI:http://dx.doi.org/10.1145/3171533.3171539

[323] William A. Pike, John Stasko, Remco Chang, and Theresa A. O'Connell. 2009. The Science of Interaction. *Information Visualization* 8, 4 (2009), 263–274.

[324] Andreas Poller, Laura Kocksch, Sven Türpe, Felix Anand Epp, and Katharina Kinder-Kurlanda. 2017. Can Security Become a Routine? A Study of Organizational Change in an Agile Software Development Group. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW '17)*. Association for Computing Machinery, New York, NY, USA, 2489?2503. DOI:http://dx.doi.org/10.1145/2998181.2998191

[325] Portia Pusey, Sr. David Tobey, and Ralph Soule. 2014. An Argument for Game Balance: Improving Student Engagement by Matching Difficulty Level with Learner Readiness. In *2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14)*. USENIX Association, San Diego, CA. https://www.usenix.org/conference/3gse14/summit-program/presentation/pusey

[326] P. Pusey, M. Gondree, and Z. Peterson. 2016. The Outcomes of Cybersecurity Competitions and Implications for Underrepresented Populations. *IEEE Security Privacy* 14, 6 (2016), 90–95.

[327] K. Qian, D. Lo, H. Shahriar, L. Li, F. Wu, and P. Bhattacharya. 2017. Learning database security with hands-on mobile labs. In *2017 IEEE Frontiers in Education Conference (FIE)*. 1–6. DOI:http://dx.doi.org/10.1109/FIE.2017.8190716

[328] Radare. 2019. Radare. (2019). https://rada.re/n/radare2.html (Accessed 11-11-2019).

[329] Adrian E Raftery. 1995. Bayesian model selection in social research. *Sociological methodology* (1995), 111–163.

[330] Gilles Raîche, Theodore A. Walls, David Magis, Martin Riopel, and Jean-Guy Blais. 2013. Non-Graphical Solutions for Cattell's Scree Test. *Methodology* 9, 1 (2013), 23–29. DOI:http://dx.doi.org/10.1027/1614-2241/a000051

[331] Sam Ransbotham, Sabyasachi Mitra, and Jon Ramsey. 2012. Are Markets for Vulnerabilities Effective. *MIS Quarterly* 36, 1 (2012), 43–64.

[332] A. Rashid, G. Danezis, H. Chivers, E. Lupu, A. Martin, M. Lewis, and C. Peersman. 2018. Scoping the Cyber Security Body of Knowledge. *IEEE Security Privacy* 16, 3 (May 2018), 96–102. DOI:http://dx.doi.org/10.1109/MSP.2018.2701150

[333] J. Rasmussen. 1983. Skills, rules, and knowledge; signals, signs, and symbols, and other distinctions in human performance models. *ICSMCCCS '83* SMC-13, 3 (May 1983), 257–266. DOI:http://dx.doi.org/10.1109/TSMC.1983.6313160

[334] Tenko Raykov. 1997. Scale Reliability, Cronbach's Coefficient Alpha, and Violations of Essential Tau-Equivalence with Fixed Congeneric Components. *Multivariate Behavioral Research* 32, 4 (1997), 329–353. DOI:http://dx.doi.org/10.1207/s15327906mbr3204_2 PMID: 26777071.

[335] Bradley Reaves, Nolen Scaife, Adam M Bates, Patrick Traynor, and Kevin RB Butler. 2015. Mo (bile) Money, Mo (bile) Problems: Analysis of Branchless Banking Applications in the Developing World.. In *USENIX Security Symposium*. 17–32.

[336] Theodore Reed, Kevin Nauer, and Austin Silva. 2013. Instrumenting Competition-Based Exercises to Evaluate Cyber Defender Situation Awareness. In *Foundations of Augmented Cognition*, Dylan D. Schmorrow and Cali M. Fidopiastis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–89.

[337] Tony Rice, Josh Brown-White, Tania Skinner, Nick Ozmore, Nazira Carlage, Wendy Poland, Eric Heitzman, and Danny Dhillon. 2018. *Fundamental Practices for Secure Software Development*. Technical Report. Software Assurance Forum for Excellence in Code.

[338] M. P. Robillard. 2009. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software* 26, 6 (2009), 27–34.

[339] M. P. Robillard, W. Coelho, and G. C. Murphy. 2004. How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering* 30, 12 (Dec 2004), 889–903. DOI:http://dx.doi.org/10.1109/TSE.2004.101

[340] Martin P. Robillard and Robert J. Walker. 2014. *An Introduction to Recommendation Systems in Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–11. DOI:http://dx.doi.org/10.1007/978-3-642-45135-5_1

[341] John P Robinson, Phillip R Shaver, and Lawrence S Wrightsman. 1991. Criteria for scale selection and evaluation. *Measures of personality and social psychological attitudes* 1, 3 (1991), 1–16.

[342] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012a. How Do Professional Developers Comprehend Software?. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 255–265. http://dl.acm.org/citation.cfm?id=2337223.2337254

[343] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012b. How Do Professional Developers Comprehend Software?. In *ICSE '12*. IEEE Press, Piscataway, NJ, USA, 255–265. http://dl.acm.org/citation.cfm?id=2337223.2337254

[344] Karol G Ross, Gary A Klein, Peter Thunholm, John F Schmitt, and Holly C Baxter. 2004. *The recognition-primed decision model*. Technical Report. Army Combined Arms Center Military Review. 6–10 pages.

[345] Ernst Z Rothkopf and MJ Billington. 1979. Goal-guided learning from text: inferring a descriptive processing model from inspection times and eye movements. *Journal of educational psychology* 71, 3 (1979), 310.

[346] Dale C. Rowe, Barry M. Lunt, and Joseph J. Ekstrom. 2011. The Role of Cyber-security in Information Technology Education. In *Proceedings of the 2011 Conference on Information Technology Education (SIGITE '11)*. ACM, New York, NY, USA, 113–122. DOI:http://dx.doi.org/10.1145/2047594.2047628

[347] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L. Mazurek, and Piotr Mardziel. 2016. Build It, Break It, Fix It: Contesting Secure Development. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 690–703. DOI:http://dx.doi.org/10.1145/2976749.2978382

[348] Andrew Ruef, Evan Jensen, Nick Anderson, Alex Sotirov, Jay Little, Brandon Edwards, Marcin W, Dino Dai Zovi, and Mike Myers. CTF Field Guide. (????). https://trailofbits.github.io/ctf/ (Accessed 05-21-2020).

[349] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. 2004. A Comparison of Bug Finding Tools for Java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE '04)*. IEEE Computer Society, Washington, DC, USA, 245–256. DOI:http://dx.doi.org/10.1109/ISSRE.2004.1

[350] s4r. 2020. crackmes. (2020). https://crackmes.one/ (Accessed 05-27-2020).

[351] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*, Vol. 1. IEEE Computer Society, 598–608.

[352] J. H. Saltzer and M. D. Schroeder. 1975. The Protection of Information in Computer Systems. In *Symposium on Operating System Principles*. 1278–1308. DOI:http://dx.doi.org/10.1109/PROC.1975.9939

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

[354] Marlene Scardamalia, Carl Bereiter, and Rosanne Steinbach. 1984. Teachability of Reflective Processes in Written Composition. *Cognitive Science* 8, 2 (1984), 173 – 190. DOI:http://dx.doi.org/https://doi.org/10.1016/S0364-0213(84)80016-6

[355] Alan H Schoenfeld. 1983. Problem Solving in the Mathematics Curriculum. A Report, Recommendations, and an Annotated Bibliography. MAA Notes, Number 1. (1983).

[356] Ralf Schwarzer, Matthias Jerusalem, J Weinman, S Wright, and M Johnston. 1995. Measures in health psychology: A user?s portfolio. Causal and control beliefs. *Generalized Self-Efficacy Scal, NFER-NELSON, Windsor* (1995), 35–37.

[357] SDSLabs. 2020. backdoor. (2020). https://backdoor.sdslabs.co/ (Accessed 05-27-2020).

[358] K Serebryany. 2015. libFuzzer. https://llvm.org/docs/LibFuzzer.html. (2015).

[359] Elaine Seymour and Nancy M. Hewitt. 2000. *Talking About Leaving: Why Undergraduates Leave the Sciences*. Westview Press.

[360] Alan T. Sherman, David DeLatte, Michael Neary, Linda Oliva, Dhananjay Phatak, Travis Scheponik, Geoffrey L. Herman, and Julia Thompson. 2018. Cybersecurity: Exploring core concepts through six scenarios. *Cryptologia* 42, 4 (2018), 337–377. DOI:http://dx.doi.org/10.1080/01611194.2017.1362063

[361] Swapneel Sheth, Gail Kaiser, and Walid Maalej. 2014. Us and Them: A Study of Privacy Requirements Across North America, Asia, and Europe. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 859–870. DOI:http://dx.doi.org/10.1145/2568225.2568244

[362] Ben Shneiderman. 1996. The eyes have it: a task by data type taxonomy for information visualizations. In *IEEE Symposium on Visual Languages*. 336–343. DOI:http://dx.doi.org/10.1109/VL.1996.545307

[363] Ben Shneiderman and Catherine Plaisant. 2016. *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (4th ed.). Pearson.

[364] Yan Shoshitaishvili. 2020. zardus/wargame-nexus: A sorted and updated list of security wargame sites. (April 2020). https://github.com/zardus/wargame-nexus (Accessed 05-22-2020).

[365] Yan Shoshitaishvili and Connor) Nelson. 2020. pwn.college. (2020). https://pwn.college/ (Accessed 05-27-2020).

[366] Yan Shoshitaishvili, Ruoyu Wang, Audrey Dutcher, Lukas Dresel, Eric Gustafson, Nilo Redini, Paul Grosen, Colin Unger, Chris Salls, Nick Stephens, Christophe Hauser, John Grosen, Christopher Kruegel, and Giovanni Vigna. 2019. angr. (2019). http://angr.io (Accessed 08-21-2019).

[367] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. 138–157.

[368] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. 2017. Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance. In *Proc. of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM.

[369] Tarique Siddiqui, Albert Kim, John Lee, Karrie Karahalios, and Aditya Parameswaran. 2016. Effortless Data Exploration with Zenvisage: An Expressive and Interactive Visual Analytics System. *VLDB Endowment* 10, 4 (Nov. 2016), 457–468. DOI:http://dx.doi.org/10.14778/3025111.3025126

[370] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes? *SIGSOFT Softw. Eng. Notes* 30, 4 (May 2005), 1–5. DOI:http://dx.doi.org/10.1145/1082983.1083147

[371] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 17–30. DOI:http://dx.doi.org/10.1145/1926385.1926390

[372] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. 2015. Questions Developers Ask While Diagnosing Potential Security Vulnerabilities with Static Analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 248–259. DOI:http://dx.doi.org/10.1145/2786805.2786812

[373] George W Snedecor and Witiiam G Cochran. 1989. Statistical methods. *Ames: Iowa State Univ. Press Iowa* (1989).

[374] Mukesh Soni. 2006. Defect prevention: reducing costs and enhancing quality. *IBM:iSixSigma.com* 19 (2006).

[375] GaTech SSLab. 2014. pwnable.kr. (2014). http://pwnable.kr/ (Accessed 05-27-2020).

[376] StackOverflow. StackOverflow - Where Developers Learn, Share, and Build Careers. (????). https://stackoverflow.com/ (Accessed 06-05-2021).

[377] StackOverflow. 2018. StackOverflow Developer Survey Results 2018. https://insights.stackoverflow.com/survey/2018. (2018).

[378] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS '16*. Internet Society, 1–16.

[379] Anselm Strauss and Juliet Corbin. 1990. *Basics of qualitative research*. Vol. 15. Newbury Park, CA: Sage.

[380] Timothy C Summers, Kalle J Lyytinen, Tony Lingham, and Eugene Pierce. 2013. How Hackers Think: A Study of Cybersecurity Experts and Their Mental Models. In *Proc. of the 3rd International Conference on Engaged Management Scholarship (EMS '13)*. EDBAC.

[381] James Surowiecki. 2005. *The wisdom of crowds*. Anchor.

[382] Larry Suto. 2007. *Analyzing the Effectiveness and Coverage of Web Application Security Scanners*. Technical Report. BeyondTrust, Inc. https://www.beyondtrust.com/resources/white-paper/analyzing-the-effectiveness-and-coverage-of-web-application-security-scanners/

[383] Larry Suto. 2010. *Analyzing the Accuracy and Time Costs of Web Application Security Scanners*. Technical Report. BeyondTrust, Inc. https://www.beyondtrust.com/wp-content/uploads/Analyzing-the-Accuracy-and-Time-Costs-of-Web-Application-Security-Scanners.pdf

[384] John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive Science* 12, 2 (1988), 257 – 285. DOI:http://dx.doi.org/https://doi.org/10.1016/0364-0213(88)90023-7

[385] Syncack. 2017. Synack - Crowdsourced Security. (2017). https://www.synack.com (Accessed 06-08-2017).

[386] Synopsys. 2019. Coverity Scan - Static Analysis. (2019). https://scan.coverity.com/ (Accessed 05-30-2019).

[387] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: A DSL for the Definition of Incremental Program Analyses. In *ASE '16*. ACM, New York, NY, USA, 320–331. DOI:http://dx.doi.org/10.1145/2970276.2970298

[388] Barbara G Tabachnick, Linda S Fidell, and Jodie B Ullman. 2007. *Using multivariate statistics*. Vol. 5. Pearson Boston, MA.

[389] Madiha Tabassum, Stacey Watson, and Heather Richter Lipford. 2017. Comparing Educational Approaches to Secure Programming : Tool vs. TA. In *Proceedings of the 3rd Workshop on Security Information Workers (SOUPS '17)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/soups2017/workshop-program/wsiw2017/tabassum

[390] Gregory Tassey. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project* 7007, 011 (2002).

[391] Taylor Telford and Craig Timberg. 2018. Marriott discloses massive data breach affecting up to 500 million guests. (2018). https://www.washingtonpost.com/business/2018/11/30/marriott-discloses-massive-data-breach-impacting-million-guests/?utm_term=.ffddfc1dd7e6

[392] Smash the Stack Wargaming Network. 2020. Smash the Stack. (2020). http://smashthestack.org/ (Accessed 05-27-2020).

[393] Tyler W. Thomas, Heather Lipford, Bill Chu, Justin Smith, and Emerson Murphy-Hill. 2016. What Questions Remain? An Examination of How Developers Understand an Interactive Static Analysis Tool. In *Proceedings of the 2nd Workshop on Security Information Workers (WSIW '16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/soups2016/workshop-program/wsiw16/presentation/thomas

[394] Tyler W. Thomas, Madiha Tabassum, Bill Chu, and Heather Lipford. 2018. Security During Application Development: An Application Security Expert Perspective. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 262, 12 pages. DOI:http://dx.doi.org/10.1145/3173574.3173836

[395] David H. Tobey, Portia Pusey, and Diana L. Burley. 2014a. Engaging Learners in Cybersecurity Careers: Lessons from the Launch of the National Cyber League. *ACM Inroads* 5, 1 (March 2014), 53?56. DOI:http://dx.doi.org/10.1145/2568195.2568213

[396] David H. Tobey, Portia Pusey, and Diana L. Burley. 2014b. Engaging Learners in Cybersecurity Careers: Lessons from the Launch of the National Cyber League. *ACM Inroads* 5, 1 (March 2014), 53–56. DOI:http://dx.doi.org/10.1145/2568195.2568213

[397] Roger Tourangeau and Ting Yan. 2007. Sensitive questions in surveys. *Psychological bulletin* 133, 5 (2007), 859.

[398] Meng-Jung Tsai, Ching-Yeh Wang, and Po-Fen Hsu. 2019. Developing the Computer Programming Self-Efficacy Scale for Computer Literacy Education. *Journal of Educational Computing Research* 56, 8 (2019), 1345–1360. DOI:http://dx.doi.org/10.1177/0735633117746747

[399] Li tze Hu and Peter M. Bentler. 1999. Cutoff criteria for fit indexes in covariance structure analysis: Conventional criteria versus new alternatives. *Structural Equation Modeling: A Multidisciplinary Journal* 6, 1 (1999), 1–55. DOI:http://dx.doi.org/10.1080/10705519909540118

[400] Upworki. 2017. Hire Freelancers, Make Things Happen — Upwork. https://upwork.com. (2017).

[401] U.S. Department of Commerce. 2017. National Vulnerability Database. (2017). https://nvd.nist.gov/ (Accessed 08-01-2017).

[402] Jeroen JG Van Merrienboer and John Sweller. 2005. Cognitive load theory and complex learning: Recent developments and future directions. *Educational psychology review* 17, 2 (2005), 147–177.

[403] Manasi Vartak, Sajjadur Rahman, Samuel Madden, Aditya Parameswaran, and Neoklis Polyzotis. 2015. SeeDB: Efficient Data-driven Visualization Recommendations to Support Visual Analytics. *VLDB Endowment* 8, 13 (Sept. 2015), 2182–2193. DOI:http://dx.doi.org/10.14778/2831360.2831371

[404] Vector35. 2019a. Binary.Ninja: A Reverse Engineering Platform. (2019). https://binary.ninja/ (Accessed 05-30-2019).

[405] Vector35. 2019b. Vector35/Community-Plugins. (2019). https://github.com/Vector35/community-plugins/tree/master/plugins (Accessed 05-30-2019).

[406] Wayne F. Velicer, Andrew C. Peacock, and Douglas N. Jackson. 1982. A Comparison Of Component And Factor Patterns: A Monte Carlo Approach. *Multivariate Behavioral Research* 17, 3 (1982), 371–388. DOI:http://dx.doi.org/10.1207/s15327906mbr1703_5 PMID: 26800757.

[407] Markos Viggiato, Ricardo Terra, Henrique Rocha, Marco Tulio Valente, and Eduardo Figueiredo. 2018. Microservices in Practice: A Survey Study. (2018).

[408] Anneliese Von Mayrhauser and Steve Lang. 1998. Program comprehension and enhancement of software. In *In Proceedings IFIP World Computing Congress-Information Technology and Knowledge Engineering*.

[409] A. Von Mayrhauser and A.M. Vans. 1995. Industrial Experience with an Integrated Code Comprehension Model. *Software Engineering Journal* 10, 5 (1995), 171–182.

[410] Daniel Votipka, Desiree Abrokwa, and Michelle L. Mazurek. 2020a. Building and Validating a Scale for Secure Software Development Self-Efficacy. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–20. DOI:http://dx.doi.org/10.1145/3313831.3376754

[411] Daniel Votipka, Kelsey R. Fulton, James Parker, Matthew Hou, Michelle L. Mazurek, and Michael Hicks. 2020b. Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 109–126. https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-understanding

[412] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S. Foster, and Michelle L. Mazurek. 2020c. An Observational Investigation of Reverse Engineers' Processes. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1875–1892. https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-observational

[413] Daniel Votipka, Rock Stevens, Elissa M Redmiles, Jeremy Hu, and Michelle L Mazurek. 2018. Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes. *Proc. of the IEEE* (2018).

[414] L.S. Vygotsky. 1980. *Mind in Society: The Development of Higher Psychological Processes*. Harvard University Press. https://books.google.com/books?id=Irq913lEZ1QC

[415] L.S. Vygotsky, E. Hanfmann, G. Vakar, and A. Kozulin. 2012. *Thought and Language*. MIT Press. https://books.google.com/books?id=B9HClB0P6d4C

[416] Jan Vykopal and Miloš Barták. 2016. On the Design of Security Games: From Frustrating to Engaging Learning. In *2016 USENIX Workshop on Advances in Security Education (ASE 16)*. USENIX Association, Austin, TX. https://www.usenix.org/conference/ase16/workshop-program/presentation/vykopal

[417] T. Wang, T. Wei, G. Gu, and W. Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *S&P '10*. 497–512.

[418] L.W. Watson. 2002. *How Minority Students Experience College: Implications for Planning and Policy*. Stylus. https://books.google.com/books?id=g6_7R-IX_E0C

[419] Stacey Watson and Heather Richter Lipford. 2017. A Proposed Visualization for Vulnerability Scan Data. In *Proceedings of the 3rd Workshop on Security Information Workers (WSIW '17)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/soups2017/workshop-program/wsiw2017/watson

[420] C. Weir, L. Blair, I. Becker, A. Sasse, and J. Noble. 2018. Light-Touch Interventions to Improve Software Development Security. In *2018 IEEE Cybersecurity Development (SecDev)*. 85–93. DOI: http://dx.doi.org/10.1109/SecDev.2018.00019

[421] Arnold D Well and Jerome L Myers. 2003. *Research Design & Statistical Analysis* (2nd ed.). Psychology Press.

[422] Susan C Weller and A Kimball Romney. 1988. *Systematic data collection*. Vol. 10. Sage publications.

[423] Joseph Werther, Michael Zhivich, Tim Leek, and Nickolai Zeldovich. 2011. Experiences in Cyber Security Education: The MIT Lincoln Laboratory Capture-the-flag Exercise. In *Proc. of the 4th Conference on Cyber Security Experimentation and Test (CSET'11)*. USENIX Association, Berkeley, CA, USA, 12–12. http://dl.acm.org/citation.cfm?id=2027999.2028011

[424] Michael Whitney, Heather Lipford-Richter, Bill Chu, and Jun Zhu. 2015. Embedding Secure Coding Instruction into the IDE: A Field Study in an Advanced CS Course. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. ACM, New York, NY, USA, 60–65. DOI:http://dx.doi.org/10.1145/2676723.2677280

[425] Elizabeth J Whitt, Marcia I Edison, Ernest T Pascarella, Amaury Nora, and Patrick T Terenzini. 1999. Women's perceptions of a" chilly climate" and cognitive outcomes in college: Additional evidence. *Journal of College Student Development* (1999).

[426] Jordan Wiens. 2019. Practice CTF List. (May 2019). `https://captf.com/practice-ctf/` (Accessed 05-21-2020).

[427] Chamila Wijayarathna and Nalin A. G. Arachchilage. 2018. Why Johnny Can't Store Passwords Securely?: A Usability Evaluation of Bouncycastle Password Hashing. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering (EASE'18)*. ACM, New York, NY, USA, 205–210. `DOI:http://dx.doi.org/10.1145/3210459.3210483`

[428] Jim Witschey, Olga Zielinska, Allaire Welk, Emerson Murphy-Hill, Chris Mayhorn, and Thomas Zimmermann. 2015. Quantifying Developers' Adoption of Security Tools. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15)*. ACM, 260–271. `DOI:http://dx.doi.org/10.1145/2786805.2786816`

[429] Michelle Y Wong and David Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware.. In *NDSS '16*. Internet Society, 21–24.

[430] Irene M.Y. Woon and Atreyi Kankanhalli. 2007. Investigation of IS professionals' intention to practise secure development of applications. *International Journal of Human-Computer Studies* 65, 1 (2007), 29 – 41. `DOI:http://dx.doi.org/https://doi.org/10.1016/j.ijhcs.2006.08.003`

[431] Zhengchuan Xu, Qing Hu, and Chenghong Zhang. 2013. Why Computer Talents Become Computer Hackers. *Commun. ACM* 56, 4 (April 2013), 64–74. `DOI:http://dx.doi.org/10.1145/2436256.2436272`

[432] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. 2016. Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. *2016 IEEE Symposium on Security and Privacy (SP)* 00 (2016), 158–177. `DOI:http://dx.doi.org/doi.ieeecomputersociety.org/10.1109/SP.2016.18`

[433] T. Yamaguchi, H. Nitta, J. Miyamichi, and T. Takagi. 2000. Distributed sensory intelligence architecture for human centered ITS. In *IECON '00*, Vol. 1. 509–514 vol.1. `DOI:http://dx.doi.org/10.1109/IECON.2000.973202`

[434] J. S. Yi, Y. a. Kang, and J. Stasko. 2007. Toward a Deeper Understanding of the Role of Interaction in Information Visualization. *IEEE Transactions on Visualization and Computer Graphics* 13, 6 (Nov 2007), 1224–1231. `DOI:http://dx.doi.org/10.1109/TVCG.2007.70515`

[435] Michal Zalewski. 2014. American Fuzzing Lop (AFL). `http://lcamtuf.coredump.cx/afl/`. (2014).

[436] Andreas Zeller. 2009. *Why programs fail: a guide to systematic debugging*. Elsevier.

[437] Kim Zetter. 2016. Inside the Cunning, Unprecedented Hack of Ukraine's Power Grid. (2016). `https://www.wired.com/2016/03/inside-cunning-unprecedented-hack-ukraines-power-grid/`

[438] Mingyi Zhao, Jens Grossklags, and Peng Liu. 2015. An Empirical Study of Web Vulnerability Discovery Ecosystems. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 1105–1117. `DOI:http://dx.doi.org/10.1145/2810103.2813704`

[439] Mingyi Zhao, Aron Laszka, Thomas Maillart, and Jens Grossklags. 2016. Crowdsourced Security Vulnerability Discovery: Modeling and Organizing Bug-Bounty Programs. In *Proceedings of the 4th AAAI Workshop on Mathematical Foundations of Human Computation (HCOMP '16)*.

[440] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. 2012. SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *SPSM '12*. ACM, New York, NY, USA, 93–104.

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

[442] Caroline E Zsambok and Gary Klein. 2014. *Naturalistic decision making*. Psychology Press.