Rochester Institute of Technology

RIT Digital Institutional Repository

Theses

6-2021

The State of Practice for Security Unit Testing: Towards Data Driven Strategies to Shift Security into Developer's Automated Testing Workflows

Danielle Nicole Gonzalez
dng2551@rit.edu

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
Gonzalez, Danielle Nicole, "The State of Practice for Security Unit Testing: Towards Data Driven Strategies to Shift Security into Developer's Automated Testing Workflows" (2021). Thesis. Rochester Institute of Technology. Accessed from

This Dissertation is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

The State of Practice for Security Unit Testing: Towards Data Driven Strategies to Shift Security into Developer's Automated Testing Workflows

by

Danielle Nicole Gonzalez

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in Computing and Information Sciences

B. Thomas Golisano College of Computing and Information Sciences

Rochester Institute of Technology
Rochester, New York
June 2021

The State of Practice for Security Unit Testing: Towards Data Driven Strategies to Shift Security into Developer's Automated Testing Workflows

by

Danielle Nicole Gonzalez

Committee Approval:

We, the undersigned committee members, certify that we have advised and/or supervised the candidate on the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements of the degree of Doctor of Philosophy in Computing and Information Sciences.

Dr. Mehdi Mirakhorli Dissertation Advisor	Date
Dr. Laurie Williams Dissertation Committee Member	Date
Dr. Mohamed Wiem Mkaouer Dissertation Committee Member	Date
Dr. Christian Newman Dissertation Committee Member	Date
Dr. Darren Narayan Dissertation Defense Chairperson	Date

Certified by:

Dr. Pengcheng Shi Ph.D. Program Director, Computing and Information Sciences	Date

©2021 Danielle Nicole Gonzalez
All rights reserved.

The State of Practice for Security Unit Testing: Towards Data Driven Strategies to Shift Security into Developer's Automated Testing Workflows

by

Danielle Nicole Gonzalez

Submitted to the
B. Thomas Golisano College of Computing and Information Sciences Ph.D. Program in Computing and Information Sciences
in partial fulfillment of the requirements for the
Doctor of Philosophy Degree
at the Rochester Institute of Technology

Abstract

The pressing need to "shift security left" in the software development lifecycle has motivated efforts to adapt the iterative and continuous process models used in practice today. *Security unit testing* is praised by practitioners and recommended by expert groups, usually in the context of DevSecOps and achieving "continuous security". In addition to vulnerability testing and standards adherence, this technique can help developers verify that security controls are implemented correctly, *i.e. functional* security testing. Further, the means by which security unit testing can be integrated into developer workflows is unique from other standalone tools as it is an *adaptation* of practices and infrastructure developers are already familiar with. Yet, software engineering researchers have so far failed to include this technique in their empirical studies on secure development and little is known about the state of practice for security unit testing. This dissertation is motivated by the disconnect between promotion of security unit testing and the lack of empirical evidence on how it *is* and *can be* applied. The goal of this work was to address the disconnect towards identifying actionable strategies to promote wider adoption and mitigate observed challenges. Three mixed-method empirical studies were conducted wherein practitioner-authored unit test code, Q&A posts, and grey literature were analyzed through three lenses: *Practices* (what they do), *Perspectives and Guidelines* (what and how they think it should be done), and *Pain Points* (what challenges they face) to incorporate both technical and human factors of this phenomena. Accordingly, this work contributes novel and important insights into how developers write functional unit tests for at least nine security controls, including a taxonomy of 53 authentication unit test cases derived from real code and a detailed analysis of seven unique pain points that developers seek help with from peers on Q&A sites. Recommendations given herein for conducting and adopting security unit testing, including mitigating challenges and addressing gaps between available and needed support, are grounded in the guidelines and perspectives on the benefits, limitations, use cases, and integration strategies shared in grey literature authored by practitioners.

Acknowledgments

"I am the combined effort of everyone I've ever known."
Chuck Palahniuk, *Invisible Monsters*

It takes a village to raise a PhD student, and I would like to thank all those who have supported, motivated, comforted, inspired, and challenged me.

I thank Dr. Darren Narayan for first seeing my research potential and mentoring me as a REU student. This experience was my introduction to the research world and instilled a sense of confidence in my ability to succeed in unfamiliar territory. I credit my interest in pursuing research as a career to the time spent in my PhD advisor Dr. Mehdi Mirakhorli's lab as an undergraduate student. I thank him for encouraging and sponsoring my entry into the field of software engineering research and for helping me navigate the intimidating worlds of academia, paper writing, rejections, conferences, funding, presentations, international travel, and much more throughout my time in his lab. Special thanks are extended to Dr. Thomas Zimmermann for his mentorship and for the opportunities to work with him and the SAINTes group at Microsoft Research in person and virtually during the pandemic. These experiences broadened my perspectives, challenged me in new ways, and left me with a lasting desire to make impactful and actionable contributions with my research in support of those whose work I study, *i.e.* developers. I am also incredibly grateful for Microsoft Research's support in awarding me a PhD Fellowship, which empowered me to concentrate on pursuing my ideas to the depths described herein. To any students who may read this in the future, having a group of PhD friends to lean on and learn from is vital to maintaining your health and sanity. Thanks to Nuthan Munaiah, Joanna Cecilia da Silva Santos, Blagovesta Kostova, Jamie Davis, and all of the PhD students who have climbed this hill alongside me and I am so fortunate to now call my friends.

There are countless others I am lucky to have in my personal and professional life who have also contributed to my success — if you think this is about you, *it probably is.*

To my parents Christine and Manuel, who empowered and enabled me to pursue my dreams through their sacrifices and unwavering belief in my potential.

To Joseph, the light by which I persevere, and so much more.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Security testing is a class of techniques for verifying and validating software security, and is a crucial component of a secure software development effort. Without such testing, there is no guarantee that the software is safe from attack, or that security strategies from the system's design have been correctly realized. In practice the most commonly (often *only*) used technique is penetration testing, which requires a fully operational instance of a system [13,99,169]. The objective is to simulate a real attack; an authorized external entity, without knowledge of the system's design or implementation, attempts various attacks and try to gain as much access as possible [62]. Techniques like model-based testing can be conducted pre-construction to verify requirements, specifications, and designs. For security testing during construction (implementation), the most-recommended techniques are manual secure code reviews and automated static analysis. These are effective techniques for identifying violations of secure coding best practice *e.g.* buffer overflows and the detection and prevention of known vulnerabilities.

The industry's acceptance of and reactions to omnipresent security threats continually evolves alongside the landscape of software development itself. As a 'nonfunctional' quality attribute [24, 148], security has traditionally been tested late in the development lifecycle. In many cases, tests are executed manually on fully-functional system instances by security experts with no involvement in their design or construction [62]. The application landscape is also highly competitive, especially in web and mobile.

Over time development processes have evolved to keep up with technological innovations and emerging security threats by shifting from linear Waterfall [167] models to iterative and continuous delivery models, *i.e.* Agile [67] and DevOps [120]. These models help organization stay competitive with rapid release schedules and increased automation. For example, the DevOps model unites development and operations with continuous integration (CI), continuous delivery (CD) & infrastructure as a service (IaaS). For example, the DevOps model unites development and operations with continuous integration (CI), continuous delivery (CD) & infrastructure as a service (IaaS). Projects following these model leverage automation to balance speed with quality. For example, they use automated unit and integration tests to continuously evaluate low-level system functionality during implementation. This type of developer-driven testing is now a standard (or at least familiar) practice in most development workflows.

1.1 Motivation and Research Goal

Now more than ever projects must incorporate security into *every* stage of the development lifecycle, an admittedly non-trivial undertaking. The need to 'shift security left' has motivated efforts to adapt these process models by integrating automated security testing into developer workflows, including at the unit and integration levels [203]. However, the current state of practice, and any existing strategies for integrating this technique into developer workflows is unknown. Further, for developers this is a non-traditional application of infrastructure and practices they are already familiar with, and there is little insight into any potential barriers that would impede explicitly security-focused testing at this level.

Research Goal

The **goal** of this work was to understand the state of practice for security unit testing and identify data-driven, actionable strategies to mitigate observed challenges faced by developers.

1.2 Research Questions

The insights needed to achieve this goal were formalized into the following research questions:

RQ 1. How is security unit testing conducted in **practice**?

Unit testing is primarily performed by developers; typically QA/testers use black box techniques. Hence, much of this work will focus on the perceptions and behaviors of software developers who use unit testing in their regular workflows. Related studies indicate security is important to developers [14, 98], and they use a specific set of security-focused tools and techniques (*e.g.* threat modeling, secure coding) to build security into their software [131]. However, little is known about how developers utilize unit testing, a common technique for evaluating fine-grain system behaviors, to verify behaviors of security control specifically. To answer this question (Chapter 4), I designed studies to analyze topics discussed in Q&A posts related to security unit testing and create a taxonomy of unit test cases written for token authentication from open source.

RQ 2. What **pain points** do developers experience when unit testing security controls?

The study I designed based on analysis of Q&A posts (mentioned above) also included a qualitative analysis to curate an ontology of pain points unique to security unit testing (Chapter 5). In this way, the understanding of developers practices can be supplemented with associated pain points when testing security controls can be more appropriately generalized. Further, this knowledge can be applied to influence and motivate the creation of tools and resources to support developers as they integrate this practice into their workflows.

RQ 3. What are practitioner's **perspectives** on security unit testing, and to what extent are actionable **guidelines** available for using this technique?

Recommendations for integrating security unit testing into developer workflows must be influenced by both technical and *socio*-technical factors. In this regard, I designed an extensive qualitative review of practitioner-authored grey literature to investigate, *in their own words*, what they perceive as the motivations, limitations, and use cases of security unit testing as well as their recommended strategies for workflow integration. As part of this review, I also sought to identify gaps in instructional resources, *i.e.* recommended use cases and strategies for which existing guidelines provide no concrete, actionable support for applying them.

1.3 Contributions

I devised the research design for this work under supervision of my advisor, Dr. Mehdi Mirakhorli. These studies were conducted with support from two other graduate students, Paola Peralta Perez and Michael Rath. From these efforts we contribute to the body of knowledge for development phase security testing, filling and/or identifying gaps related to unit-level test practices. Such empirical data is a critical precondition to devising guidelines and strategies for adopting security unit testing, and further for Action research towards interventions to address identified barriers [180].

Publication Record

The following is a list of the 11 publications I have led or contributed to over the course of my studies. Publications 1 (under review) and 5 [82] are of direct relevance to the work herein, and publications 3 [166], 8 [81], 9 [83], and 10 [80] also make contributions towards understanding and supporting unit testing in practice outside (but including) a security context. Publications 2 [84] and 4 [85] were collaborations with industry researchers at Microsoft Research in which I used similar data-driven empirical approaches to extract technical and sociotechnical insights from repository artifacts. Publication 6 [79] contributes an automated classification technique for vulnerability reporting, again based on learning from existing artifacts and in publication 7 [78] this data is used to understand domain-specific architectural security weaknesses. Finally, I contributed to the work described in publication 10 [220] towards automated training set creation for traceability research.

1. **Danielle Gonzalez**, Paola Peralta Perez, and Mehdi Mirakhorli. *Barriers to Shift-Left Security: The Unique Pain Points of Writing Automated Tests Involving Security Controls*, [*Under Review*] 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2021), Bari, Italy, 2021.

2. **Danielle Gonzalez**, Thomas Zimmermann, Patrice Godefroid, and Matt Schaefer. *"Anomalicious: Automated Detection of **Anomal**ous and Potentially Mal**icious** Commits on GitHub"*, In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Madrid, ES, 2021 pp. 258-267.

3. Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, **Danielle Gonzalez**, and Mehdi Mirakhorli. *"DeepTC-Enhancer: Improving the Readability of Automatically Generated Tests"*, In 35th IEEE/ACM International Conference on Automated Software Engineering, September 21–25, 2020, Virtual Event. ACM, New York, NY, USA, 12 pages. 2020.

4. **Danielle Gonzalez**, Thomas Zimmermann, and Nachiappan Nagappan. *"The State of the ML-universe: 10 Years of Artificial Intelligence & Machine Learning Software Development on GitHub"*, In Proceedings of the 17th International Conference on Mining Software Repositories, pp. 431-442. 2020.

5. **Danielle Gonzalez**, Michael Rath, and Mehdi Mirakhorli. *"Did You Remember To Test Your Tokens?"*, In Proceedings of the 17th International Conference on Mining Software Repositories, pp. 232-242. 2020.

6. **Danielle Gonzalez**, Fawaz Alhenaki, and Mehdi Mirakhorli. *"Architectural security weaknesses in industrial control systems (ICS) an empirical study based on disclosed software vulnerabilities"*, In 2019 IEEE International Conference on Software Architecture (ICSA), pp. 31-40. IEEE, 2019.

7. **Danielle Gonzalez**, Holly Hastings, and Mehdi Mirakhorli. *"Automated Characterization of Software Vulnerabilities"*, In 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 135-139. IEEE, 2019.

8. **Danielle Gonzalez**, Suzanne Prentice, and Mehdi Mirakhorli. *"A fine-grained approach for automated conversion of JUnit assertions to English"*, In Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering, pp. 14-17. 2018.

9. **Danielle Gonzalez**, Joanna CS Santos, Andrew Popovich, Mehdi Mirakhorli, and Mei Nagappan. *"A large-scale study on the usage of testing patterns that address maintainability attributes: patterns for ease of modification, diagnoses, and comprehension"*, In 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 391-401. IEEE, 2017.

10. **Danielle Gonzalez**, Andrew Popovich, and Mehdi Mirakhorli. *"TestEX: A Search Tool for Finding and Retrieving Example Unit Tests from Open Source Projects"*, In 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 153-159. IEEE, 2016.

11. Zogaan, Waleed, Ibrahim Mujhid, Joanna CS Santos, **Danielle Gonzalez**, and Mehdi Mirakhorli. *"Automated training-set creation for software architecture traceability problem"*, Empirical Software Engineering 22, no. 3 (2017): 1028-1062.

1.4 Outline

The remainder of this dissertation is organized as follows. An extensive overview of the background concepts for this work and related academic studies is provided in Chapter 2. Chapter 3 describes the three studies I designed to analyze practitioner-authored test cases, Q&A posts, and grey literature. The state of practice for security unit testing is examined in Chapters 4-5; each chapter describes this phenomena through one of the three lenses represented by my research questions: *Practices* (Chapter 4), *Perspectives & Guidelines* (Chapter 6), and *Pain Points* (Chapter 5). These insights facilitate the data-driven discussion and recommendations I provide in Chapter 7, where in I summarize the state of practice holistically and make recommendations for mitigating the barriers and support gaps this research has revealed, towards shifting security into developer's testing workflows.

Chapter 2

Background and Related Work

2.1 The Software Development Lifecycle

Software is *engineered* when a “systematic, disciplined, quantified approach” is applied to the “development, operation, and maintenance of software” [101]. Such approaches are referred to as process models or methodologies in practice, and they help bring “order to chaos” [109] by guiding project teams through the six core phases of the Software Development Lifecycle (SDLC): communication (requirements), planning (analysis), modeling (design), construction (implementation), testing, and deployment (maintenance) [35, 150]. Each model defines a unique flow (*e.g.* incremental, iterative) through the SDLC and prescribes a set of practices to be completed in each phase [35, 143, 150, 187].

Technological advances continually expand the capabilities (*e.g.* internet, artificial intelligence), applications (*e.g.* critical systems, social media), and availability (*e.g.* mobile, IoT) of software systems. Many aspects of daily life are now reliant on software either implicitly (*e.g.* transportation) or explicitly (*e.g.* social media), and society places a great deal of trust on these systems to securely manage sensitive data and make safety critical decisions.

2.2 Shifting Security Left

The risks introduced by the rapid increases in software connectivity, extensibility, & complexity have sparked renewed interest in the secure construction of software, *i.e.* “*building security in* [to software]” [123, 204]. Yet, without SDLCs that support this notion [99], security is often considered only in post-development testing [13, 169]. This has prompted the introduction of several models and frameworks that guide organizations through integrating security activities into each development phase [57]. Two of the most common secure development frameworks are Microsoft's Secure Development Lifecycle (SDL) [99] and the Secure Application Maturity Model (SAMM) [55] from OWASP, the Open Web Application Security Project. Some SDLC models have also begun to adapt themselves, *e.g.* "Secure DevOps" / "DevSecOps" [152, 203].

2.2.1 Microsoft Secure Development Lifecycle (SDL)

In the Microsoft Secure Development Lifecycle, security is *everyone's* responsibility. This is reflected in its core concepts: education, continuous process improvement, and accountability [40]. The SDL itself is comprised of *practices* that map to the traditional five software development phases: requirements, design, implementation, verification, & release [99]. The first (security training) and last (response process) practices are mapped to pre- and post- SDL phases. Three practices are directly security *testing* activities: practice 2 is to perform static analysis during implementation, and practices 11 and 12 are to perform dynamic analysis and fuzz testing during verification. Six practices also influence testing in requirements (2 & 4), design (5 & 7), and post-SDL (13 & 15).

2.2.2 OWASP Secure Application Maturity Model (SAMM)

The OWASP SAMM places heavy emphasis on business and management concepts with a focus on using organizational introspection to develop a customized security strategy. The model is based on 15 security *practices* organized by 'business functions' that map to development phases. Each practice encompasses several *activities* that correspond to three maturity levels. Starting from level 0 (no activities applied) organizations progress through the model until they have applied each activity at a target level. This model does not distribute testing activities to each phase: 'requirements-driven testing' and 'security testing' are delegated to the post-implementation 'verification' function. The activities for these practices are less specific than the SDL. For example, level 1 of security testing is "Perform security testing (both manual and tool based) to discover security defects [55]". The 'baseline' for this activity is to use 'automated security testing tools', and for a 'deep understanding' it suggests manual testing of 'high risk components' [55].

2.2.3 DevSecOps: Adding Continuous Security to the DevOps Model

Secure DevOps, also called 'DevSecOps' or 'SecDevOps' is an evolution of the DevOps model [152, 203] that incorporates security through use of activities and techniques that suit the highly automated and collaborative nature of the model. This concept is not quite a new model nor a framework; it is more of an ideal which organizations realize in different ways. However, some activities are more common than others, such as automated security monitoring, code review, and testing [152]. Automated security testing techniques such as static analysis are highly recommended as they are easy to include in pre-existing continuous integration pipelines [203].

2.3 Security Practices

In this section, I describe the secure software development practices most relevant to this work: how practitioners design security into their systems, techniques used to test security throughout the development lifecycle, and prior work that has examined human factors associated with the use and integration of these practices.

2.3.1 Building Security In: Architectural Tactics and Control Mechanisms

Once the requirements for a system have been established, the design phase begins. Here, the architect defines the components, structures, and controls that will comprise the system and their respective relationships and behaviors [24]. The software industry has matured such that architects can leverage established *patterns* [68] to satisfy functional requirements and *tactics* [24] for quality requirements.

There are six (primary) security tactics that can be introduced to a system at the design level [24]:

- *Authenticate Users*: Confirmation of user identity
- *Authorize Users*: Confirmation of user privileges
- *Maintain Data Confidentiality*: Prevention of unauthorized access to sensitive data
- *Maintain Integrity*: Protection of data from malicious or unintended manipulation
- *Limit Exposure*: Keeping components separated, to prevent "all eggs in one basket" scenarios
- *Limit Access*: Protection via granting "minimum required access"

These tactics are intentionally vague, because they represent principals that are applicable to software in general; at each phase of the development lifecycle they become more concrete and tangible parts of the system. It is the role of the architect to determine which tactics relate to security requirements, and then determine the *controls* that should be used to realize them in the implementation. For example, the *Authenticate Users* tactic could be incorporated into a web application with controls such as *login forms* and *sessions*. This would be realized in the system by requiring users to register for an account and provide credentials through a login form when they want access. Once authenticated, the user's established identity could be maintained across multiple requests by creating and managing a session in the browser.

During implementation, security tactics from the design are realized as security control mechanisms. This is a complicated process , as their functionality often requires multiple interacting components across several application layers. To alleviate some of the complexity, security frameworks like Spring Security [9] are often used. Additional decisions are made at the implementation phase. For example, to implement authentication controls there are several mechanisms (*e.g.* password, PIN), types (*e.g.* token, session), and protocols (*e.g.* SAML [18], LDAP [151]) to choose from, and systems can be designed to handle more than one configuration.

2.3.2 Testing Security Throughout the SDLC

As a project moves through each lifecycle phase, it is crucial that the system is tested to ensure that no design decision or implementation error deviates from the requirements. This is especially critical for security requirements; the complexity of security control implementations elicits many opportunities for mistakes. This is not to be taken lightly either as even simple mistakes like *failing to validate input* can cause vulnerabilities [201].

Testing Objectives

Security testing is a specialized form of software testing with two objectives. Both relate to the security controls & mechanisms incorporated into the system's design and implementation to satisfy its security requirements. The first objective is to make sure these security controls *behave* correctly, *i.e.* as expected based on requirements [169]. This can also be referred to as as *security vulnerability testing* [196] or *risk-based security testing* [149]. The second objective is to ensure security controls have been *implemented* correctly, *i.e.* they do not themselves contain vulnerabilities [169]. Applying security test techniques to this goal is known as *security functional testing* [196] or *testing security mechanisms* [149].

Ultimately the two objectives are closely related, as vulnerabilities are introduced into a system via missing or incorrectly implemented security controls (mechanisms). However they are unique in the planning, actions, and mindset necessary to achieve them. For example, security vulnerability testing is uniquely conducted through the perspective of an attacker (*e.g.* penetration testing) and requires in-depth knowledge of common vulnerabilities, exploits, and attack scenarios whereas security functional testing (*e.g.* static analysis) relies more on knowledge of a system's design, implementation, and requirements.

Testing Techniques

Organizations tend to choose only one technique for their projects, but in reality there is no single 'silver bullet' technique that can effectively verify security at all the necessary scopes and project stages [169]. Thus, meeting these objectives requires applying several security testing techniques throughout of the development lifecycle. A basic understanding of the benefits and limitations of the most common techniques, and their place in the development lifecycle will suffice and is provided here. For a deeper exploration into the taxonomy of security testing techniques, several existing resources are recommended [62, 149, 196, 213].

Requirements and Design Phase Security Testing: In the earliest phases of a software project, **model-based security testing (MBST)** can be applied to establish, verify, and refine security requirements, specifications, and designs *before* development begins. As the name implies, the system under test is modeled from security requirements, sequence diagrams, known potential vulnerabilities, threat models, etc. and can be represented in several ways such as UML and finite state machines [196]. Manually selected algorithms automatically design and generate a suite of test cases for the given model(s), driven by *test generation criteria* defined by the modeler to focus test efforts on desired threats, requirements, or attacks [175].

Development Phase Security Testing: Once development has begun, several techniques can be used for functional and vulnerability testing. Teams can conduct manual **secure code reviews** or run **static analysis security testing (SAST)** tools to ensure continued adherence to security requirements and correct realization of security controls included in the system's design. In both techniques, code is analyzed but not executed, with the objective of detecting violations of secure coding practices. Any identified issues can be addressed before the code moves forward in the lifecycle, reducing cost and time to fix. Secure code reviews are usually conducted manually and require participants with extensive security knowledge and awareness of the threats and vulnerabilities most applicable to the system under test. SAST is essentially an automated code review that relies on pre-defined patterns, which it will attempt to match in the code. Again, security expertise is needed to establish correct and relevant patterns. Security *unit* testing is discussed in more detail (being the topic of this reseearch) in Section 2.3.3.

Post-Development Phase Security Testing Once a running instance of a system is available, verification efforts are devoted to security *vulnerability* testing. Several techniques can be applied at this point to evaluate the performance of the security controls under normal and attack conditions. **Penetration testing** allows elaborate simulations of real attacks. "Pen-testers" attempt several strategies to gain unauthorized access to a system; once inside, they try to collect valuable assets and make malicious changes. Other techniques like **fuzzing** and **dynamic taint analysis** use specially crafted data to evaluate how date flows through the system and its handling of malicious inputs. These techniques are good for catching vulnerabilities such as SQL injection and buffer overflows. Finally, **security regression testing** can be used to monitor how system changes affect security controls; these automated techniques can select and prioritize test cases that are relevant to the given changes, and optimize test suites.

2.3.3 Security Unit Testing

Security unit testing is a encouraged by respected security organizations [44, 169] and practitioners [25, 29, 48, 50, 203], but its use in practice is not well understood and these recommendations are not appropriately supported with actionable guidelines [82, 171]. Although security unit tests can be used for functional [25, 29, 44, 203, 215] *and* vulnerability testing [51, 52, 128, 129, 159, 160, 161, 171, 172, 173], the limited body of academic work regarding these tests focuses only on technical strategies for generating vulnerability unit tests. My efforts to understand how developers write *functional* security unit tests is motivated by the need to address this gap so that similar techniques can be applied for this purpose.

Mohammadi *et al.* have created a program-analysis based technique for automatically generating unit tests to detect cross-site scripting (XSS) vulnerabilities [128, 129]. One benefit of this approach is that it bases the test cases, inputs, and subjects-under-test only on the project itself, meaning a source of external information is not needed to identify these components. This is possible because a pattern-based approach can be used to identify vulnerabilities like XSS in different contexts, in this case *"improper encoding of untrusted data"* [129]. Other vulnerability security testing techniques (*i.e.* static/dynamic analysis, fuzzing) are also based on these patterns.

Salva and Regainia have made several contributions [159, 160, 161, 171, 172, 173] towards helping developers in the early stages of the SDLC plan and generate vulnerability unit tests after threat modeling has been conducted. They first devised an approach for curating a knowledge base for security patterns and attack sequences [160, 161, 172, 173]. In later work [159, 171] they extend the approach towards actionable support; the knowledge base is used to guide developers in creating Attack-Defense trees, which are used to automatically generate abstract vulnerability test cases in the Gherkin *Given, When, Then* format [183]. This approach is an important effort towards creating developer supports, and generating test cases in this format allows them to be applied at different scopes and test levels in addition to unit. However, the knowledge base focuses on "black box" patterns and attack sequences, therefore functional security unit tests are out of scope for this system. Nonetheless this approach could potentially be extended or adapted to support developers planning functional security test cases if a comprehensive knowledge base could be curated.

My Contributions

To create similar support systems for functional security unit testing, the lack of resources outlining patterns or unit-level test cases forming the basis of these works must be addressed. My work towards understanding the current state of practice for security unit testing in terms of practices, perspectives, and pain points are the critical and previously missing pre-requisite for doing so. I discuss in Chapter 6.3 how existing online guides lack the data that would be needed for this effort to succeed, and make data-driven recommendations based on perspectives (Chapter 6) and pain points (Chapter 5) observed. I have also produced a more immediate and actionable resource by using findings from my study of authentication unit tests (Chapter 4.1) to develop a token authentication unit testing guide that presents test cases in a pattern-like format geared towards providing step-by-step guidance for developers to implement them [82]. To produce the generalization and level of detail needed for widespread use, no technology or context-dependent data was included in our descriptions of the emergent test cases, which were also presented as a breakdown of their key elements and organized by scenario.

Important!
This work uses the term (security) **unit tests** to represent both unit *and* integration level (security) tests written by developers using automated testing frameworks (*e.g.* JUnit)

More information about (general) automated unit and integration testing is provided in Section 2.4.

2.3.4 Human Factors Influencing Developer's Security Practices

An increasing number of organizations are shifting to process models that incorporate security into every phase of the software development lifecycle. In response, research studying technical and human aspects of secure development and testing have grown in popularity [4,5,12,13,14,22,42,98,130,131,146,152,200,209]. Unfortunately, there are no works providing specific usage measurements for security *unit* testing. A few researchers have reported other observations relating to security unit testing, but most relevant for this work, there are several studies of developer's perspectives and challenges regarding secure development & testing practices.

Assal and Chiasson have reported three studies related to developer's software security (de)motivations, perspectives, and practices [12, 13, 14]. In [12] they interviewed 13 practitioners and performed a grounded theory analysis of the transcripts to identify motivators and de-motivators for software security practices. They found that the participants were motivated by a desire for *"self-improvement"*, as well as *"professional responsibility"*, *"company reputation"*, and *"pressure"* by superiors. Demotivates identified include lack of confidence, interest, relevance (*"not my responsibility"*), and perceived value. In [13] they studied of developer's attention to security across lifecycle phases, they found that *"security consideration is virtually non-existent"* in developer testing phase. They note that functionality is the driving motivator at this phase. Their latest study [14] was a survey of 123 North American software engineers their key findings was that developer's were self-motivated to practice secure software development but their primary barrier was a lack of organizational and process support (*e.g.* security plans). This reflects a need for future research to include organizational factors or as the authors put it *"look beyond the individual"* when investigated security practices. They also noted that participants reported little "security effort" during developer testing, consistent with their previous findings [13].

In [200], Van der Linden *et al.* studied security rationale of 44 mobile app developers, noting a pattern of relying on external testers for security concerns instead of writing test code. Cruzes *et al.* [42] noted in their case study of four agile development teams that there is a lack of guidance and knowledge related to security testing. For the teams studied, automated testing was not security-focused. Morrison *et al.* [130] conducted a case study of security practices in use at IBM and noted that developers must write a unit test for any changes made, implying that security unit testing is practiced but not necessarily motivated by security. Rahman and Williams studied practitioners perspectives and practices related to security in DevOps environments through grey literature review ($n = 66$) and nine practitioner interviews [152]. They found three artifacts related to automated testing, and five of the nine practitioners interviewed felt automated testing was beneficial for security.

2.4 Automated Unit and Integration Testing

Modern development models recognize the benefits of early and repeated testing and work to "shift testing left" by incorporating appropriate testing activities into each lifecycle phase [184]. Unit and integration tests are **written by developers** to evaluate system behavior at different levels **during**

implementation. As their names imply, **unit tests** focus on the behavior of isolated 'units' (classes or methods) in the code, while **integration tests** verify interactions between multiple units or components [28]. A suite (collection) of unit and integration tests can be executed *automatically* as part of a continuous integration (CI) pipeline to test changes as they are submitted or *manually* by a developer to test changes as they are implemented.

In practice, the distinction between unit and integration tests can be unclear; most automated testing frameworks can be used to write and execute both, and defect detection performance has been shown to be similar in prior studies [141,197]. Developers may not care about this distinction in practice [194] and do not always specify which they are writing in Q&A posts or grey literature.

Key Components of Unit and Integration Tests

The following terms represent key elements of unit and integration tests. Their definitions are given as they were applied for this work, and are heavily referenced in Chapters 4 & 5:

Context: Pre-conditions and test setup *e.g.*, object initialization

Subject-Under-Test: The component, method *etc.* being evaluated in the test

Action: Manipulations of the subject-under-test, *e.g.* a method call

Scenario: A single phrase representation of a unique combination of *context* and *action*

Condition: A state or event that influences the action, *e.g.* a `null` argument to a method call

Expected Outcome: The desired state of the subject-under-test *after* the action is performed, *e.g.* a method call returning `True`

Test Case: A unique combination of a *context*, *action*, *condition*, and *expected outcome* that is represented in natural language

The Software Testing Lifecycle

The testing lifecycle consists of four *phases*: planning, design, construction, and execution. First, in the **planning** phase, developers determine what test cases they need to write for the code-under-test. The most common stopping criteria (how *much* to test) used by developers is coverage, *i.e.* the percentage of code branches, paths, or statements that are exercised by a test suite. During the **design** phase, test fixtures are created to initialize the system-under-test and satisfy any preconditions needed to create each test case. Developers then use these fixtures and an automated unit testing framework (*e.g.* JUnit) in the **construction** phase to implement test methods for the planned test cases. Finally, after test **execution** the results are evaluated: if the value matches a pre-determined expected outcome the test passes, otherwise it fails.

2.4.1 Use in Practice

Developer's adoption of unit and integration testing has been measured in numerous prior works [10, 26, 46, 69, 73, 74, 75, 83, 90, 91, 106, 115, 138, 168, 212]. A few studies focused exclusively on unit test practices [26, 46, 83, 168], but most considered software testing practices in general, which included findings related to unit testing [10, 69, 73, 74, 75, 90, 91, 106, 115, 138, 212]. Others scoped their investigation to practices within a specific country [69, 73, 74, 75, 138].

In 2004, Ng *et al.* studied testing practices in Austrailia [138] by surveying managers & testers from 65 organizations. The authors found that out of 42 (64.6%) organizations using a "structured testing methodology", only 16 (38.10%) employed "white-box" testing. Geras *et al.* conducted a similar 2004 study of testing practices in Alberta [75]; of the 55 practitioners surveyed fewer than 80% claimed to use unit testing.

In [91], Grindal *et al.* interviewed test managers from 12 software organizations to measure "test maturity" of software organizations. Participants were asked to self-report the percentage of their time spent in each phase of testing; an average of 20% was reported for component (unit) testing. The authors also noted that these findings were consistent with those of [138] and [75]. Also in 2006, Wojcicki and Strooper [212] reported that most (82.86%) of the 35 practitioners surveyed in their investigation of testing practices in concurrent systems claimed to use unit testing.

In 2010 Garousi and Varma replicated Geras *et al.*'s study [75] and found a slight increase in the use of unit testing since 2004 (96% of 35 companies) [73].

Lee *et al.* [115] surveyed testing practitioners in 2012 to gauge current practices and improvement areas. Specifically, they asked participants if they use software testing *methods* ("...techniques, procedures, patterns, or templates...") or *tools* ("...provide automated or semi-automated support...") [115] at each test level. For unit testing, 33% of the 20 participants claimed to use methods and 26% claimed to use tools. In Greiler *et al.*'s 2012 investigation of testing practices for plug-in systems [89, 90] they surveyed & interviewed 151 practitioners from the Eclipse (an open source plugin-based IDE) development community. They did not quantify use of unit testing but through grounded theory analysis of the interview transcripts, found that unit testing "plays a key role" in this community and was in fact the primary means of testing.

Garousi and Zhi (2013) [74] conducted a comprehensive survey of testing practices in Canada with 246 practitioners. Unit testing was used by 79.67% of participants, and the mean percentage of effort dedicated to unit testing was 20% (consistent with [91]). However, 59.35% reported using Test-Last Development rather than models promoting early testing like Test-Driven Development (20.33%) or Behavior-Driven Development ($< 20\%$).

In 2015, Garousi *et al.* surveyed 202 developers to "characterize and grasp a high-level view of" software engineering practices in Turkey [69]. The authors studied testing practices but did not report an exact measurement of unit test adoption. However the survey did reveal that "formal review of unit tests" was not often practiced, and the previously-reported [74] preference for Test-Last Development over TDD or BDD applied to Turkish developers as well. The same year, Beller *et al.* reported their findings from a field study of 416 software engineers writing Java code using the Eclipse IDE [26]; only 43% of participants wrote, executed, or read unit tests.

Finally, in a 2017 study I led a large-scale empirical study of the use of unit test patterns in open source software [83]. Towards our goal of measuring the use of unit test patterns [126] we mined and analyzed 82,447 open source repositories for software written in 48 languages. Ultimately, only 17.17% of these projects contained unit tests.

Table 2.1: Summary of Prior Findings for Practitioner's Use of Unit Testing

Year	**Study**	**Subjects**	**% Unit Testing**	**Alternative Finding**
2004	[138]	65	24.62%	
2004	[75]	55	$< 80\%$	
2006	[91]	12	*Not reported*	Mean 20% of effort spent on unit testing
2006	[212]	35	82.86%	
2010	[73]	35	96%	
2012	[115]	20	33%	
2012	[90]	151	*Not reported*	Unit testing "plays key role in Eclipse community"
2013	[74]	246	79.67%	Mean 20% of effort spent on unit testing
2015	[69]	202	*Not reported*	"formal review of unit tests" not often practiced
2015	[26]	416	43%	
2017	[83]	82,447 (projects)	17.17%	

Table 2.1 summarizes the research on unit test adoption. Over time, adoption rates do not follow a consistent trend but fluctuate within a range of 17.17% to 96%. It *is* interesting to note that the two studies which *empirically* measured use of unit tests [26, 83] reported rates $< 50\%$.

Three studies did not explicitly report the percentage of their subjects who use unit testing. Alternatively, Grindal *et al.* [91] reported that an average of 20% of development efforts were dedicated to unit testing (also found by [74]), Greiler *et al.* [90] found testing in the Eclipse community focuses on the unit level, and Garousi *et al.* [69] noted unit tests do not often undergo "formal review".

2.4.2 Human Factors Influencing Developer's Testing Practices

Several prior works also investigated practitioner's motivations, opinions and perceptions of testing practices [10,26,46,75,106,138,168,212]. While quantitative measurements of unit testing adoption are valuable, it is important to supplement such objective data with such context. In doing so, implications of the findings are clearer and can help researchers design future studies that are empirically relevant to practitioner's obstacles and needs.

Andersson and Runeson ran a workshop series and conducted interviews to study the verification and validation practices of 9 companies [10]. Concerning unit testing, they found that most module (unit) testing was done informally by developers. Ng *et al.* [138] asked their participants to describe any *barriers* they faced that prevented or dissuaded them from adopting software testing practices; the dominant barrier identified was a *lack of expertise*, followed by *time consuming to use.* As part of their investigation into testing practices, Geras *et al.* [75] noted that fewer management-level respondents (75%) reported their projects used unit testing than non-management respondents (92%), indicating a lack of communication between organization levels.

Per Runeson surveyed 17 practitioners about how they define unit testing and what they consider to be its strengths and weaknesses [168]. These practitioners felt that unit testing was a strong technique for developers to independently and continuously test small modules. However, they struggle with identifying appropriate units, documentation, test selection, metrics, stopping criteria, and reconciling costs. The participants of this study form a small but representative sample of organization size and domain which improves generalization, and the primary takeaway as proposed by the author is that the findings should guide larger future studies. Findings from a later (2008) study by Smith and Williams [182] investigating developer's techniques for determining when to *stop* writing tests supports the notion of "ad-hoc" testing by developers.

Kasurinen *et al.* [106] interviewed 55 practitioners and conducted a parallel qualitative study of

12 organizations to characterize the use of test automation and identify areas for improvement. This study did not explicit ask participants if they conducted unit testing, but measured their perceptions of the importance & benefit of each testing level and common automation techniques. Their participants felt that quality was built during development, not testing and favored late-stage test levels (*e.g.* functional, system) over early-stage test levels (*e.g.* unit, integration). However, they did consider unit testing one of the most efficient activities to automate.

Wojcicki and Strooper's study focused on practitioner's processes for selecting (which) verification and validation (V&V) techniques to use [212]. Their key findings were (1) unit testing frameworks were the primary method of automating V&V, (2) practitioners were very concerned about cost-effectiveness when designing their testing approach, and (3) practitioners want case studies evaluating V&V technologies to use *real* defects.

In 2014 Daka and Fraser surveyed 171 software developers about their unit testing motivation, practices, and problems to identify research problems that align with industry needs [46]. Their findings on unit testing weaknesses and struggles aligned with those of Runeson [168]: developers need more support determining *what* to test, but current research trends towards automated unit testing approaches that fail to appropriately address this need. Participants also felt that unit tests must be realistic and maintainable; this is an important consideration when developing guidelines & sample test cases.

Finally, Beller *et al.*'s 2015 field study balanced measurements of developer's testing practices (*what* & *how*) with probes into their rationales (*why*) as they investigated patterns and needs [26]. Their three key findings were (1) developers test less than they perceive/report, (2) testing, as measured within an IDE, is not a popular activity, and (3) Test-Driven Development is not a popular development model. This study highlights the importance of "mixed-methods" research when studying software engineering phenomena — there is a disconnect between practitioners expectations, beliefs, perceptions, and actual behavior. They also highlight the disconnect between researcher's beliefs regarding the wants and needs of developers and reality.

To summarize, below are the key findings from these studies:

- Unit testing is often an informal practice conducted independently by developers [10, 168, 182]
- There is a disconnect between different stakeholders (researchers, developers, managers) [46, 75, 212] regarding practices and challenges.
- Developer's perceptions of their unit testing behavior is not always consistent with empirical

observations and measurements of their actual practices [26].

- A lack of experience and guidance for unit testing can be a de-motivator or challenge for developers, especially concerning *what* to test [46, 138, 168].
- Unit testing is unpopular even though developers recognize the benefits [26, 106].

2.5 Empirical Studies of Developer's Q&A Posts

As knowledge workers, the tasks performed by software engineers are primarily related to finding, processing, and generating information [135]. In troubleshooting scenarios, wherein they seek information that will help them solve a problem, a common practice is to pose a question to their peers online. Stack Exchange is a network of specialized question and answer (Q&A) websites. Their flagship website is Stack Overflow (SO), an extremely popular site for computer programming Q&A and discussion [17]. Since it's debut in 2008 more than 21 million questions have been posted to Stack Overflow, and visiting the site is integrated into the problem solving processes of many developers [142]. Security Stack Exchange (SSE) is another popular Q&A community where developers and security engineers discuss topics related to Information Security [16].

This has made them popular amongst software engineering researchers (*e.g.* [8, 19, 23, 94, 95, 110, 116,117,118,119,124,136,163,165,195,205,214,217,219]), who have developed efficient and accurate techniques for mining, filtering and analyzing post data to study a diverse set of software development concerns and subjects. Meldrum *et al.*'s 2017 systematic mapping study ($n = 265$) [124] and Ahmad *et al.*'s 2018 literature review ($n = 166$) [8] have quantified the research community's growing interest in Stack Overflow, popular topics, and common analysis techniques. Barua *et al.* and Ye *et al.* have conducted broad studies using *all* posts on Stack Overflow to explore key discussion topics [23] and the structure and dynamics of Stack Overflow from a knowledge network perspective [217]. Other works have used Stack Overflow to study a wide variety of software development topics including but not limited to: cloud computer vision [43], mobile development [27,165,205], software testing [110], software architecture [195], web development [19], Apache Spark [163], Java [219], Docker [95], and deep learning [94].

Security Topics & Challenges

Several studies have used Stack Overflow to study developer's discussion topics and challenges related to security [118, 119, 136, 214]. Unlike this work, these studies did not include data from

Security Stack Exchange. Yang *et al.* [214] wanted to investigate what *security*-related topics developers ask questions about; to identify relevant posts, they used two tag-based heuristics. An automated approach was used to analyze and group posts by topic, specifically feature modeling and LDA. As a result they identified 30 topics, and investigated the distribution of posts to topic, most popular topic, and which topics were most difficult to answer questions about. From these insights the authors suggest researchers should investigate assistive solutions for the most popular and difficult topics, educators should focus on web security, and practitioners could use the difficulty metrics to assign topic-based work by level of expertise. Topic-based studies such as this demonstrate how this data can be used to identify specific technical concerns developers face relating to security, and these insights can help researchers maintain relevancy with their work.

Human factors relating to secure software development has become a popular research area, as we seek to understand how developers perceive, respond to, and overcome security challenges [2, 3]. Since Stack Overflow is widely-used by developers when faced with technical challenges, Lopez *et al.* [118, 119] conducted 2 studies which applied qualitative methods (coding) to analyze a small sample (20) of highly-rated security posts and their answers to explore the social context of security knowledge sharing. Their first analysis revealed insights into three dimensions: *advice and assessment*, *values and attitudes*, and *community involvement*. They found that security awareness grows through discussing suggested solutions, and although most questions are technical, developers also discuss rationale, return-on-investment, balancing developers needs with external influences (regulations, organization-level requirements), and the effectiveness and circumstances of suggested solutions. Developers perspectives are often embedded in answers to technical questions, and the authors identified common themes: principles, trust, fear, pride, and responsibilities. Given the evolving nature of security, questions "belong to the community" and discussion activity can persist years after the original posting date. [118].

In their second study, Lopez *et al.* focused on finer-grained details of participation in the same set of questions, examining features of the discussion environment such as time, edits, quoting, and naming techniques. They observed differences in the question and answer comment streams; for example, answer comment streams focus more on technical details vs the perspectives found in question streams. [119]. While a small sample was used for these studies, their findings reveal the multi-dimensional nature of security discussions on Stack Overflow, and support the notion that both technical (challenges, concepts) and social (perspectives, opinions) factors relating to security topics can be learned by studying data from this platform.

Nadi *et al.* [136] manually analyzed 100 Stack Overflow posts as part of a study conducted to understand tasks developers perform with Java cryptography APIs, and the challenges they face. Their study was motivated by a need to understand what developers need help with in this regard, and which support mechanisms would be most effective. Findings from the post analysis were supplemented with two developer surveys (question askers and API-users) and an analysis of 100 GitHub repositories using Java cryptography APIs. They observed that posters with some domain knowledge struggled primarily with correct usage of the API, such as the correct sequence of method calls needed to perform a task. Also noted was posters lacking domain knowledge struggled with choosing the correct algorithm for their needs or which libraries to use. These findings are of particular interest to this work, as most security controls are implemented using third-party libraries, which might influence the questions asked about testing.

Testing Topics & Challenges

Key discussion topics and difficulties related to testing have been studied from the general perspective by Kochar *et al.* [110] and for Android development specifically by Villanes *et al.* [205]. Kochar *et al.*'s study used a similar mixed-method approach that combines LDA topic model insights with deeper qualitative analysis. They identified eight key testing topics from over 38,000 testing-related questions. Consistent with this work, *Login* and *Client/Server* were identified amongst others unrelated to security (*e.g.* Test Framework) in the Q&A post analysis conducted for this work (see Chapter 4.1). They supplemented the topic results with several smaller exploratory analysis about temporal trends, view-based topic popularity, and a special look at mobile testing concerns. They also manually reviewed 50 posts to identify challenges; these were associated directly to topics but show that, similar to the pain points identified in this work (see Chapter 5), many relate to conceptual knowledge gaps and design confusions.

2.6 Grey Literature Reviews in Software Engineering

In empirical software engineering research, primary (generated by the individual) data about developer's practices, opinions, and challenges is typically collected from software artifacts (*e.g.* source code, repository metadata) and human-subject studies (*e.g.* interviews, surveys) with practitioners. Further, researchers often limit their search for related studies & data to peer-reviewed sources such as academic publications. While these are valuable sources of data for studying software engineering phenomena, there is another, under-utilized source of practitioner-generated data that provides tantamount data regarding their experiences, perspectives, beliefs, practices, and challenges. Referred to as *grey literature*, this includes non-peer-reviewed artifacts like wiki and news articles, blog posts, presentations, videos, Q&A forum posts, and white papers [70].

There are two key benefits to considering grey literature for empirical software engineering research. First, like interviews and surveys, the data comes from a primary source but collecting the data does not require analogous time, cost, or effort as recruiting participants & conducting an interview or disseminating & following up on a survey. Instead, such data can be mined and analyzed similar to other textual software artifacts.

Second, critically, prior studies have shown that developers find information generated by other practitioners as the most valuable [56, 155]. A 2003 study conducted by Rainer et al. analyzed transcripts of interviews with developers form 13 companies to investigate practitioner's opinion of software processes' and process opinion. The primary finding was that while developers want to use empirical evidence when formulating their opinions, they prefer information generated by other "local" practitioners. The authors infer that this preference correlates to a developer's ability to relate information to their specific situation [155]. In 2016 Devanbu et al. [56] interviewed over 500 Microsoft employees who collectively ranked research papers as the data source that *least* influenced their opinions.

To support researchers as they begin to consider grey literature in their work, several "methods" papers providing guidelines for the inclusion of grey literature in empirical SE studies have been published. In 2017, Austen Rainer examined how argumentation theory could be adapted to analyze blog posts authored by developers to study "practitioners' evidence, inference, and beliefs." [154], and in [156] Rainer and Williams propose a set of heuristics intended to "improve the relevance and rigor" of studies that include grey literature. Garousi et al. outlined best practices for conducting multi-vocal literature reviews that include grey literature in 2019 [70].

An important consideration when using grey literature is that, unlike peer-reviewed publications or objective software artifacts, quality, reliability, and accuracy cannot be guaranteed. To this end, Rainier and Williams have studied perspectives and techniques used by empirical software engineering researchers when evaluating the credibility of blog posts written by practitioners [211]. The major finding of this study was that researchers consider a variety of factors when evaluating credibility, but the four most-consistently reported factors were the clarity of writing and the reporting (by the practitioner) of empirical data, methods of data collection, and reasoning related to the post contents. Additionally, Rainer and Williams have curated a set of guidelines similar to [70] focused only on the use of blog posts [157]. Along with a suggested methodology for collection, filtering, and analysis the authors also enumerated the benefits & challenges of using grey literature, prior work, and potential resources.

In the empirical software engineering research landscape, grey literature has been included in a limited number of recent “multi-vocal” literature reviews [72, 139, 158] and at least one exclusively grey literature review [186]. Further, mining Q&A forums, specifically Stack Overflow, is gaining popularity; a 2018 literature review identified 166 publications which used Stack Overflow data from it’s creation to 2016 [8]. Baltes et al. [21] have also released a mining tool and dataset which expedites the collection process.

To summarize, empirical software engineering researchers recognize the benefits of analyzing grey literature to understand practitioner’s beliefs, perceptions, and practices and several recent studies have been published which use the technique exclusively or as part of broader literature reviews. Further, researchers have realized the need for and subsequently produced guidelines and heuristic to support future studies.

Chapter 3

Research Design

Motivation: This dissertation is motivated by the disconnect between promotion of security unit testing and the lack of empirical evidence on how it *is* and *can be* applied.

Goal: My goal was to understand the state of practice for security unit testing towards identifying data-driven, actionable strategies to promote wider adoption and mitigate observed challenges.

Approach: I took an empirical software engineering approach to meet this goal, conducting three mixed-method studies of artifacts created by practitioners — *unit test code*, *Q&A posts*, and *grey* (not peer-reviewed) *literature*. To establish a holistic framework for the state of practice, both technical and *socio*-technical (human) factors must be understood. Accordingly, these studies were designed to use the practitioner artifacts to examine this phenomena through three lenses: *Practices* (Chapter 4), *Perspectives & Guidelines* (Chapter 6), and *Pain Points* (Chapter 5).

A mixture of qualitative and quantitative methods have been applied to support the development of *data-driven* and *actionable* insights, which requires examining a study subject with breadth and depth [180]. Automated techniques for content analysis like unsupervised topic modeling make it feasible to extract quantitative measurements from large-scale datasets (*breadth*), which can then be contextualized and expanded (*depth*) with smaller, qualitative analyses *e.g.* data coding. Scripting tools were created to automate mining and processing of test code and Q&A post text.

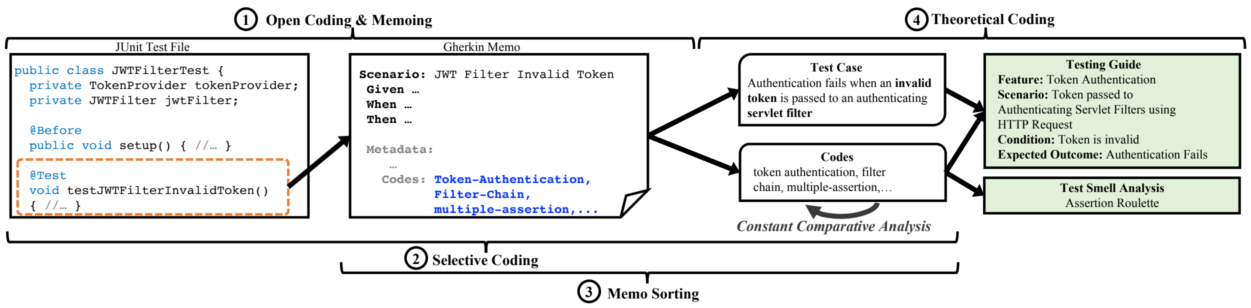
Chapter Organization: The rest of this chapter provides high-level overview of the three study designs and the methods used to collect, process, and analyze each type of artifact. Subsequent chapters synthesize findings from all three studies for broader discussion of the state of practice for security unit testing.

3.1 Study 1: Grounded Theory Analysis of Unit Test Code

Towards gaining a deeper understanding of how developers test security controls (Chapter 4), I designed a “classical” grounded theory inductive analysis [77] of **481 authentication unit tests mined from open source projects**. The **scope** of this study was limited to facilitate the manual efforts of qualitative analysis, but care was taken to address generalizability by selecting a popular language (**Java**), testing framework (**JUnit**) and security framework (**Spring Security** [9]).

A **grounded theory** (GT) [77] is a systematic inductive method for conducting qualitative research when little is known about a **phenomenon of interest**, which for this study was formally defined as *unit testing of authentication implementations*. Although developers also write unit tests for other controls (see Chapter 4.1), I chose to focused on authentication for this “deep dive” into security unit testing practices due to its ubiquity and complexity.

This methodology is driven by *emergence of concepts* [76,192], *i.e.* the progressive identification and integration of concepts from data that lead to discoveries directly supported by empirical evidence and grounded in the data [192]. The core activities are: identification of topic of interest, theoretical sampling, open, selective and theoretical data coding, constant comparative analysis, memo writing and sorting, and finally write up & literature review. Figure 3.1 shows an overview of this process.



```
public class JWTFilterTest {
  private TokenProvider tokenProvider;
  private JWTFilter jwtFilter;

  @Before
  public void setup() { //… }

  @Test
  void testJWTFilterInvalidToken()
  { //… }
```

```
Scenario: JWT Filter Invalid Token
 Given …
 When …
 Then …

 Metadata:
    …
   Codes: Token-Authentication,
          Filter-Chain,
          multiple-assertion,...
```

Test Case
Authentication fails when an **invalid token** is passed to an authenticating **servlet filter**

Codes
token authentication, filter chain, multiple-assertion,…

Testing Guide
Feature: Token Authentication
Scenario: Token passed to Authenticating Servlet Filters using HTTP Request
Condition: Token is invalid
Expected Outcome: Authentication Fails

Test Smell Analysis
Assertion Roulette

Figure 3.1: Grounded Theory Approach for Inductive Analysis of Security Unit Tests

Rationale: I chose to apply an inductive reasoning approach rather than the deductive method of formulating hypotheses at the beginning of the analysis process because the nature of the test cases (aside from relevance to authentication) was not know in advance. This allows all findings on what and how developers write tests to emerge from, and be supported by the data.

Additional Details: The unabridged methodology for this study can be found in Appendix A, including the data collection process and extended explanation of how we conducted these activities. This work is also published in the Proceedings of the 17th International Conference on Mining Software Repositories (MSR 2020) as *Did You Remember to Test Your Tokens?* [82].

3.2 Study 2: Mixed-Method Analysis of Developer's Q&A Posts

Towards a broader examination of what security unit tests developers write (Chapter 4.1) and any challenges they experience doing so (Chapter 5), I designed a mixed-method empirical study on **525 security unit testing Q&A posts mined from Stack Overflow and Security Stack Exchange**. Developers lean on their peers for problem solving help by posting questions to Q&A websites [142], making them a good source of data regarding security unit testing challenges. Formally, this study was designed to answer the following **research questions**:

RQ 1. What are the **key discussion topics** in Q&A posts related to security unit testing?

To answer this question, the **Latent Dirichlet Allocation (LDA) algorithm** [32] was used to conduct an unsupervised topic modeling analysis of the security unit testing Q&A posts. This approach is often used to study Q&A posts [19, 23, 94, 95, 110, 163, 165, 205, 214, 219] as it enables latent topics to be extracted from large corpora that would be unreasonable to analyze manually.

Rationale: The motivation for this analysis is twofold. First, prior studies show this approach is well-suited to identifying key discussion topics related to developer challenges. Second, this technique can be applied to large textual datasets, meaning the topics can be used to reason about findings from smaller, qualitative analyses. The topics identified in this study represent commonly-tested security controls, scenarios, and components.

RQ 2. What **pain points** do developers experience when writing security unit tests?

To identify potential barriers to shifting security testing left, **50 randomly selected Q&A posts** from the dataset were **manually analyzed using data coding** [39]. Conducting such in-depth analysis is critical to achieve the goal of creating an ontology of pain points unique to security-related tests enriched with sufficient contextual data to drive efforts to address them.

Rationale: The decision to use a subset of the Q&A posts collected was influenced by the rigor and effort required for manual analysis and designs used in prior work [110, 136] that supplement topic insights with qualitative study. My approach is further designed to define pain points that are *associated with* but not strictly *dependent on* specific technical contexts to improve generalizability.

Additional Details: The unabridged methodology applied for these analyses can be found in Appendix B. At the time of writing, this work has been accepted to the upcoming 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2021).

3.3 Study 3: Qualitative Analysis of Grey Literature

Towards exploring practitioner's perspectives and available guidelines for security unit testing (Chapter 6), I chose to conduct a systematic grey literature review of **51 practitioner-authored artifacts**. Practitioners publish a variety of security-related grey literature, ranging from anecdotal blog posts to expert-curated guides. This makes them a rich source of primary data to study practitioner's personal perspectives and examine available guidelines and support resources.

The **scope** of this study was grey literature that **explicitly discusses security unit testing** and was **authored by a software practitioner(s)**. After characterizing the type, tone, and source of each artifact to maintain the distinction between subjective and objective content throughout this study, they were **manually analyzed via data coding** [39].The grey literature collected and their characteristics are described in Tables 6.1, 6.2, & 6.3 and further discussed in Chapter 6.1.1.

Analysis was guided by the following **research questions**:

RQ 1. What security unit testing **perspectives, recommendations, and experiences** do practitioners share and discuss in grey literature?

Rationale: Practitioners seeking information often ignore peer-reviewed academic work in favor of grey literature from their peers in the form of white papers, online articles, blog posts, *etc.* [56]. Further, these sociotechnical insights are important to understand how security unit testing can be applied and integrated into developer workflows. This should influence the content and form of future guidelines, tools, and support systems towards this goal.

RQ 2. What **guidelines and strategies** for conducting or adopting security unit testing are provided in instructional grey literature?

Rationale: With the rising interest in automated security testing, the need for developer-centric support systems and workflow integration strategies will also increase. Yet there has been no prior investigation into the breadth, depth, and content of security unit testing discussions in grey literature to identify gaps between what information is *available* and what is *needed* to empower developers in their efforts to adopt this technique.

Additional Details: At the time of writing, this work is under preparation for future publication.

Chapter 4

Security Unit Testing in Practice

Automated security testing is an expert-recommended technique for shifting security into developer workflows [48, 55, 99, 203, 213], including at the unit and integration levels. Thus, it is reasonable to assume that developers are writing *some* unit tests for security controls, even if it is not an explicitly security-minded activity. In this chapter I discuss findings from two studies conducted to explore *what* security controls developers test at these levels and *how* they do so.

4.1 What Security Controls Are Unit Tested?

Insights from Developer's Security Unit Testing Q&A Posts

To examine what subjects-under-test developers discuss when seeking help on Q&A websites, we conducted an LDA topic modeling analysis on 525 posts related to security unit testing (Study 2). As Table 4.1 shows, the resulting topics (except *Error*) represent the security controls, components and scenarios that developers commonly test at the unit and integration levels.

Table 4.1: Controls, Components, & Scenarios Discussed in Security Unit Testing Q&A Posts

Topic Label	**Terms**
SSL Certificate	"certif", "ssl", "api", "applic", "error", "perform", "respons", "key", "server", "issu"
Encryption	"integr", "code", "unit", "encrypt", "write", "control", "call", "exampl", "method", "authent"
Cookie Authentication	"user", "app", "authent", "password", "cooki", "work", "load", "login", "web", "fail"
Token Authentication	"user", "applic", "api", "sign", "author", "token", "creat", "integr", "id", "work"
Basic Authentication	"password", "connect", "server", "usernam", "like", "would", "unit", "way", "basic", "client"
Login	"login", "page", "valid", "script", "work", "password", "credenti", "user", "usernam", "pass"
Protected Resource	"web", "authent", "access", "ad", "file", "like", "creat", "key", "want", "servic"
Client/Server	"jmeter", "unit", "certif", "client", "token", "server", "ad", "http", "send", "valid"
HTTP Request	"request", "servic", "secur", "applic", "log", "http", "authent", "code", "work", "would"
Error	"error", "develop", "work", "fail", "gener", "problem", "code", "server", "follow", "configur"

Encryption and *Authentication* were the most-discussed security controls; consistent with the complex and highly customizable nature of authentication [82], distinct topics emerged for three protocols: *Cookie*, *Token*, and *Basic*. Frequent discussion of concepts related to *Login* and *Protected Resource* indicate these are frequently-tested scenarios imply for access controls. Developers also frequently discussed *HTTP Requests*, *Client/Server* and *SSL Certificates* in their security testing posts, suggesting many seek help testing scenarios involve these components. The appearance of *Error* as a distinct topic is not surprising considering the data comes from Q&A websites and implies developers tend to seek help solving problems faced during test execution.

Insights from Grey Literature
In the grey literature reviewed to study perspectives (Study 3), practitioners suggested, instructed, or demonstrated a variety of *Use Cases* (Chapter 6.2) for security unit testing. As shown in Figure 6.7 practitioners often *recommend* concerns, controls, or scenarios they feel are well-suited for testing at the unit level. Most use cases were associated with vulnerability unit testing, such as *Input Validation*, *Buffer Overflow Attacks*, and *Man-in-the-Middle Attacks*. In more formal guidelines, these type of vulnerability tests are also discussed in the context of using unit tests to further ensure adherence to preventative secure coding practices [44, 169].

I chose to scope my research to *functional* security unit testing to address the lack of empirically-based insights into these tests. To this end, additional evidence of what functional security unit tests are *actually* written in practice emerged from the more instructional artifacts (Chapter 6.3), especially tutorials. Interestingly, **the subjects-under-test for which actionable guidelines were shared by practitioners and experts [169] map to five of the topics identified in the Q&A posts**: *Authentication*, *Protected Resources*, *Login*, *Encryption*, and *HTTP Request*. Unfortunately, the sparsity and extremely context-dependent nature (*e.g.* language,framework,tool) of these guidelines make it difficult to extract more generalized insights like those in this Chapter.

4.2 How Do Developers Unit Test Authentication?

Insights from Security Unit Test Code
For a deeper exploration of developer's functional security unit testing practices, a full grounded theory inductive analysis was performed on 481 real authentication unit test cases mined from open source (Study 1). The findings from this analysis are presented here as a taxonomy of 53 unique test cases, representing unique combinations of 17 scenarios, 40 conditions, and 30 expected outcomes that were evaluated in these tests. The flow diagrams created for each feature (Figures 4.1-4.5) serve two purposes: to make the purpose of each test case clear by explicitly outlining the key

elements (scenario, condition, expected outcome) and to emphasize that for each scenario there are *multiple* conditions and outcomes that should be evaluated.

Token Authentication

Figure 4.1 shows the 4 scenarios and 16 Unit Test Cases for using tokens to store and authenticate user credentials. A **token** stores the user's principal (*e.g.* username) & credentials (*e.g.* password) during and after authentication. Unit Test Cases for this feature focus on scenarios that occur during authentication of a token that has already been initialized with user data.

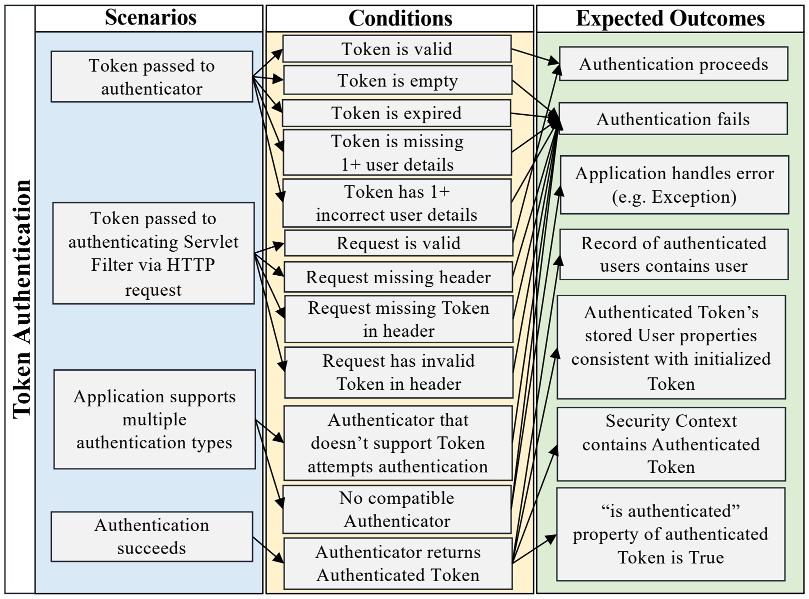

Token Authentication
Scenarios
Conditions
Expected Outcomes
Token passed to authenticator
Token passed to authenticating Servlet Filter via HTTP request
Application supports multiple authentication types
Authentication succeeds
Token is valid
Token is empty
Token is expired
Token is missing 1+ user details
Token has 1+ incorrect user details
Request is valid
Request missing header
Request missing Token in header
Request has invalid Token in header
Authenticator that doesn't support Token attempts authentication
No compatible Authenticator
Authenticator returns Authenticated Token
Authentication proceeds
Authentication fails
Application handles error (e.g. Exception)
Record of authenticated users contains user
Authenticated Token's stored User properties consistent with initialized Token
Security Context contains Authenticated Token
"is authenticated" property of authenticated Token is True

Figure 4.1: Unit Test Cases for Token Authentication

The first scenario is *Token passed to authenticator.* Tests for this scenario ensure that authentication fails when the object responsible for authenticating is given tokens containing four different types of invalid user data, such as an incorrect password. An additional test checks the 'happy path', where authentication proceeds when the authenticator is given a token with valid data. The second scenario is *Token passed to authenticating Serlvet Filter via HTTP request.* Unit test cases for this scenario verify that when a Filter intercepts and attempts to authenticate incoming HTTP requests, authentication fails when the request is invalid, such as one without a header or with an invalid token. Similar to the first scenario, there is an additional test case ensuring that authentication proceed with a valid request. Next is the *Application supports multiple authentication types* scenario. Many applications support different types of authentication: Username/Password, OpenID, OAuth 2.0, etc. For example, consider many websites allow you to login using your Facebook or Google credentials. The two tests for this scenario verify that when an incompatible authenticator attempts authentication or there is no compatible authenticator, then authentication is not successful. For example, a JAAS authentication token should not be successfully authenticated by an OAuth authenticator. The last scenario is *Authentication succeeds.* Each of the three tests have a unique expected outcome reflecting three events that should occur in this scenario: the user should be recognized as an authenticated user, the Security Context should contain the authenticated token, and the user's properties should be stored there correctly.

Token Manipulation

Figure 4.2 shows the five scenarios and 15 test cases related to *manipulating* tokens used for authentication. This a distinct feature from Token Authentication that focuses on the state of the token and it's stored properties *before* and *after* authentication occurs. Token creation, access, and validation scenarios should be tested with correct *and* incorrect inputs.

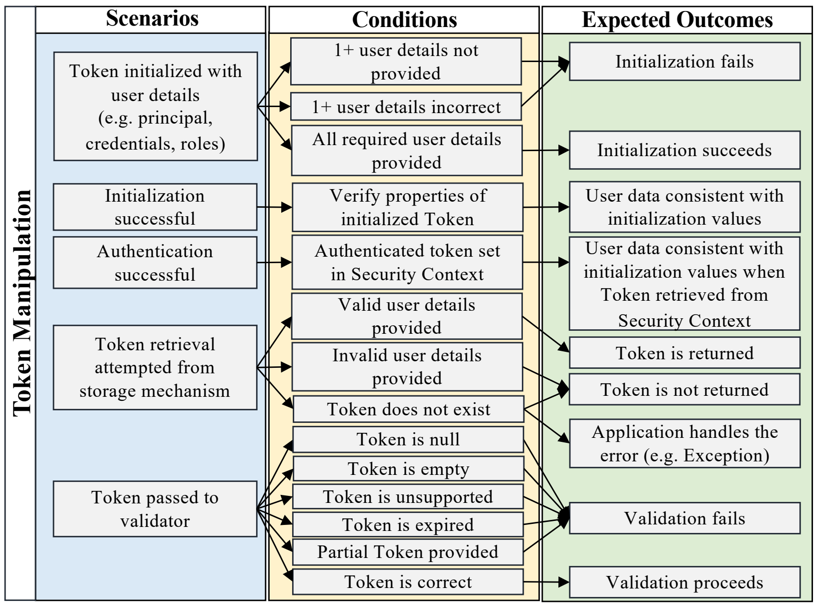

Token Manipulation
Scenarios
Conditions
Expected Outcomes
Token initialized with user details (e.g. principal, credentials, roles)
Initialization successful
Authentication successful
Token retrieval attempted from storage mechanism
Token passed to validator
1+ user details not provided
1+ user details incorrect
All required user details provided
Verify properties of initialized Token
Authenticated token set in Security Context
Valid user details provided
Invalid user details provided
Token does not exist
Token is null
Token is empty
Token is unsupported
Token is expired
Partial Token provided
Token is correct
Initialization fails
Initialization succeeds
User data consistent with initialization values
User data consistent with initialization values when Token retrieved from Security Context
Token is returned
Token is not returned
Application handles the error (e.g. Exception)
Validation fails
Validation proceeds

Figure 4.2: Unit Test Cases for Token Manipulation

The first scenario for this feature is *Token initialized with user details.* The three tests for this scenario check that initialization fails when one or more user details (such as username or password) are missing or incorrect, and proceeds when valid details are provided. The *Initialization succeeds* test ensures that the user data stored in the newly-initialized token are consistent with the provided values when retrieved. After initialization but before authenticating, the user data properties stored in the token should be checked to ensure it was stored correctly and can be retrieved. These properties should also be verified after *Successful authentication*, because unlike the companion scenario for *Token Authentication*, this test case evaluates consistency of the authenticated token's user data *before* it is stored in the application's *Security Context*, which stores all active security information. The fourth scenario is *Token retrieval attempted from storage mechanism.* For authentication implementations with storage mechanisms (*e.g.* OAuth 2.0), token *retrieval* should be tested to ensure that the application validates inputs for token requests. Two test cases check that no token is returned for requests with invalid user data or for non-existent tokens. A third test ensures that the application raises an exception for the non-existent case. Finally, a token should be returned when valid user details are provided in a request for an existing token. The fifth scenario is *Token passed to validator* containing five tests; validation should fail when an invalid (null, empty, unsupported, or partially provided) token is provided and pass for a valid token.

Refresh Token

Figure 4.3 shows the two scenarios and ten test cases related to *Refresh* tokens, which are specific to OAuth 2.0 authentication. These tokens are *renewals* of an original "access" token. This approach enables tokens to have short expiry dates, and generating new tokens does not require frequent re-authenticating. Applications supporting this refresh tokens should be tested to ensure that the properties of new tokens are consistent with the original token and creation/access behaviors only succeed with valid inputs.

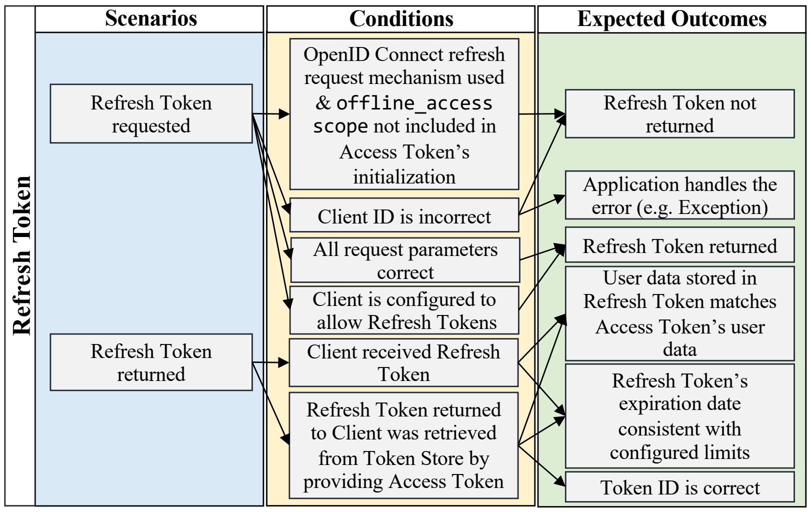

Refresh Token
Scenarios
Conditions
Expected Outcomes
Refresh Token requested
Refresh Token returned
OpenID Connect refresh request mechanism used & offline_access scope not included in Access Token's initialization
Client ID is incorrect
All request parameters correct
Client is configured to allow Refresh Tokens
Client received Refresh Token
Refresh Token returned to Client was retrieved from Token Store by providing Access Token
Refresh Token not returned
Application handles the error (e.g. Exception)
Refresh Token returned
User data stored in Refresh Token matches Access Token's user data
Refresh Token's expiration date consistent with configured limits
Token ID is correct

Figure 4.3: Unit Test Cases for Refresh Tokens

The first scenario is *Refresh token requested.* Two unit test cases for this scenario ensure that a token is not returned when the Client ID is incorrect or (for applications using the OpenID Connect refresh request mechanism) the "offline access scope" parameter is not included in the access token's initialization. Another test checks that when request parameters are correct and the client is configured to allow refresh tokens, a refresh token is returned. Five tests verify that when the client receives the requested refresh token, the user properties are consistent with the original token, the new expiry date is correct, and the token ID is correct.

Login

Figure 4.4 shows the three scenarios and seven unit test cases for the Login feature. Login behavior should be tested to ensure that the application appropriately responds to successful & unsuccessful login form submissions and unconventional request behaviors.

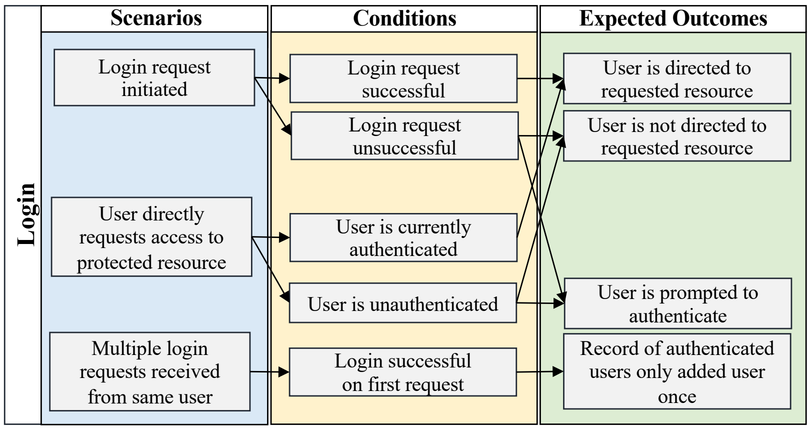

Login
Scenarios
Conditions
Expected Outcomes
Login request initiated
User directly requests access to protected resource
Multiple login requests received from same user
Login request successful
Login request unsuccessful
User is currently authenticated
User is unauthenticated
Login successful on first request
User is directed to requested resource
User is not directed to requested resource
User is prompted to authenticate
Record of authenticated users only added user once

Figure 4.4: Unit Test Cases for Login

The first scenario, *Login request initiated* is related to form submission and its 3 test cases ensure that users are directed to the requested resource if login is successful and unsuccessful login results in the user being denied access and (re-)prompted to authenticate. The second scenario is *User directly requests access to the protected resource.* For example, consider an attempt to navigate directly to your personal Facebook page instead of the homepage. The 3 tests for this scenario ensure than an authenticated user is granted access and (similar to failed login) unauthenticated users are denied access and prompted to authenticate. Finally, if *Multiple login requests are received for the same user* (*e.g.* repeated clicks of the 'submit' button), the application's record of authenticated users should not have a duplicate entry for the user.

Logout

Figure 4.5 shows the three scenarios and four test cases for the Logout feature. These should be tested to ensure that the application state is updated and the User is redirected appropriately.

The first scenario is *Logout triggered by user* (*e.g.* hitting the 'logout' button) and 1 test case ensures that the user is redirected to a designated location that is 'safe' for unauthenticated users, such as a login screen. The second test case is for the *User is currently logged in* scenario. Applications should be expected to automatically trigger a logout if the user's credentials are no longer valid. For example, if you change your Facebook password on a computer, you should be automatically logged out of the mobile app until you re-authenticate with the new password. The last scenario, *Logout complete* has 2 tests that check application state: the user's token should now be invalid and the Security Context should have a null authentication token.

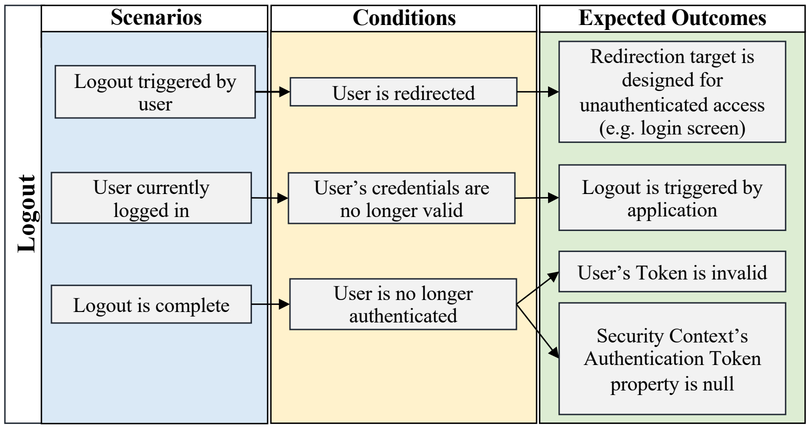

Logout
Scenarios
Conditions
Expected Outcomes
Logout triggered by user
User is redirected
Redirection target is designed for unauthenticated access (e.g. login screen)
User currently logged in
User's credentials are no longer valid
Logout is triggered by application
Logout is complete
User is no longer authenticated
User's Token is invalid
Security Context's Authentication Token property is null

Figure 4.5: Unit Test Cases for Logout

4.3 Conclusion

A majority of the authentication scenarios are concerned with tokens and user credentials. We believe the dominance of these type of tests is due to the design of the Spring Security API, which provides *interfaces* for many classes used for the creation and retrieval of user details (*e.g.* username, password) or token manipulation with OAuth2. Framework users are required to implement application-specific methods for these interfaces, resulting in the many scenarios from the dataset focusing on these properties. The complexity of security frameworks and more general implementing security controls, particularly authentication was an influential factor for many of the pain points identified for security unit testing; see Chapter 5 for details.

4.4 Study Limitations

Here I discuss the limitations of the authentication testing practices study in terms of validity threats and how we mitigated them. Threats related to the Q&A analysis are found in Chapter 5, and those related to the grey literature review in Chapter 6. For this study, we applied the core activities of grounded theory to develop a taxonomy of authentication unit test cases in order to understand how this security control is tested in practice.

Qualitative data-coding analyses must consider whether emergent codes accurately represent the source contents; this was of great importance for this study, as we sought to identify and re-present test cases in a different format. To mitigate this threat, we closely followed the key principles of grounded theory to conduct the open, selective, and theoretical coding and consistently reviewed the process to avoid deviations. Further, the resulting codes were peer reviewed to ensure consistency.

The manual analysis approach used to analyze the of unit test methods can be prone to bias and errors, which we mitigated with frequent review and discussions of agreement. The coding task was performed by two researchers with multiple years of experience with the Java programming language, JUnit, and automated unit testing. Emerging codes were constantly discussed and compared among existing ones to observe commonalities and differences. Feedback and interviews with 2 practitioners and 2 authors of dataset tests, who found the results valuable and important in practice were conducted (see [82]) to show the study's *practical* significance.

This analysis of authentication unit testing practices was based only on open-source projects, with further limitations being that the source code was implemented using Java and Spring Security and tested using JUnit. It was important to collect as may tests as possible for such a state of practice investigation, and mining GitHub repositories is often used for this purpose. To mitigate generalization issues, the findings extracted from this data are represented in a generic format that is applicable to other security frameworks. As argued by Wieringa et.al [210], describing the context of studied cases as we did allows us to "generalize" our findings by analogy; *i.e.* findings may apply to other security frameworks that provide an API for implementing token based authentication, token manipulation, refresh tokens, and user login/logout. To verify that the derived test cases are generalizable to other *Java* web security frameworks, documentation was reviewed for JAAS and Apache Shiro. Only framework-agnostic objects (*e.g.* Security Context, Servlet Filter) are identified by name in test cases. Further, including other languages or frameworks would have minimal impact on the final results, as token authentication follows a generic flow [179] and the APIs use language-agnostic protocols such as OAuth2 (RFC 6749) and OpenID Connect.

Chapter 5

Security Unit Testing Pain Points

Insights from Developer's Security Unit Testing Q&A Posts
Security, as a non-functional quality attribute, has traditionally been tested late in the development lifecycle. In most cases these tests were executed manually on fully-functional system instances by security experts with no involvement in their design or construction [62]. However, the evolution of software engineering processes and models towards *shifting security left* [203] has motivated the integration of automated security testing into developer workflows.

For developers, security unit testing is a non-traditional application of infrastructure and practices they are already familiar with, and presently there is little insight into any potential barriers that would impede explicitly security-focused testing at this level. These challenges must influence the creation or enhancement of strategies and guidelines, as there is already an observed gap between available security resources for developers and the tasks they struggle with [2, 3, 136].

5.1 The Unique Pain Points of Writing Security Unit Tests

From qualitative analysis of 50 randomly-selected Q&A posts emerged seven pain points uniquely experienced by developers when writing unit tests involving security controls. Summarized in Table 5.1, they span all test phases and affect tests involving the 9 security controls, scenarios, and components identified by the topic model (Table 4.1) as discussed in Chapter 4.1.

Mocking Security Components
Before individual test methods can be written, a test environment (*i.e.* fixture) must be setup to satisfy *preconditions* that recreate the system state needed for each test scenario. For example, an active session with an authenticated user must be established to test logout scenarios for a

Table 5.1: An Ontology of Security Unit Testing Pain Points from Q&A Posts

Pain Point	**Description**	**Example (Excerpts from a Related Question)**
Mocking Security Components	Developers struggled to correctly design and configure mocks for complex security components or providers implemented using third-party security frameworks (*e.g.* Spring Security) and had trouble debugging associated test failures and errors.	*"I'm **trying to test my spring OAuth2 authorization and authentication...using spring's MockMvc class**...The fundamental issue I'm facing is the fact that **my custom authentication provider is never called** even if I have registered it as one of the authentication providers used by spring security." (Post ID: 49079406)*
Interacting with Security Services and APIs	Not knowing "how to test" scenarios that involve distributed or third-party security services and APIs (*e.g.* OpenID, Paypal) can be an obstacle, often due to uncertainty of what should be simulated.	*"I want to write **unit test to test my Paypal Express Checkout integration**. I have **problem in the step where buyer authorize payment** in Paypal screen...**Is there a way to simulate this action in my test code?**" (Post ID: 19763494)*
Creating User Fixtures	The unique ways security controls interact with user data make this task particularly frustrating when testing login and password validation scenarios. Developers struggle to understand how to design appropriate fixtures, and incorrectly configured credentials can lead to confusing test failures.	*"**I've verified that [fake user fixture] is correctly being loaded** into the test database time and time again. I can grab the User object...I can verify the password is correct...The user is active. **Yet, invariably, [login command] FAILS**. I'm baffled as to why. I think I've read every single Internet discussion pertaining to this." (Post ID: 2619102)*
Bypassing Access Controls for Protected Resources	The procedure for simulating authenticated state or "skipping" authentication often depends on the system-under-test's design and the security tools, libraries, etc. used to implement these controls. Developers sought help designing tests and debugging errors and failures caused by missing or incorrectly-implemented steps.	*"I currently have an ASP 5/ASP Core Web API that I need to **integration test with the OWIN Test Server**...I use IdentityServer as the authorization server in production and **I do not want to include the authorization as part of my integration testing**...How can I successfully **fake authenticate the user and bypass the [Authorize]**...?" (Post ID: 37223397)*
Designing Authentication Flows	Due to the complexity and framework-specific properties of most authentication control implementations, developers had a hard time identifying the sequence of events ("flow") needed to recreate low-level scenarios and mock components.	*"...**I want to login a user...using forms authentication**...My question is **how do I write a test to justify this code?** Is there a way to check that the `SetAuthCookie` method was called with the correct parameters? Is there any way of injecting a fake/mock FormsAuthentication? (Post ID: 366388)*
Creating HTTP Request Objects	Errors caused by incorrectly configured login routes and trouble attaching certificates, csrf tokens, or credentials to request objects were obstacles to testing scenarios involving security controllers and authentication. Usually developers knew how to do this in production but were confused about how to do so in their test environment.	*"...In writing **functional tests for [their API's user authentication controller]**...I am **running in to an issue testing HTTP Basic auth**... I have found numerous blogs that mention [code snippet] should be used to **spoof headers**...[authenticate method] does not see the headers and therefore is **returning false even in the presence of valid credentials**. (Post ID: 1165478)*
Configuring Systems for Test Environments	Identifying the appropriate system settings, build configurations, *etc.* that must be adjusted, and the correct approach to change them, was a painful process for developers who wanted to use custom or mocked components. Settings related to certificates and (real) distributed components were especially difficult.	*"I have **tried instructing Maven to use a local keystore**, the one that comes with the JRE, in an effort to **keep the expired cert** [used by Embedded Glassfish] **from being used**...the **expired cert is still being found** in whatever trusted keystore it defaults to." (Post ID: 18304232)*

web application. Fixtures are often initialized in `setUp` methods accessible by test methods to ensure the consistent system state needed for repeatable tests. As part of this process, it is often necessary to simulate (*i.e.* mock) some system components, objects, or functionality. For example, edit or delete scenarios may be tested with fake users to avoid compromising the confidentiality, integrity, or availability of real user's data. **Mocking security-related components was the most frequently-observed pain point in our entire analysis.** Attempts to write tests without mocking (when appropriate) or incorrectly implementing a mock often lead to errors and unexpected test failures that were difficult to debug. The rationale behind the mocking decisions observed were not investigated in this study but Spadini *et al.* [188] suggests developers are often influenced by application-specific factors and individual preferences. Correctly recreating interactions between mocked fixtures and "real" system components can also be a frustrating process.

In the example given for this pain point in Table 5.1, the post author is trying to test OAuth-based authentication and authorization controls that were implemented with the Spring Security framework. Their test fixture uses the `MockMVC` class provided by Spring Security to simulate communication with API endpoints. Despite referencing multiple tutorials and related Q&A posts, the author can not figure out why their custom authentication provider is not being called in this context, which causes the given test method to throw a 401 error when executed. None of the three answers to this question were accepted, but the discussions indicate that the author has missed a configuration step.

Interacting With Security Services & APIs

A wide variety of services and APIs are available that developers can use for their control implementations. This is considered a good practice [121] because it reduces the developer's responsibility for correctly implementing the actual functionality of complex controls. Instead, developers integrate them into their code bases and configure mechanisms to enable distributed communication. Many of the posts examined involved distributed third-party services for authentication like OAuth, Okta, and OpenSSO, and OpenID. Developers struggled to recreate these interactions within their test environments. They asked for help constructing tests for scenarios that require accessing and manipulating session properties, communicating with an application's web API, and communicating with servers within test environments. Interactions between services and APIs that are distributed (outside of the test environment) and/or controlled by a third-party were especially difficult.

Other types of services were also being used in the posts examined; the example post provided for this pain point in Table 5.1 is asking how to unit test interactions with a Paypal (financial transactions) API. Based on the question description, the author was attempting to communicate with the API's live interface from a sandbox account, but did not know how to simulate buyer authentication. In this case, the accepted answer suggests to mock the interface instead, and links to documentation for the sandbox to help.

Creating User Fixtures

Security controls are designed and built into software to protect the confidentiality, integrity, and availability (CIA) of system services and data. The these controls must therefore be tested to ensure they have been correctly implemented and data is not compromised when exercised under different conditions and scenarios. Naturally most security functionality involves manipulating data related to *users*, especially credentials. Creating user fixtures may be a painful experience when testing other components like databases that interact with user data. However, testing security control behaviors requires unique access to and manipulations of user data, namely credentials, that warrant its inclusion as a *security* unit and integration testing pain point. Namely, developers struggled with user fixtures when testing login and password validation scenarios. There was a clear lack of awareness that the unique relationship between security controls and users should influence the design of these fixtures. Posts seeking design recommendations ("How do I test this?") indicate developers did not know how to create user objects, and discussed options like creating local user objects with hard-coded credentials and dynamically initializing fake users to store in a database. Developers also had a hard time debugging test failures caused by mis-configured user fixtures.

Table 5.1 lists an example question for this pain point that demonstrates these challenges. The author cannot figure out why a login test for their Django application fails, describing their fixture setup and the code used to initialize users. The accepted answer points out that this problem can be traced to incorrect initialization of the user object. The question author incorrectly initialized their user due to an apparent misunderstanding of how the user creation method handles the credential parameters. accidentally set their raw password string as the hashed value. Because they did not realize this method interpreted the password argument as the hashed value, they used the same string to call the login function. The author allegedly looked for answers elsewhere (*"I think I've read every single Internet discussion pertaining to this."*) before posting, but does not disclose whether they checked documentation for the framework used to implement the login function.

Bypassing Access Controls

An important observation stemming from this pain point is that developers may need to write tests involving security controls even when they are not the subject-under-test. *Protected Resources* was identified as key topic in the full dataset, and in the manual analysis a clear pattern was observed in which developers sought help designing and debugging tests that attempt to bypass access controls.

The appropriate procedure to achieve this often depends on the system-under-test's design and the security tools, libraries, etc. used in the implementation, and several posts sought help debugging errors and failures caused by missing or incorrectly-implemented steps.

In the example post for this pain point (Table 5.1), the author asks how to write tests for their web API that bypass the authorization server used in production. The accepted answer is from the post author themselves, who seems to have identified the correct configurations needed to do so. These observations also show pain points can emerge even when developers want to avoid testing security-related scenarios.

Designing Authentication Flows

The fine-grained nature of unit and integration testing forces developers to recreate the scenarios-under-test using a similar process applied to implement them, *i.e.* executing a sequence of statements calling methods or manipulating isolated components. The complexity of security control implementations made it difficult for developers to identify correct sequences, or "flows" for many scenarios, but observations made during post analysis indicate developers had an especially hard time designing *authentication* flows. This may be influenced by the wide variety of protocols that can be used, as evidenced by the emergence of *Token, Cookie, & Basic Authentication* as distinct key discussion topics in the entire post dataset.

The example post (Table 5.1) for this pain point follows a common ("how do I test this?") question pattern, in this case asking how to test form authentication (the protocol was not specified). The only (accepted) answer provides a code snippet explaining the technology-specific process, but neither this response or any of the comments link to any resource that would help. This is consistent with other studies highlighting the need for more example-based [136] security documentation and usage guidelines [3] for testing at this level.

Creating HTTP Request Objects

HTTP Requests are an essential aspect of distributed interactions which are a significant factor for web application development, including but not limited to *Client/Server* communication. Security functionality is often facilitated through such requests (*e.g.* submitting a login form) due to the distributed nature of security controls implementations. Even though request-based communication is a well-known concept that is not specific to security *or* testing, developers in the posts analyzed described challenges that made this task a uniquely painful experience when writing security unit and integration tests. They had the most trouble constructing requests for scenarios involving mocked security fixtures; when creating requests for authorization and authentication scenarios, developers asked how to attach *SSL Certificates*, credentials, and tokens. Other posts revealed difficulty or confusion when selecting the correct request headers and configuring routing behavior.

In the example given for this pain point (Table 5.1), the post author asks for help debugging an unexpected failure they encountered when testing the *Basic Authentication* component of their web API. In the description, the author implies they have tried using recommended methods for spoofing request headers from “numerous blogs” (not linked) with no success. The only (accepted) answer provides a solution via code snippet but does not link other resources.

Configuring Systems for Test Environment
Aside from creating test fixtures, other system settings related to build and runtime behavior may need to be specially configured to correctly recreate the necessary state using a mixture of real and simulated components. In this context, developers asked questions about how to secure storage of sensitive data, change settings for system components they have limited access to, configure build tools to use mocked certificates, and set runtime to trust all certificates. Configuring settings related to *SSL Certificates* was difficult for developers whether or not they were using real certificates.

In the example (Table 5.1) given for this pain point the post author is seeking help configuring their Maven build tool to reference a local keystore rather than the default used in production in an effort to circumvent an issue they had with their Embedded Glassfish testing server, which was using an expired certificate.

Common Factors Across Pain Points
Although the 7 pain points identified are distinct, they can co-occur, *e.g. Interacting with Security Services and APIs* and *Bypassing Access Controls for Protected Resources.* All pain points were influenced by the inherent complexity and customizability of security control designs. The third-party services, tools, libraries, and frameworks developers rely on to implement them abstract away much of the functionality, which can compound these negative influences. This is the driving force behind pain points such as *Designing Authentication Flows* but play some role in making all of the tasks in Table 5.1 painful.

5.2 Conclusion

The ubiquity of unit and integration testing means software projects usually already have the necessary infrastructure and writing these tests is a familiar practice for most developers. Therefore, shifting security testing to this level can be achieved by helping developers adapt their existing skills to this context instead of introducing an entirely new tool. It follows that the identified pain points may be barriers to success and should therefore influence the development of strategies and guidelines. I make recommendations towards this in Chapter 7.

5.3 Study Limitations

This section acknowledges the limitations of our Q&A post analysis in terms of threats to the validity and our mitigations. The topics mentioned herein were discussed in *Practices* (Chapter 4) as they represent subjects-under-test in the posts. Topics and pain points were identified by analyzing 525 Q&A posts from Stack Overflow and Security Stack Exchange. No claim is made that the results are exhaustive, but findings from the quantitative topic model experiment, which used the entire dataset, were compared to the qualitative findings and considered holistically to strengthen confidence in their representation and domain or technology dependent factors were not used to define pain points to ensure generalizability. The limitations of keyword-based methods of identifying posts related to a specific subject [23] can affect internal validity (*i.e.* capturing all available data), and this was mitigated in data collection by sourcing candidate posts from a security-focused site (SSE) and basing the SO search on a dataset of post IDs already verified as security related [114]. The keyword based filtering that was used to ensure relevant to unit and integration testing specifically was manually evaluated for recall and precision. Construct validity threats were mitigated using techniques from prior work [19, 110, 136, 195], *i.e.* randomly selecting 50 of the top 200 most-viewed posts.

Chapter 6

Perspectives and Guidelines

Insights from Grey Literature
To understand the state of practice, it is important to investigate human factors that may influence behavior. For this purpose, grey literature is an excellent source of primary data, as practitioners freely share their beliefs, expertise, and experiences with their peers through blogs, white papers, *etc.* [71, 157]. Practitioners also trust and prefer these artifacts over peer-reviewed, academic literature [56, 155] — hence by reviewing "grey" guidelines and references we can examine the contents of resources developers are most likely to refer to in practice. We conducted a manual analysis of 51 grey literature artifacts (Study 3) to explore how security unit testing is discussed in practice and examine existing resources. To study perspectives, we investigated the contexts in which security unit testing is discussed and what practitioners perceive as the motivations, limitations, and use cases for integrating this practice into development workflows. We also investigated how these concepts are presented in instructional resources and guidelines, but the primary focus for these artifacts was identifying what support was and was *not* provided for security unit testing.

6.1 Characterization of Practitioner-Authored Grey Literature

Before discussing the findings from this analysis, this section provides a characterization of the grey literature from which these insights were learned. As mentioned in Chapter 3.3, the artifacts were first classified by a number of properties, *e.g.* source, scope, and tone. These attributes are defined in Tables 6.2 & 6.3, which also show the percentage of artifacts with each value. Figures 6.1-6.5 provide visual representations of these distributions. Each artifact has a unique ID based on this classification: `[Tone][Type][Count]`. For example, the first *Blended Article* has ID `BA1` [47]. These identifiers are used throughout this work. A listing of the grey literature artifacts analyzed for this study and their individual classifications is provided in Table 6.1.

Table 6.1: Characterization of Grey Literature Artifacts Discussing Security Unit Testing

ID	Year	Type	Tone	Source	Scope	Purpose	Content
BA1 [47]	2015	Article	Blended	Company	Developer Security Testing	Recommend	Strategies and Tools
BA3 [111]	2005	Article	Blended	Media	Security Unit Testing	Persuade to Adopt	Use Cases
BA4 [132]	2020	Article	Blended	Company	DevSecOps	Recommend	Strategies and Tools
BB1 [15]	2018	Blog	Blended	Company	Security Unit Testing	Share Experiences	Tool Demo
BB11 [134]	2016	Blog	Blended	Company	DevSecOps	Recommend	Strategies and Tools
BB12 [147]	2019	Blog	Blended	Personal	Security Unit Testing	Share Experiences	Tutorial
BB13 [164]	2014	Blog	Blended	Personal	Security Unit Testing	Share Experiences	Tutorial
BB15 [181]	2019	Blog	Blended	Company	DevSecOps	Recommend	Strategies and Tools
BB17 [198]	2014	Blog	Blended	Company	DevSecOps	Recommend	Strategies and Tools
BB2 [20]	2014	Blog	Blended	Company	Developer Security Testing	Recommend	Strategies and Tools
BB4 [45]	2016	Blog	Blended	Personal	Security Unit Testing	Share Experiences	Tutorial
BB5 [54]	2018	Blog	Blended	Company	Security Unit Testing	Recommend	Strategies and Tools
BB6 [63]	2020	Blog	Blended	Company	DevSecOps	Persuade to Adopt	Strategies and Tools
BB7 [100]	2014	Blog	Blended	Company	Security Unit Testing	Demonstrate	Use Cases
BB8 [102]	2019	Blog	Blended	Company	DevSecOps	Recommend	Strategies and Tools
BB9 [105]	2019	Blog	Blended	Company	DevSecOps	Recommend	Strategies and Tools
BBk1 [30]	2016	Book	Blended	Personal	DevSecOps	Recommend	Strategies and Tools
BBk2 [122]	2006	Book	Blended	Personal	Risk-Based Security Testing	Recommend	Strategies and Tools
BP1 [104]	2018	Presentation	Blended	Organization	DevSecOps	Recommend	Strategies and Tools
BP2 [113]	2017	Presentation	Blended	Company	Security Unit Testing	Persuade to Adopt	Strategies and Tools
BP3 [41]	2019	Presentation	Blended	Government	DevSecOps	Recommend	Strategies and Tools
BW1 [53]	2020	White Paper	Blended	Company	DevSecOps	Demonstrate	Use Cases
BW2 [49]	2006	White Paper	Blended	Organization	Security Unit Testing	Demonstrate	Use Cases
BW3 [107]	2019	White Paper	Blended	Company	Developer Security Testing	Persuade to Adopt	Strategies and Tools
IA1 [38]	2013	Article	Instructional	Government	Developer Security Testing	Recommend	Strategies and Tools
IA2 [92]	2013	Article	Instructional	Media	Security Unit Testing	Advise	Tutorial
IA6 [153]	2015	Article	Instructional	Organization	Security Unit Testing	Advise	Tutorial
IB1 [7]	2017	Blog	Instructional	Personal	Security Unit Testing	Share Experiences	Tutorial
IB10 [207]	2017	Blog	Instructional	Company	DevSecOps	Recommend	Strategies and Tools
IB2 [34]	2017	Blog	Instructional	Personal	DevSecOps	Recommend	Strategies and Tools
IB3 [58]	2020	Blog	Instructional	Personal	Security Unit Testing	Share Experiences	Tutorial
IB4 [93]	2007	Blog	Instructional	Personal	Security Unit Testing	Share Experiences	Tutorial
IB7 [125]	2016	Blog	Instructional	Company	Security Unit Testing	Demonstrate	Use Cases
IB8 [202]	2020	Blog	Instructional	Company	DevSecOps	Recommend	Strategies and Tools
IB9 [144]	2020	Blog	Instructional	Company	Security Unit Testing	Advise	Tutorial
IBk1 [216]	2020	Book	Instructional	Personal	Security Unit Testing	Recommend	Use Cases
ID1 [66]	2016	Documentation	Instructional	Organization	Security Unit Testing	Advise	Tutorial
ID2 [86]	2020	Documentation	Instructional	Company	Security Unit Testing	Advise	Tutorial
ID3 [170]	2020	Documentation	Instructional	Company	Security Unit Testing	Advise	Tutorial
ID4 [178]	2017	Documentation	Instructional	Company	Security Unit Testing	Advise	Tutorial
ID5 [177]	2020	Documentation	Instructional	Organization	Security Unit Testing	Advise	Tutorial
ID6 [96]	2019	Documentation	Instructional	Company	Security Unit Testing	Advise	Tutorial
IG1 [6]	2019	Guide	Instructional	Government	DevSecOps	Advise	Strategies and Use Cases
IG3 [44]	2020	Guide	Instructional	Organization	Application Security Testing	Advise	Strategies and Use Cases
IG6 [169]	2020	Guide	Instructional	Organization	Application Security Testing	Advise	Strategies and Use Cases
IP1 [60]	2006	Presentation	Instructional	Company	Application Security Testing	Demonstrate	Strategies and Use Cases
IP2 [88]	2017	Presentation	Instructional	Company	Security Unit Testing	Demonstrate	Strategies and Use Cases
PA2 [140]	2013	Article	Perspective	Company	DevSecOps	Share Experiences	Strategies and Tools
PB2 [31]	2014	Blog	Perspective	Personal	Security Unit Testing	Demonstrate	Use Cases
PB3 [97]	2019	Blog	Perspective	Company	Developer Security Testing	Persuade to Adopt	Strategies and Tools
PB4 [208]	2017	Blog	Perspective	Personal	Security Unit Testing	Share Experiences	Strategies and Tools

6.1.1 Sources, Types, and Tones

This section describes the *formats* of the grey literature practitioners wrote to share their thoughts, recommendations, and experiences with security unit testing. The definitions and distributions for these attributes are shown in Table 6.2.

Table 6.2: Sources, Types, and Tones of Security Unit Testing Grey Literature

Source	**Count**	**Classification Definition**	**% of Total**
Company	27	Work for Profit	**52.94%**
Personal	12	Individual Representing Themselves	23.53%
Organization	7	Work for Common Goal	13.73%
Government	3	Owned or Funded by Govt.	5.88%
Media	2	Hosts Contributions from Multiple Authors	3.92%

Type	**Count**	**Classification Definition**	**% of Total**
Blog	24	Explicitly Labeled by Source	**47.06%**
Article	7	”Other”	13.73%
Documentation	6	Instructions for Using a Product from Owner	11.76%
Presentation	5	Slides or Video, Not a Lecture	9.80%
Book	3	Published and has ISBN	5.88%
Guide	3	Standalone Document, Explicitly Labeled by Source	5.88%
WhitePaper	3	Explicitly Labeled By Source, Industry, Marketing	5.88%

Tone	**Count**	**Classification Definition**	**% of Total**
Blended	24	Mix of Subjective & Objective Content	**47.06%**
Instructional	23	Objective, Actionable: ”how”, ”tutorial”, ”example”	45.10%
Perspective	4	Subjective, Opinions, Persuasive: ”should”, ”why”, ”experts”	7.84%

An important consideration for this analysis was the need to maintain the distinction between what practitioners *believe* and what they actually *do*. There is sometimes a disconnect between these two concepts, as evidenced by a prior study that compared empirical measurements of developer's (general) unit testing practices and their *perceptions* of these practices [26]. One of the benefits to using grey literature to study state of practice is that it is easy to seek out and compare insights for the same concept through both lenses in multiple contexts. For example, a developer may share their process for writing security unit tests in the form of a tutorial (*e.g.* `IB4` [93]), or an example test might be shown as part of a *Company's* persuasive argument for adopting the practice (*e.g.* `BB7` [100]).

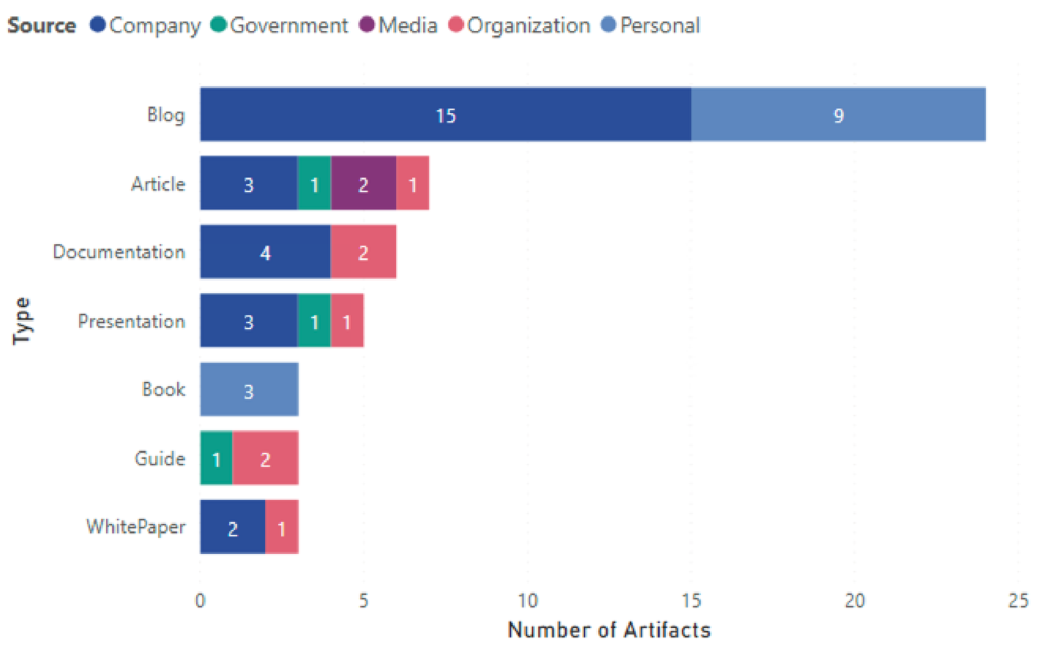


Figure 6.1: Distribution of Grey Literature by Type and Source

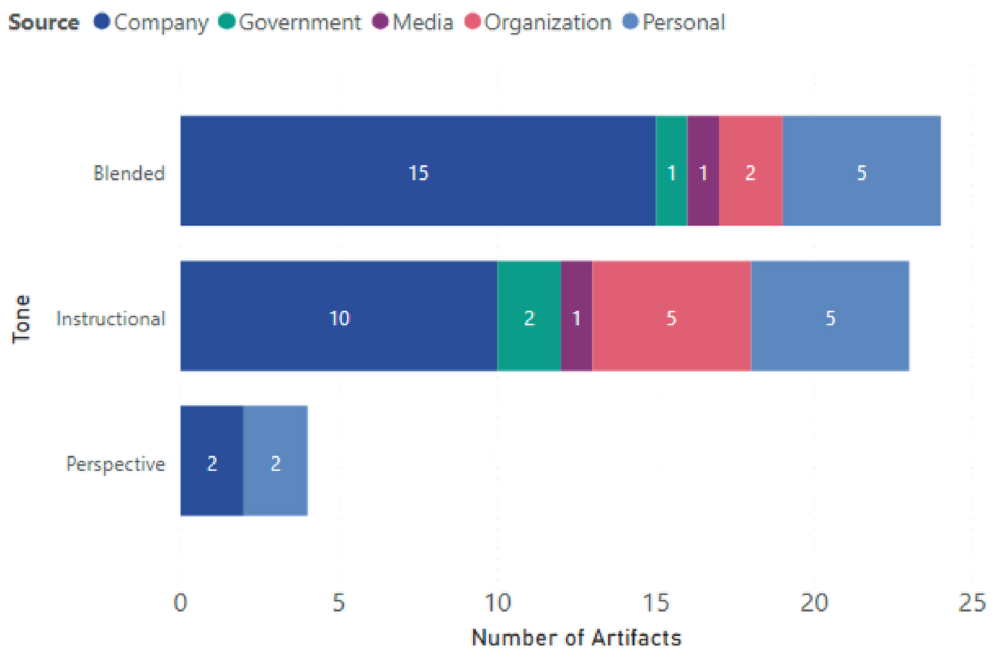


Figure 6.2: Distribution of Grey Literature by Tone and Source

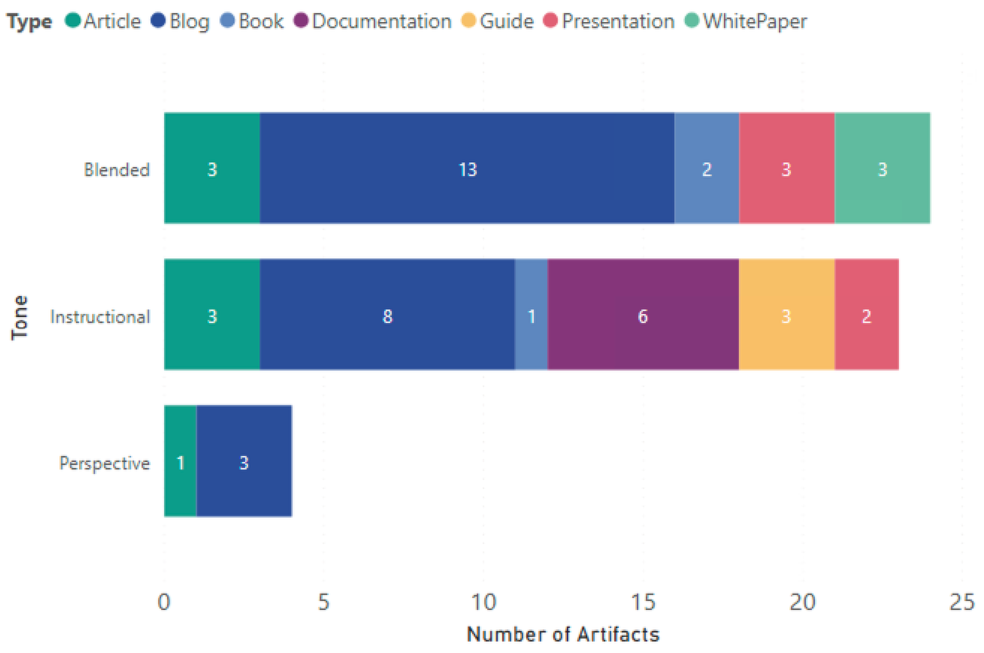


Figure 6.3: Distribution of Grey Literature by Tone and Type

Types and Sources of Security Unit Testing Grey Literature

Practitioners share perspectives and guidelines for security unit testing in the form of blogs, articles, documentation, presentations, books, guides, and white papers. Figure 6.1 shows the distribution of artifacts by type and source.

Almost half (47.06%) of the literature came from *Blogs* published by two sources: 62.50% were written by practitioners representing private *Companies*. The other nine blog posts were written by individuals (*Personal*) representing themselves. *Articles* and *Documentation* comprise 25.50% of the literature and also came predominantly from *Companies*. Three of the five *Presentations* were also given by practitioners representing *Companies*. None of the three *Books* or *Guides* found in the search came from this source, however. The *Guides* came from *Organizations* (OWASP) and in one case a *Government* agency, while all of the *Books* were authored by individuals. *Organizations* were the second most-common source of a variety of grey literature, producing at least one of every type except *Books*. The most source-diverse artifact type were *Articles*, a label used when published on a site not explicitly labeled as a *Blog*.

***Instructional*, *Perspective*, and *Blended*: Tones of Security Unit Testing GL**

The data one is able to extract from an artifact is dependent on its tone, which represents the objectivity or subjectivity of the artifact's content and ties back to distinguishing practice from perception. The three tones defined for this review were *Instructional*, *Perspective*, and *Blended*; Figure 6.3 shows the distribution of grey literature by tone and type and Figure 6.2 shows this distribution in terms of tone and source.

Subjective *Perspectives*: Practitioners shared their subjective or opinion-based *Perspectives* on security unit testing in four artifacts, labeled as such due to a complete focus on persuading the reader to do something mixed with a lack of actionable or sourced instruction. Three of these artifacts were *Blog* posts, two of which were produced by *Companies*. From these artifacts we can learn about practitioner's personal feelings and challenges in specific technical contexts, but do not consider content when discussing practices or guidelines.

Objective *Instruction*: Artifacts with this tone are written to inform the reader of security testing techniques or guide the reader through applying these techniques in practice. *Blogs* and *Documentation* make up about 60% of *Instructional* grey literature. *Documentation*, provided by *Companies* and *Organizations*, is naturally very technology-dependent and difficult to use in generalizations about the state of practice. Practitioners on the other hand used their *Personal Blogs* to share experiences and expertise with security unit testing in the form of tutorials, and one *Personal Book* provides guidance and example tests for functional security unit testing of firmware without making persuasive arguments for or against the practice. The analysis of existing guidelines and resources is based on these artifacts, but we do not examine them for opinions or challenges.

Blending Perspectives with Instruction: Most often, practitioner discussed security unit testing by using objective facts or data as support for their personal opinions or suggestions. These artifacts usually take a form similar to *"What is X and Why You Should Do X"*. About 47% of the grey literature discussing security unit testing uses this blended tone, so many of the insights into practitioner's perspectives about this practice are based on these artifacts.

6.1.2 Scopes, Purposes, and Content Types

Table 6.3 defines the scope, purpose, and content type attributes used to classify the artifacts and shows the percentage with each value. Figures 6.6 & 6.5 visualize these distributions in pairs.

Table 6.3: Scopes, Purposes, and Content Types of of Security Unit Testing Grey Literature

Scope	Count	Classification Definition	% of Total
Security Unit Testing	26	Only topic is security unit testing	**50.98%**
DevSecOps	16	Multiple topics re: shifting security into DevOps or CI/CD	31.37%
Developer Security Testing	5	Multiple topics re: develoment-phase security testing	9.80%
Application Security Testing	3	Multiple topics re: security testing throughout SDLC	5.88%
Risk-Based Security Testing	1	Multiple topics re: risk-based testing strategies	1.96%

Purpose	Count	Classification Definition	% of Total
Recommend	18	Lacks actionable guidance and/or includes subjective content	**35.29%**
Advise	12	Gives actionable guidance w/o subjective content	23.53%
Share Experiences	9	Describe how or what to do based on personal experiences	17.65%
Demonstrate	7	Show how to do [content] in a specific context	13.73%
Persuade to Adopt	5	Motivate reader to use [content] using obj. or sub. arguments	9.80%

Content Type	Count	Classification Definition	% of Total
Strategies and Tools	23	Describe best practices or tips and tools	**45.10%**
Tutorial	15	Step-by-step instructions for completing a specific task	29.41%
Use Cases	7	Contexts or subjects for utilizing a technique, strategy, or tool	13.73%
Strategies and Use Cases	5	Describe best practices or tips and contexts or subjects	9.80%
Tool Demo	1	Demonstration use of a tool to acheive a specific goal	1.96%

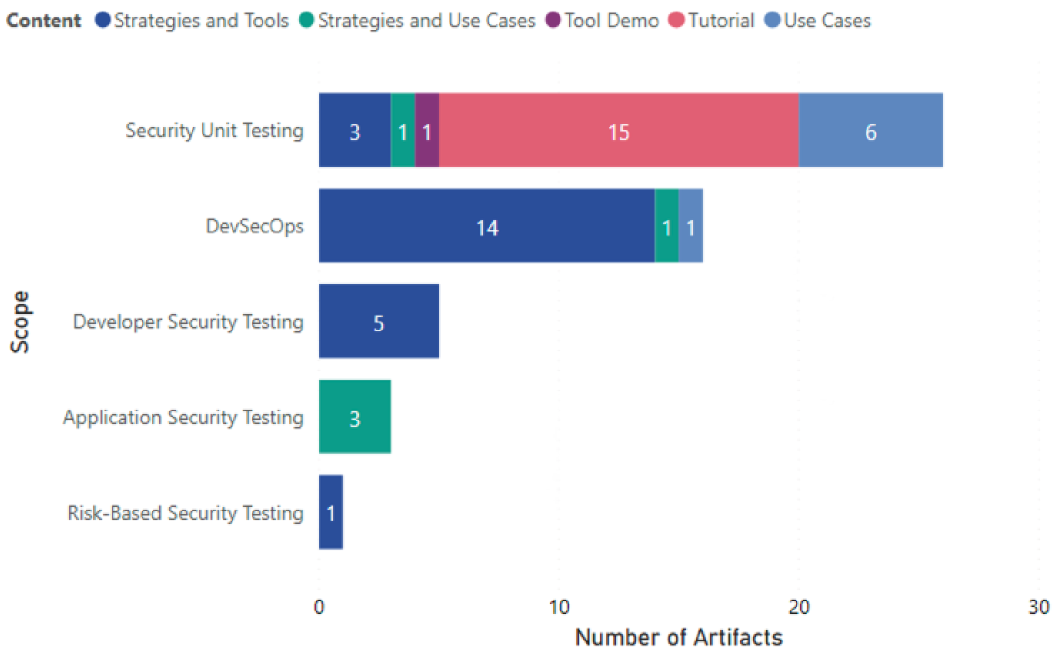


Figure 6.4: Distribution of Grey Literature by Scope and Content Type

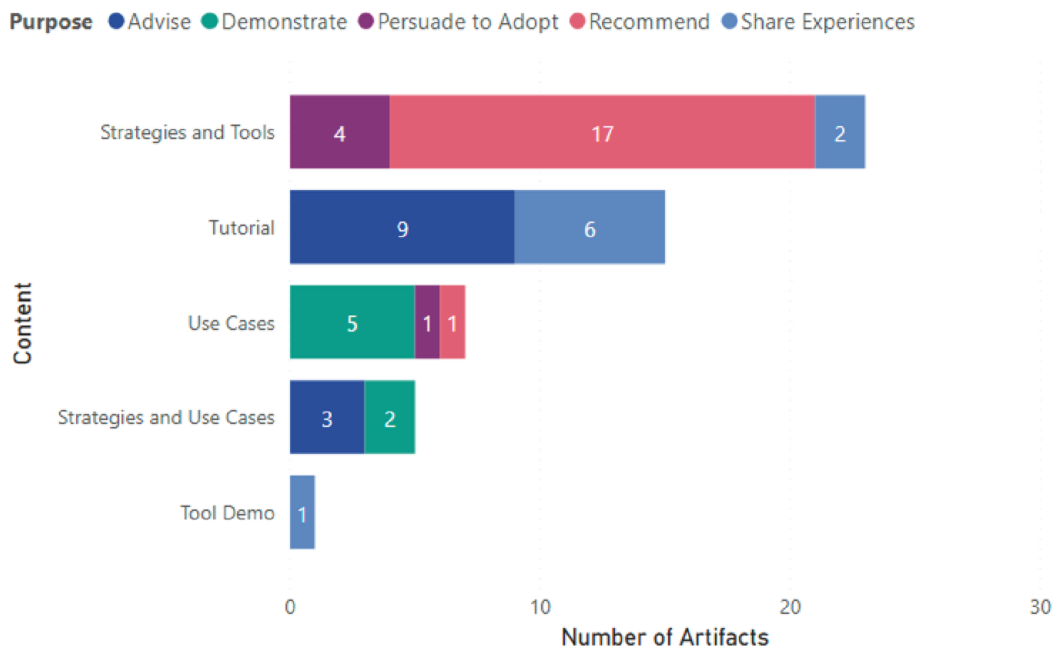


Figure 6.5: Distribution of Grey Literature by Content Type and Purpose

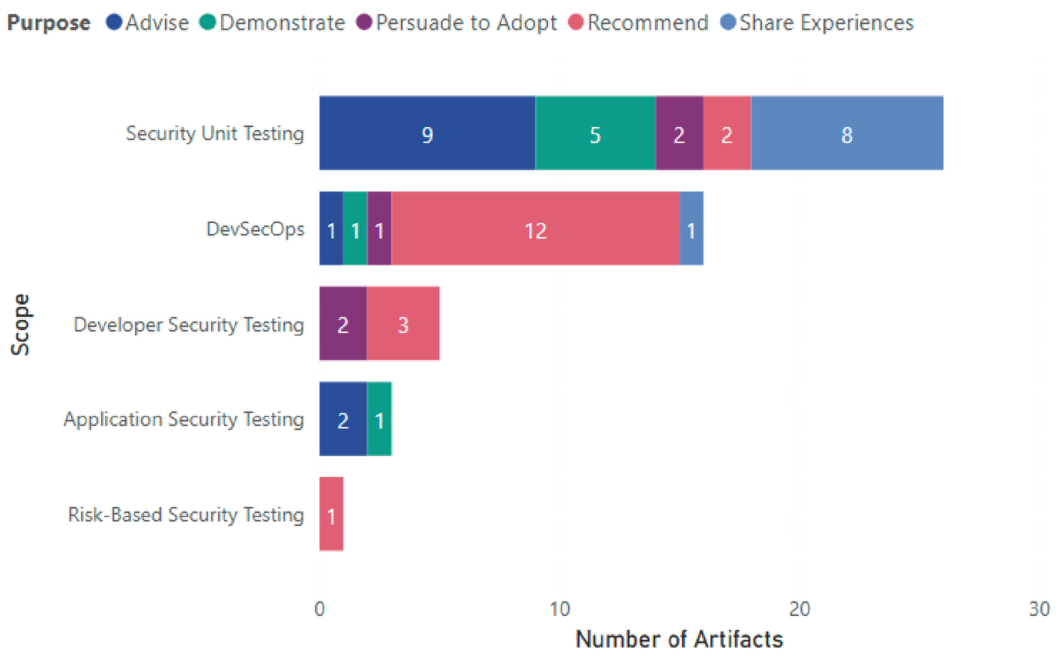


Figure 6.6: Distribution of Grey Literature by Scope and Purpose

Purposes: Practitioner's Aims When Writing Grey Literature

Without asking them (or unless explicitly written) it is not possible to definitively identify a practitioner's motivation for publishing grey literature. However, an artifact's *purpose*, or in other words what the practitioner intends to achieve by writing it, can be inferred by considering its contents in conjunction with the attributes discussed prior. The literature reviewed was written to *Advise*, *Demonstrate*, *Persuade to Adopt*, *Recommend*, and *Share Experiences*.

The most common artifact purpose (35.29%) was to *Recommend* some type of content to the reader. The majority (72.22%) of these artifacts had a *Blended* tone, and the others were *Instructional*. The 23.53% of grey literature meant to *Advise* readers all have *Instructional* tones. The distinction between these purposes is how actionable the content is and whether subjective content is included; for example, a practitioner can *Recommend* a set of security testing tools based on personal opinions of their benefits (*e.g.* `BB17` [198]) or *Advise* readers how to *use* a tool (*e.g.* `IA2` [92]).

Practitioners also wrote to *Share Experiences* — for example `IB4` [93] is a tutorial based on the author's personal experience writing unit tests for a specific authorization attribute. Similar to the previous two purposes, practitioners also create grey literature to *Demonstrate* a use case for a specific type of content. For example, `BB7` [100] and `PB2` [31] were written to demonstrate how security unit testing could have prevented previous security exploits in a specific context. Finally, almost 10% of the grey literature was written to *Persuade* (readers) *to Adopt* something. In these artifacts, practitioners explicitly express their desire to convince the reader to take some action, with or without actionable suggestions for doing so or evidence for the benefits.

Scopes and Types of Content in Security Unit Testing Grey Literature
Practitioners shared five types or combinations of content in the grey literature reviewed: *Strategies and Tools* (45.10%), *Tutorials* (29.41%), *Use Cases* (45.10%), *Strategies and Use Cases* (9.80%), and one artifact was a *Tool Demo*. All of the grey literature studied is related to security unit testing, and 50.98% of the artifacts only discuss this topic. Practitioners seek to *Advise* or *Share Experiences* with security unit testing in these artifacts, usually in the form of a *Tutorial*. The remaining grey literature have larger scopes, where security unit testing is one of several topics of discussion. The main subject of 31.37% of the grey literature was *DevSecOps*; practitioners often *Recommend* security unit testing as one of multiple *Strategies and Tools* for integrating security into DevOps workflows. Practitioners also share perspectives on security unit testing when discussing *Strategies*, *Tools*, and *Use Cases* for *Developer Security Testing*, *Application Security Testing*, and in one case, *Risk-Based Security Testing*.

In the following sections, I report the results of the qualitative analysis of these grey literature. An description of the process used to synthesize these insights is given in Chapter 3.3. Section 6.2 focuses on practitioner's perspectives whereas Section 6.3 is a thorough examination of the actionable *Instruction* and *Advice* regarding security unit testing in existing guidelines.

6.2 Perspectives on Security Unit Testing

This section focuses on insights that emerged from analyzing literature with *Blended* and *Perspective* tones, *i.e.* those artifacts written to share personal beliefs and feelings or to provide lightweight recommendations. Figure 6.7 summarizes these perspectives.

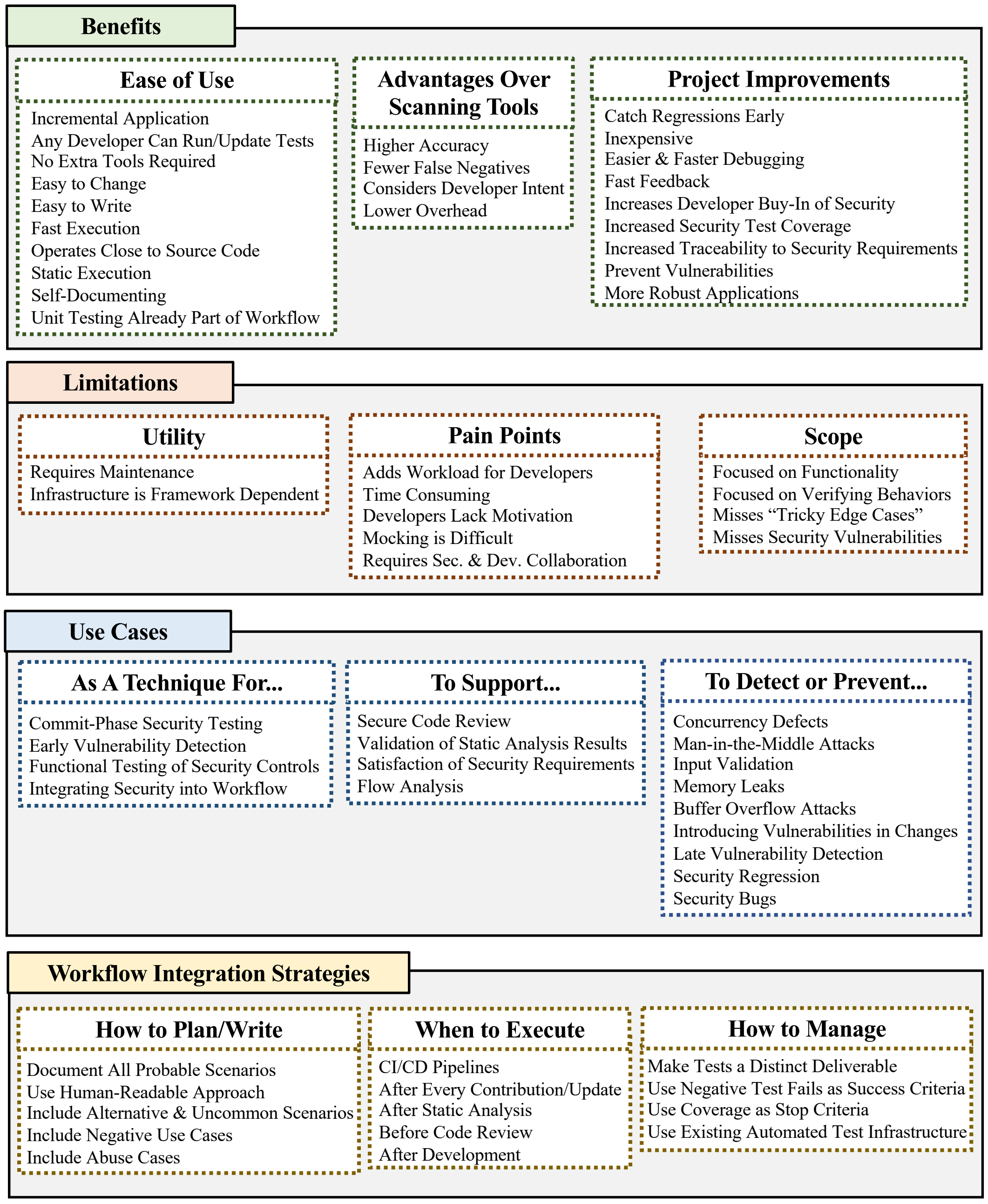

Benefits
Ease of Use
Incremental Application
Any Developer Can Run/Update Tests
No Extra Tools Required
Easy to Change
Easy to Write
Fast Execution
Operates Close to Source Code
Static Execution
Self-Documenting
Unit Testing Already Part of Workflow
Advantages Over Scanning Tools
Higher Accuracy
Fewer False Negatives
Considers Developer Intent
Lower Overhead
Project Improvements
Catch Regressions Early
Inexpensive
Easier & Faster Debugging
Fast Feedback
Increases Developer Buy-In of Security
Increased Security Test Coverage
Increased Traceability to Security Requirements
Prevent Vulnerabilities
More Robust Applications
Limitations
Utility
Requires Maintenance
Infrastructure is Framework Dependent
Pain Points
Adds Workload for Developers
Time Consuming
Developers Lack Motivation
Mocking is Difficult
Requires Sec. & Dev. Collaboration
Scope
Focused on Functionality
Focused on Verifying Behaviors
Misses “Tricky Edge Cases”
Misses Security Vulnerabilities
Use Cases
As A Technique For...
Commit-Phase Security Testing
Early Vulnerability Detection
Functional Testing of Security Controls
Integrating Security into Workflow
To Support...
Secure Code Review
Validation of Static Analysis Results
Satisfaction of Security Requirements
Flow Analysis
To Detect or Prevent...
Concurrency Defects
Man-in-the-Middle Attacks
Input Validation
Memory Leaks
Buffer Overflow Attacks
Introducing Vulnerabilities in Changes
Late Vulnerability Detection
Security Regression
Security Bugs
Workflow Integration Strategies
How to Plan/Write
Document All Probable Scenarios
Use Human-Readable Approach
Include Alternative & Uncommon Scenarios
Include Negative Use Cases
Include Abuse Cases
When to Execute
CI/CD Pipelines
After Every Contribution/Update
After Static Analysis
Before Code Review
After Development
How to Manage
Make Tests a Distinct Deliverable
Use Negative Test Fails as Success Criteria
Use Coverage as Stop Criteria
Use Existing Automated Test Infrastructure

Figure 6.7: Practitioner Perspectives on Security Unit Testing

Although evidence from all three studies in this work suggest that security unit testing is not widely-adopted, practitioners regard this practice positively in their writing and perceive it as a technique well-suited for shifting security testing (further) left in the development lifecycle. The rationales given for these perspectives enumerate or demonstrate its benefits, limitations, and use cases. Also shared were recommendations for developer-centric strategies to integrating this practice into automated testing workflows.

Practitioners praised and recommended security unit testing to their peers, sharing a number of **benefits** to adopting the practice. They highlight it's *Advantages Over Scanning Tools*, describe attributes contributing to its *Ease of Use*, and give examples of *Project Improvements* that can stem from use of this technique. They seldom shared what they perceived as the **limitations**, however those mentioned are related to the *Utility*, *Scope*, and perceived *Pain Points* of security unit testing. Practitioners also supplement their recommendations and persuasive arguments for security unit testing with descriptions of appropriate **use cases** and **strategies** for integration into developer workflows. They suggested three types of use case: *As A Technique For..* achieving select security goals, *To Support...* other security activities or efforts, and *To Detect or Prevent...* nine types of functional or vulnerability-based concerns. Their strategic perspectives for workflow integration concentrate on *How to Plan/Write* tests, *How to Manage* security unit test suites, and *When to Execute* the tests. These perspectives are presented independently in Figure 6.7, but in the remainder of this section I describe them in a blended narrative to better summarize them and preserve relationships between perspectives and rationales.

6.2.1 When, How, & Why Practitioners Want You to Write Security Unit Tests

Practitioners share their perspectives on security unit testing in the context of strategies for shifting security "left" into the development phase, usually towards securing continuous workflows *i.e.* moving from *DevOps* to *DevSecOps*. A few artifacts begin by acknowledging that most attention in this phase is focused on automated static or dynamic analysis testing tools (SAST/-DAST), which scan code to identify violations of secure coding standards or known vulnerabilities (*e.g.* BB4 [45], BA4 [132], BB13 [164]). However, security unit testing is never suggested as a replacement for these tools, regardless of whether its "underdog" status is acknowledged. Instead, **security unit testing is perceived as a complement to scanning tools**; one practitioner cited this when contemplating how unit tests could have prevented a past exploit:

> *"Static analysis tools are getting better at detecting duplicate code, but* ***programmers writing unit tests are still the first and most effective line of defense.****"* (PB2 [31])

The context of these discussions is also reflected in the *Use Cases* suggested for security unit testing in that they **recommend and demonstrate how this technique can be used as a means to shift security left**. They don't always express this so explicitly, but mention related activities *e.g.* functional testing of security controls, integrate security into existing workflows, and commit-phase security testing. Other use cases take the form of specific vulnerabilities and concerns that are well-suited for testing at the unit level. Many of the *Blended* artifacts, which mix technical content with subjective comments, focus on these examples. However, suggesting that input validation can be unit tested is not the same as providing actionable guidelines for actually writing such tests.

What practitioners consider as the *Benefits* of security testing are mostly the same or similar to those given for automated unit testing in general [184, 194]. In short, **practitioner's main argument in favor of security unit testing is that it is an *adaptation* of already-standard practices and infrastructure, whose benefits can be inherited to improve the security posture of a project with minimal disruption to developer's current workflows**.
As one practitioner put it,

> *"The first tactic* [for building a DevSecOps pipeline] *is weaponizing your unit tests;* ***you already have them, so why not reuse them?"*** (`BB8` [102])

Some unique benefits emerged as well, particularly when discussing how use of security unit testing can lead to *Project Improvements*. For example, in `BW2` [49] the author supports their argument that *"security can, and should, be integrated into unit, integration, and acceptance testing"* by claiming that doing so results in *"A shorter security testing phase"*, *"More robust applications..."*, and **increased *"Developer buy-in to the security process"***.

Following the thread of security unit testing as a ready-made technique, the *Workflow Integration Strategies* proposed in this literature are based on *adapting* existing practices and concepts. Practitioners highlight that, by using automated unit testing infrastructure for security testing, these tests can be run repeatedly (*continuously*, even) in the existing CI/CD pipelines after every update and even post-development. Of particular interest are the suggestions for planning and writing security unit tests. Across the literature, practitioners suggest leveraging the closer relationship between security and development teams in DevOps workflows to support developers identifying *what* to test, which is noted as a barrier to security unit testing (`BB5` [54]). Several practitioners shared the perspective that developers are familiar with "happy path" test cases and using techniques like equivalence partitioning to plan tests. This highlights a key sociotechnical factor that should be considered when providing adaptation-based guidelines. In fact, practitioners (*e.g.* `BA1` [47], `BP1` [104]) and experts (*e.g.* `IG6` [169], `IB2` [34]) use this sentiment as a rationale for suggesting that helping developers devise negative, alternative, and abuse cases is a critical part of adapting this practice. This notion of developer's focus on positive scenarios and the need for collaboration with security teams have also been given as reasons why adoption is currently low (*e.g.* in `BP1` [104]).

As a few practitioners acknowledged, adaptations can also suffer from the same limitations. Most of the limitations were used to support the use of automated unit testing *in conjunction* with SAST/DAST to shift security left. One artifact in particular (`PB3` [97]) championed this perspective, although three other practitioners (`BB15` [181], `BP1` [104], `BP2` [113]) mention them in a more off-hand way. These practitioners noted that *Utility* is limited in the sense that unit tests require maintenance (`BP1` [104]) and automated testing frameworks are language and framework dependent (`BP2` [113]).

Personal opinions about developer behavior were sometimes mixed with more objective (or at least, measurable) comments about pain points in these arguments. For example:

> *"**Most developers agree that unit testing is a good idea but they don't tend to do it**. They often **don't have the time or motivation** to do so and manual unit testing is **time consuming and tedious**. Also, unit testing requires **stubbing and mocking** which is an added workload. Unit tests are themselves code and **add to the maintenance workload** since changes in the unit under test can mean changes to the unit test code."* (`PB3` [97])

Prior work studying human factors of unit testing (Chapter 2.4) support my claim that most of the *Pain Points* mentioned are also inherited. However, in the security context developers may require extra support not provided by those developed to alleviate general unit testing challenges. I discuss this in-depth in the next Chapter, but note here that the difficulty of mocking security-related fixtures expressed in these artifacts was a common theme in security unit testing pain points that emerged from analysis of Q&A posts. Supporting developers in identifying and mitigating these challenges will require additional research and workflow adaptations; see Chapter 7 for recommendations. The *Scope* limitations mentioned were vague and not well supported; I report them but also wish to highlight that many of the strategies given for workflow integration (*e.g.* including negative and abuse cases in test plans) serve to mitigate them. Also, as demonstrated by Mohammadi *et al.* [128] and Salva and Regainia [171] unit tests *can* be used for security vulnerability testing. In fact, while the grey literature promoted both functional and vulnerability tests, the latter was mentioned most often.

In the next section, I will describe how security unit testing is represented (or not) in the objective, instructional security testing material curated by practitioners and expert groups for developers.

6.3 Instruction and Guidelines for Security Unit Testing

Grey literature in which practitioners shared objective or evidence-based guidelines, instructions, and integration strategies without interjecting personal opinions or experiences were analyzed to understand in greater detail the breadth and depth of actionable support currently available. Many of the suggestions and recommendations provided in the artifacts discussed prior are mirrored in these literature — I reiterate that they are discussed separately to maintain distinction between opinion and practice as well as identifying concrete actionable guidance.

6.3.1 In Formal Security Testing Guides

We identified only three formal (curated and maintained by experts and organizations) security testing guides that explicitly discuss security unit testing. **These guides enumerate abstract functional and vulnerability security test cases but do not demonstrate or explain how to write tests**. The OWASP Application Security Verification Standard (ASVS) [44] (`IG3` [44]), a respected resource for security requirements and test cases, says the following about using it as a guide for security unit testing:

> *"By building **unit and integration tests that test for specific and relevant fuzz and abuse cases**, the application becomes nearly **self-verifying** with each and every build. For example, additional tests can be crafted for the test suite for a login controller, testing the username parameter for common default usernames, account enumeration, brute forcing, LDAP and SQL injection, and XSS. Similarly, a test on the password parameter should include common passwords, password length, null byte injection, removing the parameter, XSS, and more."* (`IG3`, [44, pg.13])

The suggestions given in this statement are a mix of functional and vulnerability test cases. The authors of this guide also highlight functional testing concerns that are *only* possible to automate at the unit and integration level:

> *"**Business logic flaws** and **access control** testing is only possible using human assistance. These should be turned into unit and integration tests."* (`IG3`, [44, pg.10])

This is the only place where security unit testing is mentioned explicitly; the test cases/requirements that comprise the remainder of this guide are provided in natural language. This is without question a valuable resource for security testing at any level, including unit and integration tests. However, it is left to the developer to determine how such tests should be designed and implemented as no examples or concrete guidance is provided.

`IG6` is another resource from OWASP, the Web Security Testing Guide (WSTG) [169]; this guide also mentions security unit testing in its introductory material, suggesting developers use it to *"ensure they are producing secure code"* as part of *"normal code and unit testing procedures"* [169, pg.6]. Unlike the previous guide, however, an entire page is devoted to discussing strategies for security unit test planning and workflow integration.
Mirroring perspectives shared by individual practitioners, this guide *Advises* the following:

> "*A* ***good practice for developers is to build security test cases as a generic security test suite that is part of the existing unit testing framework****. A generic security test suite could be* ***derived from previously defined use and misuse cases*** *to* ***security test functions, methods and classes.***" (`IG6`, [169, pg.30])

The other security unit testing-specific strategies in `IG6` [169] relate to verifying adherence to secure coding guidelines, in associated with vulnerability testing. The test cases in this guide are also presented in natural language without guidelines or examples for translation to unit test format.

6.3.2 In Informal Guidelines from Practitioners

Section 6.2 described the *Perspective* (subjective) and the less-objective *Blended*-toned literature practitioners shared to *Recommend* and *Persuade* readers to adopt *Strategies and Tools* for shift-left security. These artifacts provided only high-level discussion and explanation of their subjects. However, practitioners also publish resources with *Instructional* or *Blended* (objective-subjective mix) tones whose *Content Types* are primarily *Tutorials* and *Use Cases.* The target audience for these resources are individual developers, to whom they *Advise*, *Demonstrate*, and *Share Experiences* with writing security unit tests. The topics covered in these informal guidelines are consistent with those developers most commonly discussed in security unit testing-related Q&A posts (Chapter 4.1): *Authentication*, *Protected Resources*, *Login*, *Encryption*, *Client/Server*, and *HTTP Requests.*

These resources most commonly took the form of *Tutorials*, which are the more actionable but less generalizable than other guidelines due to their technology and context-dependent scopes. *Companies* and *Organizations* that produce APIs and frameworks sometimes include *Tutorials* in official *Documentation*; for example, `ID1` [66] shows how to set up fixtures for unit and integration tests involving components of the Java security framework Apache Shiro. In *Personal Tutorials*, individual practitioners show their peers how they construct functional and vulnerability-detecting security unit tests.

6.4 Conclusion

To understand how practitioners perceive security unit testing and examine available resources we conducted a rigorous, we conducted a qualitative analysis of 51 grey literature in which security unit testing is explicitly discussed. As part of this analysis, we characterized these artifacts to understand the tone, purpose, *etc.* in which they were written.

Although analyzed separately, upon review we found that many of the use cases and workflow integration strategies that practitioners shared in blog posts, articles, *etc.* were also suggested in expert-curated guides and other *Instructional* artifacts. Across all categories of literature reviewed much of the discourse and recommendations shared were unhelpfully abstract, failing to provide or direct readers to resources with specific steps that can be taken to realize the activities suggested. The exception to this are the informal guidelines meant to *Advise*, *Demonstrate*, and *Share Experiences* individual developers through actionable *Tutorials* and *Use Cases*. However, these artifacts focus on test design and construction without much attention to planning and are are heavily dependent on technological context, making them less generalizable.

With this study we contribute empirical evidence of practitioners' enthusiasm for more widespread adoption of security unit testing, especially in DevSecOps and continuous security contexts. Admittedly, relative to the large volume of literature discussing secure development and automated testing, the number of artifacts that explicitly discuss the intersection of these practices is limited. This is, however, a reflection of the ad-hoc adoption observed in my *Practices* study and the overall obscurity of this technique. Nonetheless, examining these literature allowed us to learn practitioner's security unit testing beliefs, opinions, recommendations, and experiences *in their own words*. This data is a valuable baseline for future empirical studies and these perspectives form the basis of the integration strategies and recommendations to promote I provide in Chapter 7.

6.5 Study Limitations

This section discusses the limitations of using grey literature to study perspectives and resources, including threats to validity and our efforts to mitigate them.

We identified 51 artifacts that directly and explicitly refer to security unit testing. This small sample size, relative to the body of literature available for security and unit testing in general, and the high variability of quality, content, and format in grey literature [71] could affect how representative the findings are (external validity) to general practice. There is also the threat of selection bias in which some valid literature may have been missed in the search process. To mitigate these threats we followed guidelines prescribed for grey literature reviews in software engineering [70, 71, 156, 157]: research questions and exclusion criteria were established prior to the search, and as stopping criteria we used theoretical saturation (no new concepts emerged) and when conducting the search. Further, in these artifacts practitioners discussed security unit testing in a wide variety of contexts as described in Section 6.1.1. We are therefore confident that this collection is sufficiently representative of practitioner's *publicly-shared* perspectives on the subject of security unit testing, even if some literature was somehow missed by our search parameters.

Following this line of thought, construct validity could be threatened by our choice to use grey literature to study perspectives rather than other common methods *i.e.* interviews and surveys. This was a deliberate decision made to make a novel contribution to existing empirical evidence of security and testing perspectives. As discussed in Chapters 2.4 and 2.4, there are a large number of interview and survey-based academic studies on developer's perspectives on security and testing. As security unit testing lies at the intersection of these concepts many of these study's findings also apply. Grey literature review is recommended in empirical software engineering guidelines as a equally valid technique for collecting perspectives and insights on poorly understood and under-studied concepts, although care must be taken to avoid biases or misrepresentation [71].

Figure 6.8 from Garousi *et al.*'s guidelines for using GL in software engineering research [70], lists conditions where inclusion is appropriate; **this work meets conditions 2, 3, 6, and 7.**

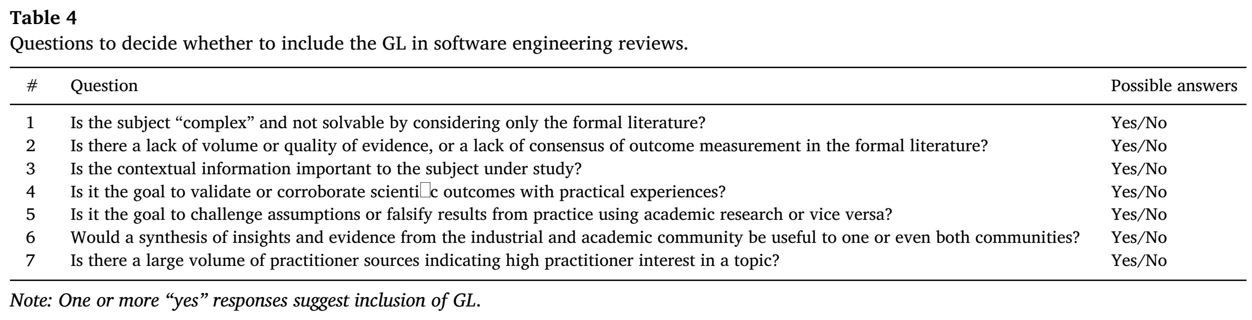


Table 4
Questions to decide whether to include the GL in software engineering reviews.

#	Question	Possible answers
1	Is the subject "complex" and not solvable by considering only the formal literature?	Yes/No
2	Is there a lack of volume or quality of evidence, or a lack of consensus of outcome measurement in the formal literature?	Yes/No
3	Is the contextual information important to the subject under study?	Yes/No
4	Is it the goal to validate or corroborate scienti□c outcomes with practical experiences?	Yes/No
5	Is it the goal to challenge assumptions or falsify results from practice using academic research or vice versa?	Yes/No
6	Would a synthesis of insights and evidence from the industrial and academic community be useful to one or even both communities?	Yes/No
7	Is there a large volume of practitioner sources indicating high practitioner interest in a topic?	Yes/No

Note: One or more "yes" responses suggest inclusion of GL.

Figure 6.8: Garousi *et al.*'s Criteria for Including Grey Literature in SE Reviews [70]

Future work, however, should include interview or survey studies to to evaluate interventions or support systems that stem from the findings of this study and to corroborate with *private perspectives* not shared in grey literature.

The internal validity of the work may be threatened by the analysis process used to identify and interpret concepts from the literature. To mitigate this, we followed prior work in using open and axial coding techniques [39,70,158], which is well-suited for exploring poorly understood phenomena and ensures findings are grounded in, and can be linked directly to, the data. Myself and another researcher independently analyzed the literature and applied codes as they emerged, guided by a broad set of categories to ensure the insights needed to answer the research questions were collected. We then conducted two rounds of review and agreement discussion to ensure that our interpretations were consistent and valid.

Chapter 7

Discussion

In this section I summarize the state of practice for security unit testing, and make recommendations for supporting and empowering developer's efforts to integrate this technique into their workflows. This includes immediately actionable strategies as well as a roadmap for future research.

7.1 The State of Practice for Security Unit Testing

It is clear that **security unit testing is, in practice, an ad-hoc and informal activity**. Relative to what is available for security testing and automated testing at-large, artifacts that explicitly discuss this technique are sparse. This could in part be explained by the omission of security unit testing as a formal activity in secure process models and frameworks like the Microsoft Secure Development Lifecycle (SDL) that practitioners use to integrate security across the software development lifecycle. In its place are recommendations for vulnerability-focused automated static and dynamic scanning tools, and as a result these models and other guidelines fail to encourage practitioners to write security tests during unit and integration testing.

Practitioners hail security unit testing as a means to reduce workflow interruptions in shift-left efforts because the infrastructure and procedures are already part of continuous development pipelines. In their writing, they outline benefits, limitations, use cases, and suggest workflow integration strategies in the context of DevSecOps and continuous security. Many of their persuasive arguments are based on benefits that are *inherited* from the general practice of automated unit testing, *e.g.* speed, low cost, easier debugging, earlier defect detection *etc.*. Both subjective and objective literature focus more on vulnerability testing than functional security control testing although both are discussed and recommended.

Our analysis of real security unit tests and Q&A posts are evidence that developers do write these tests, even if they do so in an ad-hoc manner or without a security testing mindset. From all artifacts collected, we observed a common set of security controls, components, and scenarios as subjects-under-test, with **authentication being the primary control tested at this level**. Developers experience **seven pain points *unique* to writing security-related unit tests** associated with these test subjects, and we found that even generic unit testing tasks like *Creating HTTP Request Objects* and *Creating User Fixtures* can be unexpectedly difficult due to the unique properties of security control implementations.

A fair number of guidelines, which we consider any artifact that provides actionable and/or objective content, support or recommend security unit testing. We compared the *Content Types* and *Purposes* of *formal* guides (*e.g.* `IG3`, [44]) that are curated and maintained by authorities and expert groups, and *informal* guides (*e.g.* `IB1` [7]) produced by individuals). Formal guidelines provide abstract workflow integration strategies, context-agnostic requirements, and generic test cases making them a good resource for team-level activities like security requirements, design, and test planning. Informal guidelines, usually written as *Tutorials*, provide developers with narrowly-scoped but actionable instructions for designing and writing security unit tests. Although some *Companies* include fixture setup *Tutorials* for unit testing within their security API *Documentation* (*e.g.* `ID1` [66]), it does not appear sufficient to help developers mitigate associated pain points. This is consistent with prior work examining the contents of developer-oriented security resources [5] and the usability of cryptography APIs [1,136], which recommend security API owners provide more task and example-based documentation.

Neither formal or informal guidelines provide concrete support to help developers implement unit test methods for security requirements or abstract test cases. Unit testing is unique to other types of software testing in that developers *implement* these tests as executable methods using xUnit frameworks and the system's own source code. To write tests for other functional requirements, developers identify their underlying scenarios, actions, and expected outcomes (see Chapter 2.4), which can be traced back to the code. However, this familiar practice can be challenging to adapt for security requirements without adequate security expertise or guidance. Consider, for example, this test case for weak passwords that is recommended in the OWASP Web Security Testing Guide `IG6` [169, pg.159]: *"Is the user prevented from using his username or other account information (such as first or last name) in the password?"* — to write a unit test for this case, the developer must map this description to the preconditions (scenario), code-under-test (action), and the appropriate behavior (expected outcome) as they are represented in the source code. To implement security controls, developers rely heavily on third-party frameworks and APIs that can exacerbate the obscurity and complexity of a control's inherent design and make this process more difficult than it is for other functionality. This problem motivated us to leverage our taxonomy of authentication test cases (Chapter 4.1) to produce a unit testing guide formatted to relieve developers of this cognitive burden [82].

7.2 Recommendations

The following recommendations are, in part, a culmination of strategies, guidelines, recommendations *etc.* practitioners have shared over the last decade which until now, have not been compiled in a formal matter. I supplement these with novel insights that emerged from analyzing by-products of security unit testing efforts, *i.e.* test code and Q&A posts. The following are steps that practitioners can take *immediately* to begin integrating this technique into their workflows and promote wider adoption. In the next section I discuss future work towards developing support systems and pain point mitigations that would further empower developers in writing security unit tests.

7.2.1 Who Should Follow This Advice?

In an ideal world security unit testing would be a part of *every* development project. However these recommendations are given under non-ideal conditions as it can not be said that that *every* software project meets the prerequisites, *i.e.* (1) uses a secure development model and (2) has made automated unit testing a formal part of the development workflow. It follows that projects without these practices in place likely are not or could not be motivated to adopt security unit testing.

The teams in the best position to adopt security unit testing *today* are those that follow DevOps, DevSecOps, or continuous testing models and have *already committed* to shift security into their development workflows.

I make the distinction about prior commitment to emphasize that motivating teams to shift security left in general is out of scope for this work. **As long as these conditions are met, this technique can be adopted regardless of a team or project's current security posture.** They will already have the automated testing infrastructure in place, so learning how to design and write security-related tests would truly be an adaptation of existing workflows. Introducing a testing framework at the same time would pose a greater interruption to developer's current practices, which can negatively impact productivity [36]. Practitioner's enthusiasm for shifting security left, especially in continuous workflows, is evident from the recent "boom" in activity around the development of frameworks, guidelines, tools, *etc.* for DevSecOps. It is therefore reasonable to assert that **now is the ideal time to "close the gap" between perception and practice for security unit testing** by introducing it through these initiatives.

In the next section I discuss how the secure development community can promote and support more formal adoption of security unit testing in this context and recommend workflow integration strategies for software organizations and teams.

7.2.2 Strategies for Formalization and Workflow Integration

The findings of my work suggests that the choice to write security unit tests is an ad hoc decision made by individual developers rather than a formal one made at the team level. Developers may be demotivated to adopt security practices by a lack of organizational and process support [14], so formalization is a key element of my strategy for integration and promotion. The following actions can be take immediately to address this at the community and team levels.

Add Security Unit Testing to SSDLC Frameworks, Especially for DevSecOps

Security unit testing is praised by practitioners and listed as a recommended practice in many in expert-curated test guides. However, it has been consistently omitted from secure software development lifecycle (SSDLC) frameworks and models produced by respected authorities such as the Microsoft SDL [99], OWASP SAMM [55], and the NIST SSDF [57]. These resources outline security activities for each phase of the development lifecycle and help teams track their adoption progress by mapping these actions to maturity levels. Scoping to abstract lifecycle phases and keeping activity suggestions broad (*e.g.* "use scanning tools during development") facilitates more widespread applicability. However, prior studies on DevOps adoption, namely the seminal work of Forsgren *et al.* [64, 65], provide evidence that practitioners also value support provided by context-aware models and frameworks for evaluating progress and measuring effectiveness of new practices.

To better leverage and integrate with the unique infrastructure and workflows inherent of DevOps, these authoritative groups have begun development of *DevSecOps frameworks*. Unfortunately, security unit testing is not presently a recommended activity in the Microsoft Secure DevOps framework [1], the NIST DevSecOps framework [2], or OWASP's DevSecOps Guidelines [3] and Maturity Model [4]. Given the reasonable emphasis on techniques for vulnerability detection and prevention in secure development discourse, it makes sense that these frameworks recommend SAST and DAST

[1]https://www.microsoft.com/en-us/securityengineering/devsecops

[2]https://csrc.nist.gov/Projects/devsecops

[3]https://owasp.org/www-project-devsecops-guideline/

[4]https://owasp.org/www-project-devsecops-maturity-model/

tools over security unit testing. Although vulnerability prevention is one of the recommended use cases for security unit testing in grey literature and the focus of academic efforts in this area [129, 171] it appears that many practitioners have not considered unit tests for non-functional purposes. This is reflected in these framework's failure to formally encourage functional security testing, for which security unit testing is *uniquely* well-suited. Therefore, **I recommend adding security unit testing to DevSecOps frameworks as a *functional* security testing activity to formally motivate and support adoption**.

Empower Your Developers to Apply a "Security Mindset" to Unit Testing

The primary benefit and motivator for this technique as cited by practitioners is that it can be conducted by adapting the existing infrastructure used for automated testing. Accordingly **workflow integration strategies for security unit testing are based on helping developers write and design tests with a *security mindset***. However, the burden of obtaining and maintain expert-level security knowledge in addition to their other priorities should not be placed solely or immediately on developers. This can pose a challenge to shift-left efforts, as teams must find an appropriate strategy to reconcile this with the desire to include security-related activities in developer's workflows. The key to solving this problem is to **Avoid placing the burden of security expertise *immediately or solely* on developers by empowering them to develop these skills over time**. Teams and organizations should provide training, encourage collaboration with security champions, and engage developers in other security activities *e.g.* threat modeling.

A common perspective amongst practitioners is that developers tend to plan tests only for "happy paths"; to adapt this practice, these efforts should **focus on helping developers identify negative, alternative, abuse, misuse, *etc.* cases they should test for security controls**. This is applicable for testing both functional security scenarios (*e.g.* login with invalid token) and vulnerabilities (*e.g.* buffer overflow). However, I recommend teams who already apply static or dynamic analysis (SAST/DAST) concentrate on supporting functional security test planning at the unit level — these tools are better suited for vulnerability testing because they can look for the same pattern across the entire system, whereas a unit test would have to be written for each associated component. The pain point analysis in Chapter 5 also indicates that developers sometimes write unit tests involving security controls outside of a security context, for example when testing protected resources. Hence, another approach to consider is one that would help developers design and write security-related unit tests regardless of motivation. As a complement to my previous recommendation, I suggest that **all test cases that will evaluate *or include* security-related code should be explicitly identified and documented during test planning.** This activity would force the team to *actively* and *preemptively* think about which security-related classes and components will be involved, and reveal points of uncertainty or potential complexity. Security liaisons can then help developers design the appropriate fixtures and test environments. **Over time, teams can leverage the products of these collaborations to create reusable and project-specific test plans, security test suites and fixtures classes**. Conducting these activities as a team is a step towards motivating developers with organizational support; making these resources available to developers *and* involving them in the creation process are further steps towards empowering developers with the expertise needed for independent efforts. I do acknowledge that security resource and personnel availability will vary across teams and organizations, and that this type of communication can be difficult to acheive at-scale [29]. To truly follow DevSecOps principles a *security champion* [11, 25, 29, 185, 206] should be identified and made available to developers but in any case **independent *and* collaborative test planning efforts can leverage external references such as the OWASP Application Security Verification Standard (`IG3` [44]) to identify test cases and requirements**.

7.2.3 Summary & Important Considerations

In this section I described strategies for promoting security unit testing as a formal DevSecOps practice that are grounded in practitioner's perspectives and existing guidelines. Following their top-down approach, I first call upon security authorities to include functional security unit testing as a development-phase activity in Secure Development Lifecycle (SSDLC) models and frameworks as a means of encouraging organizational-level adoption support. Next, I recommend teams leverage the DevOps principle of cross-team (Dev + Sec) communication to help their developers apply a “security mindset” to adapt their current unit testing procedures and acquire security expertise that will empower them to work more independently over time. These are important first steps designed to shift from ad-hoc to formalized and sustained adoption by providing developers with organizational support (extrinsic motivation) and empowering them to increase their own security expertise (intrinsic motivation) through active participation in security activities. However **these data-driven recommendations reflect the community's general lack of attention towards accommodating developer's existing priorities and mitigating pain points they may experience beyond test planning**. This is likely a consequence of not understanding the current state-of practice (addressed by this work); for example we are the first to identify and define pain points unique to writing security-related unit tests. In the next section, I outline research directions and community efforts that must be undertaken to develop such support systems.

7.3 A Roadmap to Supporting Developer-Driven Security Testing

The previous section discussed strategies to support integrating security unit testing into developer workflows, with a strong focus on promoting formalized adoption and helping developers apply a "security mindset" to identify test cases. Beyond test case planning, **developers will need new support systems to overcome unique pain points associated with designing and constructing security unit tests**. At this stage in the testing lifecycle, developers need support systems that can help them recreate fine-grained security functionality (*e.g.* password validation) and configure fixtures and mocks for security components (*e.g.* fake certificates).

The research directions and support system recommendations described herein focus on creating reference material and automated tools to help developers *independently* overcome security unit testing pain points. In Section 7.2.2 I suggested that over time, consistent developer-security collaborations for test planning could produce reusable resources such as generic security test fixtures. However, the resource-availability assumption underlying this recommendation impacts its widespread applicability and places it in the realm of what is possible under "ideal conditions". To remain pragmatic, this section makes no assumptions about the availability or involvement of security personnel. In any case, unit test design and construction require in-depth knowledge of the system under test's implementation details that security personnel should not be expected to maintain, just as developers are not expected to hold expert-level security knowledge.

7.3.1 Towards Actionable Guidelines for Test Design & Construction

Challenges inherent to developer testing are well studied, and many resources enumerating best practices, patterns, *etc.* for avoiding them are available [108,127]. Likewise, there are guidelines and best practices for security testing such as those provided by OWASP [44,169]. Security unit testing lies at the intersection of these concepts, but presents unique challenges. **I call upon researchers and practitioners to work together towards creating more developer-centric security testing guidelines, using the following directions as a starting point.** The extensive supply of security APIs, tools, frameworks *etc.* available and the ever-evolving nature of security make independent efforts impractical, hence a *community effort* is needed for sustained improvement.

Task & Example-Based Documentation for Security APIs, Frameworks, Tools: Contextual Support for Recreating Security Scenarios in Test Environments

The lack of actionable guidelines to support developers in adapting unit testing practices for security are consistent with those from Acar *et al.*'s investigations into developer-centric security resources [3, 5]: testing guides focused on post-deployment techniques, support for 'automated testing' tools was restricted to static analyzers or fuzzers, and guidelines fall short in helping developers *use* security libraries and APIs. **Developers generally lack the security expertise needed to implement complex security controls and instead follow the best practice [121] of using those provided by third-party APIs and frameworks, making their usability an important attribute.** Prior work on the usability of cryptography APIs found that developers struggle to use them for implementation tasks and want more task and example-based documentation [1, 136].

Our study of security unit testing pain points (Chapter 5) suggests that developers also experience unique pain points when trying to write unit and integration tests with security APIs and frameworks. A consequence of relying on them to implement security controls is that fine-grained implementation details are abstracted from the developer. This obfuscation, combined with the inherent complexity of APIs themselves, were underlying factors for five of the seven pain points identified, most notably *Mocking Security Components*, *Designing Authentication Flows*, and *Interacting With* (Distributed) *Security Services & APIs* in test environments. **Although some efforts have been made to provide support for these challenges, available resources still do not appear to sufficiently cover or address all the needs of developers.** We first found evidence of this gap in our Q&A post analysis wherein we observed that documentation wikis and tool repository links (which may also contain documentation) were frequently added by post authors to indicate the resources they have checked before turning to their peers. Spring Security framework goes as far as providing libraries with additional components to support fixture design with corresponding example-based documentation to help developers use it [189], but our review found that developers still struggled to design test fixtures using these components; see the first pain point description in Table 5.1 for an example. In our grey literature review (Chapter 6.3) we noted that the topics and use cases described in the instructive material shared by practitioners match the subjects under test in these posts, further evidence of developer's reliance on peer support to guide them when more official sources fall short [2, 3].

The overall lack of task or example-based documentation for security APIs & frameworks and their limited use case coverage negatively affect developer's experiences using them for implementation *and* testing purposes. I encourage practitioners to continue sharing tutorials and demonstrations for challenges they have overcome, but the community should not rely on these to support developers due to the ad-hoc manner in which they are shared and updated. Instead, **we can begin to work towards providing developers with the support they need by focusing on adding concrete solutions for the most troublesome scenarios and tasks to official documentation, using the pain points defined this work as a starting point**. Our study was the first to provide such a examination of documentation needs and challenges for security unit testing, but more research should be also conducted to extend and further contextualize our pain point ontology, including looking beyond Q&A posts.

Guides and Resources for Developer-Driven Security Testing: Support for Realizing Abstract Test Cases as Executable Test Methods

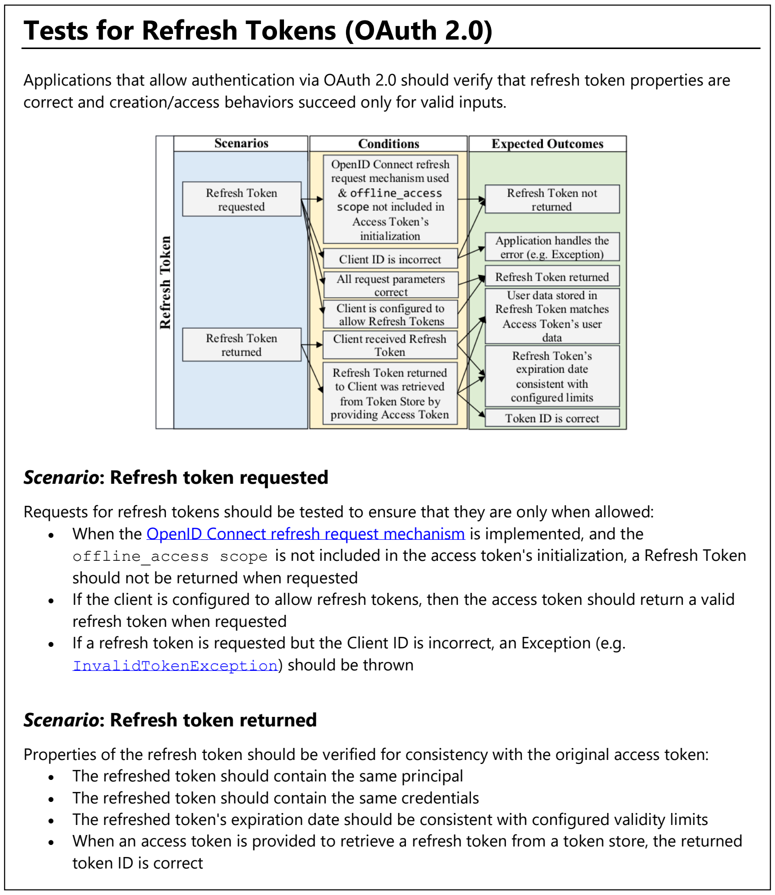
Unit tests [5] are unique from other testing techniques because they are implemented (with the help of automated testing frameworks) as executable methods that *directly exercise specific components of the system-under-test* to recreate and evaluate fine-grained system behaviors. Identifying the correct components, sequence of method calls, *etc.* to realize a test case requires in-depth knowledge of the system-under-test's design *and* it's implementation. The abstraction resulting from using third-party security frameworks, tools, and APIs can make this task even more difficult in addition to the previously-discussed challenges this introduces to security unit testing. As such **developers may find it difficult to receive the help they need post-planning from security personnel whose may lack sufficient depth-of-knowledge for control implementations**.

The need to trace test cases and requirements to specific source components and lines of code for unit testing can impact the usability of existing security references that present these only in natural language. This choice is appropriate to ensure these guides can be used regardless of technical context, and I do not suggest trying to augment these with unit testing details. Instead, **new references that supplement the authoritative security guides** (*e.g.* [44, 162, 169]) **already embraced by the community could provide developer-centric support to apply these guidelines in continuous security and DevSecOps contexts**. Future work should explore different approaches and directions for identifying and curating the appropriate content as well as evaluating format and delivery mechanisms. One of the first barriers to security unit testing is knowing what can be tested at this level. From our novel investigation of current practices (Chapter 4.1) we know that developers already write tests for at least nine security controls, components, and scenarios (Table 4.1) and our review of grey literature (Chapter 6) indicates this is consistent with those suggested as appropriate use cases by expert groups and independent practitioners. The OWASP Application Security Verification Standard [44] is a good starting point to derive formally-defined test cases and requirements that are unit-testable given its indications of categories that are well or poorly suited to this level of testing.

Many existing resources understandably orient their requirement and test recommendations towards detecting vulnerabilities and secure coding enforcement, so **to "close the gaps" between**

[5] As noted in Chapter 2.3.3, this work uses the term **unit tests** to represent unit *and* integration-level tests, security or otherwise, that are written by developers using automated testing frameworks (*e.g.* JUnit).

recommendations and guidelines, new references are also needed to help developers write functional unit tests for security controls. We took the initiative on creating functional security unit testing guidelines by publishing one for token authentication [82] [6]. This guide presents the taxonomy of test cases that emerged from our grounded theory analysis of unit test code (Chapter 4.1) in a format that relieves developers of the cognitive burden of identifying the "unit test format" for each test cases. Specifically we pair visual flow diagrams (Figures 4.1- 4.5) with natural language test case descriptions and additional contextual information. An excerpt from the guide showing tests for *Refresh Tokens* is given in Figure 7.1.


Refresh Token
Scenarios
Conditions
Expected Outcomes
Refresh Token requested
Refresh Token returned
OpenID Connect refresh request mechanism used & offline_access scope not included in Access Token's initialization
Client ID is incorrect
All request parameters correct
Client is configured to allow Refresh Tokens
Client received Refresh Token
Refresh Token returned to Client was retrieved from Token Store by providing Access Token
Refresh Token not returned
Application handles the error (e.g. Exception)
Refresh Token returned
User data stored in Refresh Token matches Access Token's user data
Refresh Token's expiration date consistent with configured limits
Token ID is correct

Tests for Refresh Tokens (OAuth 2.0)

Applications that allow authentication via OAuth 2.0 should verify that refresh token properties are correct and creation/access behaviors succeed only for valid inputs.

***Scenario*: Refresh token requested**

Requests for refresh tokens should be tested to ensure that they are only when allowed:

- When the OpenID Connect refresh request mechanism is implemented, and the `offline_access scope` is not included in the access token's initialization, a Refresh Token should not be returned when requested
- If the client is configured to allow refresh tokens, then the access token should return a valid refresh token when requested
- If a refresh token is requested but the Client ID is incorrect, an Exception (e.g. `InvalidTokenException`) should be thrown

***Scenario*: Refresh token returned**

Properties of the refresh token should be verified for consistency with the original access token:

- The refreshed token should contain the same principal
- The refreshed token should contain the same credentials
- The refreshed token's expiration date should be consistent with configured validity limits
- When an access token is provided to retrieve a refresh token from a token store, the returned token ID is correct

Figure 7.1: Excerpt from our Token Authentication Unit Test Guide

To evaluate the usefulness and accuracy of the testing guide, we sought feedback from practitioners. We corresponded with 4 practitioners from two groups: *unit test authors*, who wrote unit tests we analyzed in this study and *external practitioners*, who did not contribute to our data set but have

[6]The full guide is publicly available for download and viewing at **`https://github.com/daniellegonzalez/awesome-security-unit-testing/blob/main/TokenAuthentication-UnitTestGuide.pdf`**

experience with implementing authentication in Java. Overall, practitioners consider the token authentication unit testing guide to be useful, and none of the included test cases were marked as invalid. The guide's narrow scope and some missing test cases were the only significant drawbacks brought to our attention. Both unit test authors, who are on opposite ends of the *authentication experience spectrum*, felt the guide was useful. Their main concern was, in practice, developers plan testing from the *coverage* perspective. However, one felt they would benefit from using the guide for retrospective or later-stage testing — this insight was influential to the research directions for automated support I describe in the next section. We incorporated most of the minor formatting suggestions and missing test cases into the final version of the test guide. More details about this evaluation are available in the published version of this work [82].

7.3.2 Towards Automated Tools that Recognize Developer's Needs & Priorities

In this section, I will propose two new research directions towards creating automated support systems to improve the security unit testing experience by further empowering developers to independently and actively engage in their team's security efforts while limiting workflow interruptions. Both are designed to accommodate developer's pre-existing priorities and responsibilities, keeping them in control of when and to what degree these tools are applied. Human (*i.e.* sociotechnical) factors can influence adoption of security practices, as evidenced by the vast body of research exploring developer's experiences and challenges. A "developers as users" mindset can be applied to examine the usability of support systems from the developer perspective [3,33,112], or explorations and interventions can be studied to further explore "developer centered security" [146,193].

Insights into the perspectives and challenges that influence developer's security behavior (discussed further in Chapter 2.3.4) are highly relevant to the automation-heavy strategies proposed to shift security left, which often focus on introducing new tools (*e.g.* scanners) to continuous development pipelines. Practitioners are enthusiastic about security unit testing because it can be applied by adapting existing practices *without* adding new tools. Nonetheless, existing tools and frameworks for automated unit testing can potentially be augmented or supplemented to address some of the unique challenges of adapting this technique for security. It follows that as security unit testing lies at the intersection of secure development and functional unit testing, it is important to understand and account for the effect that developer's prior experiences with secure development and unit testing may have on their response to these efforts. My work towards establishing a holistic, empirical understanding of the state of practice for security unit testing takes this a step further by allowing factors unique to this technique to be incorporated as well.

Help Me (By) Mock(ing) This: Automated Generation of Test Fixtures for Security Controls

Action research for unit testing is mostly centered on removing the responsibility from developers though the automatic generation of test cases, oracles, and inputs. However, the findings from this study indicate developers would benefit from automated tools that can *guide them through* the process of designing security unit tests. Research could examine potential strategies for automatic generation of security fixtures (especially mocks), which was a major pain point observed in this study. Some language-based automated solutions have been proposed (*e.g.* [61]), but the unique challenges associated with testing security controls make them a prime candidate for focused efforts.

Code Coverage Now, Security Coverage Later: Test Suite Auditors as On-Demand Support for Test Management and Planning

To verify the findings from our grounded theory analysis of security unit test code and evaluate the applicability of our resulting token authentication unit testing guide we sought feedback from practitioners, some of whom authored tests that we analyzed. Developer's unit testing practices are motivated and driven by code coverage — our evaluation indicates that **developer's prioritization of meeting coverage goals may influence their interest in adding security unit testing to their workflow.** Even when asked about their thought processes when writing security unit tests that we analyzed, the interviewed developers insisted that they were only trying to achieve higher coverage overall and were not using a "security mindset". **After reviewing our testing guide, practitioner's feedback suggested that they would find it useful for taking a *retroactive* look at how well their security controls were tested.** I consider this one of the most promising directions for automated security unit testing support, because it is satisfies many of the criteria I have outlined throughout this chapter for successful adoption. Namely, it accounts for developer's existing priorities and motivations, provides support for security expertise, and would keep them engaged and in control of the process. **Future work should investigate potential solutions for aggregating and updating the data needed to develop a system to compare the contents of a unit test suite to a checklist of test cases that should be included to achieve high coverage of security-related scenarios and behaviors.** These efforts can draw from work and advances made in related areas such as automated test case generation, API misuse detection, and developer-centric security tooling. As demonstrated by similar efforts for security vulnerability testing [159,160,161,171,172,173], tools and resources designed for Behavior-Driven Development can help maintain the needed trace links and provide a standardized and human-readable format for automatic maintenance of test case knowledge bases.

Chapter 8

Conclusions

This work provides a comprehensive examination on the state of practice for security unit testing, an adaptive technique for shifting security testing into development workflows using existing infrastructure. I have examined this phenomena through three practitioner-centric lenses: *Practices* (what they do), *Perspectives and Guidelines* (what/how they think it should be done), and *Pain Points* (what challenges/barriers they face) to incorporate technical and socio-technical factors. I have applied an empirical, mixed method approach to design three studies in which we analyzed primary, practitioner-authored artifacts (unit test code, Q&A posts, and grey literature) to understand the state of practice from their point-of-view and reconcile beliefs with actual behavior.

This empirical understanding of current practices and barriers is a necessary precondition to developing strategies for workflow integration, providing developers with the right resources to address barriers, and to conduct further action research towards additional supportive interventions. To this end, I have highlighted gaps in existing guidelines that should be addressed to support unit-level functional security testing, taking the initiative on these efforts by producing a token authentication unit test guide built from the taxonomy of text cases that emerged from the analysis of unit test code. I have also made data-driven recommendations for immediately-applicable strategies to empower developers to write these tests as well as highlighting research directions to further mitigate barriers for which no support currently exists.

Bibliography

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

[2] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. You get where you're looking for: The impact of information sources on code security. In *2016 IEEE Symposium on Security and Privacy (SP)*, New York, USA, May 2016. IEEE.

[3] Yasemin Acar, Sascha Fahl, and Michelle L. Mazurek. You are Not Your Developer, Either: A Research Agenda for Usable Security and Privacy Research Beyond End Users. In *Proceedings - 2016 IEEE Cybersecurity Development, SecDev 2016*, pages 3–8, New York, USA, February 2017. Institute of Electrical and Electronics Engineers Inc.

[4] Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L. Mazurek, and Sascha Fahl. Security developer studies with github users: Exploring a convenience sample. In *Proceedings of the Thirteenth USENIX Conference on Usable Privacy and Security*, SOUPS '17, page 81–95, USA, 2017. USENIX Association.

[5] Yasemin Acar, Christian Stransky, Dominik Wermke, Charles Weir, Michelle L. Mazurek, and Sascha Fahl. Developers Need Support, Too: A Survey of Security Advice for Software Developers. In *Proceedings - 2017 IEEE Cybersecurity Development Conference, SecDev 2017*, pages 22–26, New York, USA, September 2017. IEEE.

[6] US General Services Administration. It security procedural guide: Ociso devsecops program cio-it security-19-102. `https://www.gsa.gov/cdnstatic/OCISO_DevSecOps_Program_%5bCIO-IT_Security_19-102_Initial_Release%5d_09-13-2019_0.pdf`, 9 2019.

[7] Arpit Aggarwal. Mocking spring security context for unit testing – arpit aggarwal blog. `https://aggarwalarpit.wordpress.com/2017/05/17/mocking-spring-security-context-for-unit-testing/comment-page-1/`, 5 2017.

[8] Arshad Ahmad, Chong Feng, Shi Ge, and Abdallah Yousif. A survey on mining stack overflow: question and answering (q&a) community. *Data Technologies and Applications*, 52(2):190–247, January 2018.

[9] Ben Alex, Luke Taylor, Rob Winch, Gunnar Hillert, Joe Grandja, and Jay Bryant. Spring security reference 5.1.2.release, 2017.

[10] C. Andersson and P. Runeson. Verification and validation in industry - a qualitative survey on the state of practice. In *Proceedings International Symposium on Empirical Software Engineering*. IEEE Comput. Soc, 2002.

[11] Debi Ashenden and Gail Ollis. Putting the sec in DevSecOps: Using social practice theory to improve secure software development. In *New Security Paradigms Workshop 2020*. ACM, October 2020.

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

[13] Hala Assal and Sonia Chiasson. Security in the Software Development Lifecycle. In *Proceedings of the Fourteenth USENIX Conference on Usable Privacy and Security*, SOUPS '18, pages 281–296, USA, 2018. USENIX Association.

[14] Hala Assal and Sonia Chiasson. "Think Secure from the Beginning": A Survey with Software Developers. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, pages 1–13, New York, NY, USA, 2019. Association for Computing Machinery.

[15] Jimmy Astle. Using mitre att&ck for research & testing — red canary. `https://redcanary.com/blog/security-testing-using-mitre-attack/`, 9 2018.

[16] Jeff Atwood and Joel Spolsky. Security stack exchange. `https://security.stackexchange.com/`, 2008.

[17] Jeff Atwood and Joel Spolsky. Stack overflow. `https://stackoverflow.com`, 2008.

[18] M. Jones B. Campbell, C. Mortimore. Security assertion markup language (saml) 2.0 profile for oauth 2.0 client authentication and authorization grants. RFC 7522, RFC Editor, 5 2015.

[19] Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. Mining questions asked by web developers. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, New York, USA, 2014. ACM.

[20] Hari Balasundaram. A baseline approach to security testing — box blog. `https://blog.box.com/a-baseline-approach-to-security-testing`, 9 2014.

[21] S. Baltes, L. Dumani, C. Treude, and S. Diehl. Sotorrent: Reconstructing and analyzing the evolution of stack overflow posts. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 319–330, 2018.

[22] Steffen Bartsch. Practitioners' perspectives on security in agile development. *Proceedings of the 2011 6th International Conference on Availability, Reliability and Security, ARES 2011*, pages 479–484, 2011.

[23] Anton Barua, Stephen W. Thomas, and Ahmed E. Hassan. What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering*, 19(3):619–654, November 2012.

[24] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice.* Addison-Wesley Professional, 2003.

[25] L. Bell, M. Brunton-Spall, R. Smith, and J. Bird. *Agile Application Security: Enabling Security in a Continuous Delivery Pipeline.* O'Reilly Media, 2017.

[26] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 179–190, 2015.

[27] Stefanie Beyer and Martin Pinzger. A manual categorization of android app development issues on stack overflow. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 531–535, New York, USA, September 2014. IEEE.

[28] R. Binder and B. Beizer. *Testing Object-oriented Systems: Models, Patterns, and Tools.* Addison-Wesley object technology series. Addison-Wesley, Boston, USA, 2000.

[29] J. Bird. *DevOpsSec: Securing Software Through Continuous Delivery.* O'Reilly Media, 2016.

[30] Jim Bird. *Security as code: security tools and practices in continuous delivery*, pages 29–68. O'Reilly Media, Incorporated, California, USA, 2016.

[31] Mike Bland. Goto fail, heartbleed, and unit testing culture. `https://martinfowler.com/articles/testing-culture.html`, 6 2014.

[32] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3(null):993–1022, March 2003.

[33] Lars-Ola Bligård, Helena Strömberg, and MariAnne Karlsson. Developers as users: Exploring the experiences of using a new theoretical method for usability assessment. *Advances in Human-Computer Interaction*, 2017:1–13, 2017.

[34] Matt Boegner. Integrating security into the ci/cd pipeline: Step-by-step recommendations. `https://mattboegner.com/secure_cicd_pipeline_2/`, 4 2017.

[35] Pierre Bourque, Richard E Fairley, et al. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0.* IEEE Computer Society Press, 2014.

[36] Duncan P. Brumby, Christian P. Janssen, and Gloria Mark. *How Do Interruptions Affect Productivity?*, pages 85–107. Apress, Berkeley, CA, 2019.

[37] Ilene Burnstein. *Practical software testing: a process-oriented approach.* Springer Science & Business Media, Berlin/Heidelberg, Germany, 2006.

[38] US-CERT CISA. Risk-based and functional security testing — cisa. `https://us-cert.cisa.gov/bsi/articles/best-practices/security-testing/risk-based-and-functional-security-testing`, 7 2013.

[39] Juliet M. Corbin and Anselm L. Strauss. *Basics of qualitative research: techniques and procedures for developing grounded theory.* SAGE, fourth edition edition, 2015.

[40] Microsoft Corporation. Simplified implementation of the sdl. Technical report, Microsoft Secure Development Lifecycle, 2010.

[41] MITRE Corporation. Devsecops – security and test automation briefing. `https://www.mitre.org/sites/default/files/publications/pr-19-0769-devsecops_security_test_automation-briefing.pdf`, 3 2019.

[42] Daniela Soares Cruzes, Michael Felderer, Tosin Daniel Oyetoyan, Matthias Gander, and Irdin Pekaric. How is security testing done in agile teams? a cross-case analysis of four software teams. In Hubert Baumeister, Horst Lichter, and Matthias Riebisch, editors, *International Conference on Agile Software Development*, volume 283 of *Lecture Notes in Business Information Processing*, pages 201–216, Cham, 2017. Springer, Cham, Springer International Publishing.

[43] Alex Cummaudo, Rajesh Vasa, Scott Barnett, John Grundy, and Mohamed Abdelrazek. Interpreting cloud computer vision pain-points. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, New York, USA, June 2020. ACM.

[44] Daniel Cuthbert, Andrew Van der Stock, and Jim Manico. Owasp application security verification standard, v4. *OWASP Foundation*, 4, 2021.

[45] Anderson Dadario. Security unit tests are important — dadario's blog. `https://dadario.com.br/security-unit-tests-are-important/`, 3 2016.

[46] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211. IEEE, 2014.

[47] Cypress Defense Data. Secure software development life cycle - development phase (ssdlc). `https://brittanyandrobert.com/why-secure-testing-is-important-6/`, 12 2015.

[48] Stephen De Vries. Security testing web applications throughout automated software tests. *OWASP Europe Conference*, 1(1):1–13, 2006.

[49] Stephen de Vries. Stephendevries-v2.pdf. `https://owasp.org/www-pdf-archive/AutomatedSecurityTestingofWebApplications-StephendeVries.pdf`, 5 2006.

[50] Stephen de Vries. Software Testing for security. *Network Security*, 2007(3):11–15, 3 2007.

[51] Stephen de Vries. Attacking Java Clients: A Tutorial, 2010.

[52] Stephen de Vries. Automated Security Testing in a Continuous Delivery Pipeline, 2015.

[53] Rebecca Deck. Adapting appsec to a devops world. `https://www.sans.org/reading-room/whitepapers/application/paper/38305`, 2 2018.

[54] Rebecca Deck. Improving application security through automated testing — directdefense. `https://www.directdefense.com/improving-application-security-automated-testing/`, 6 2018.

[55] Seba Deleersnyder and Bart De Win. OWASP software assurance maturity model (samm). `https://owaspsamm.org/`, 2009. Accessed 03-2020.

[56] Prem Devanbu, Thomas Zimmermann, and Christian Bird. Belief & evidence in empirical software engineering. In *Proceedings - International Conference on Software Engineering*, volume 14-22-May-, pages 108–119. IEEE Computer Society, May 2016.

[57] Donna Dodson, Murugiah Souppaya, and Karen Scarfone. Mitigating the risk of software vulnerabilities by adopting a secure software development framework (SSDF). Technical report, National Institute of Standards and Technology, April 2020.

[58] Matthew Dolan. Testing jetpack security with robolectric — by matthew dolan — proandroiddev. `https://proandroiddev.com/testing-jetpack-security-with-robolectric-9f9cf2aa4f61`, 8 2020.

[59] Julien DuBois, Deepu K. Sasidharan, and Pascal Grimaud. JHipster: Open Source application platform for creating Spring Boot + Angular/React projects in seconds!, 10 2011.

[60] Mark Fallon. Automating security testing. `https://www.commoncriteriaportal.org/iccc/7iccc/t3/t3201130.pdf`, 9 2006.

[61] Amin Milani Fard, Ali Mesbah, and Eric Wohlstadter. Generating fixtures for JavaScript unit testing (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, New York, USA, November 2015. IEEE.

[62] Michael Felderer, Matthias Büchler, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner. *Security Testing: A Survey*, volume 101. Academic Press Inc., Amsterdam, NL, January 2016.

[63] Ihor Feoktistov. What is devsecops and why should you care about the security as code culture? `https://relevant.software/blog/devsecops-the-security-as-code-culture/`, 8 2020.

[64] Nicole Forsgren, Jez Humble, and Gene Kim. *Accelerate: The Science of Lean Software and DevOps Building and Scaling High Performing Technology Organizations*. IT Revolution Press, 1st edition, 2018.

[65] Nicole Forsgren, Monica Chiarini Tremblay, Debra VanderMeer, and Jez Humble. Dora platform: Devops assessment and benchmarking. In *International Conference on Design Science Research in Information System and Technology*, pages 436–440. Springer, 2017.

[66] Apache Software Foundation. Apache shiro — simple. java. security. `https://shiro.apache.org/testing.html`.

[67] Martin Fowler, Jim Highsmith, et al. The agile manifesto. *Software Development*, 9(8):28–35, 2001.

[68] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[69] Vahid Garousi, Ahmet Coşkunçay, Aysu Betin-Can, and Onur Demirörs. A survey of software engineering practices in turkey. *Journal of Systems and Software*, 108:148–177, 2015.

[70] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology*, 106:101–121, 2019.

[71] Vahid Garousi, Michael Felderer, Mika V. Mäntylä, and Austen Rainer. Benefitting from the Grey Literature in Software Engineering Research. *Contemporary Empirical Methods in Software Engineering*, pages 385–413, 2020.

[72] Vahid Garousi, Austen Rainer, Per Lauvås, and Andrea Arcuri. Software-testing education: A systematic literature mapping. *Journal of Systems and Software*, 165:110570, July 2020.

[73] Vahid Garousi and Tan Varma. A replicated survey of software testing practices in the canadian province of alberta: What has changed from 2004 to 2009? *Journal of Systems and Software*, 83(11):2251–2262, 2010.

[74] Vahid Garousi and Junji Zhi. A survey of software testing practices in canada. *Journal of Systems and Software*, 86(5):1354–1376, 2013.

[75] Adam M Geras, Michael R Smith, and James Miller. A survey of software testing practices in alberta. *Canadian Journal of Electrical and Computer Engineering*, 29(3):183–191, 2004.

[76] Barney G Glaser. *Basics of grounded theory analysis: Emergence vs forcing*. Sociology press, CA, USA, 1992.

[77] Barney G Glaser, Anselm L Strauss, and Elizabeth Strutzel. The discovery of grounded theory; strategies for qualitative research. *Nursing research*, 17(4):364, 1968.

[78] D. Gonzalez, H. Hastings, and M. Mirakhorli. Automated characterization of software vulnerabilities. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 135–139. IEEE, 2019.

[79] Danielle Gonzalez, Fawaz Alhenaki, and Mehdi Mirakhorli. Architectural security weaknesses in industrial control systems (ics) an empirical study based on disclosed software vulnerabilities. In *2019 IEEE International Conference on Software Architecture (ICSA)*, pages 31–40. IEEE, 2019.

[80] Danielle Gonzalez, Andrew Popovich, and Mehdi Mirakhorli. TestEX: A Search Tool for Finding and Retrieving Example Unit Tests from Open Source Projects. In *2016 IEEE 27th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 153–159. IEEE, 2016.

[81] Danielle Gonzalez, Suzanne Prentice, and Mehdi Mirakhorli. AssertConvert: Automatically Convert JUnit Assertions to English, November 2018.

[82] Danielle Gonzalez, Michael Rath, and Mehdi Mirakhorli. Did You Remember to Test Your Tokens? In *2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR)*, New York, USA, June 2020. IEEE. To Appear.

[83] Danielle Gonzalez, Joanna C.S. Santos, Andrew Popovich, Mehdi Mirakhorli, and Mei Nagappan. A Large-Scale Study on the Usage of Testing Patterns That Address Maintainability Attributes: Patterns for Ease of Modification, Diagnoses, and Comprehension. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 391–401, New York, NY, 2017. IEEE.

[84] Danielle Gonzalez, Thomas Zimmermann, Patrice Godefroid, and Max Schäfer. Anomalicious: Automated detection of anomalous and potentially malicious commits on github. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 258–267, 2021.

[85] Danielle Gonzalez, Thomas Zimmermann, and Nachiappan Nagappan. The state of the ml-universe: 10 years of artificial intelligence & machine learning software development on github. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 431–442, 2020.

[86] Google. Build unit tests — firebase. `https://firebase.google.com/docs/rules/unit-tests`.

[87] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.

[88] Christopher Grayson. Security regression: Addressing security regression by unit testing. `https://qconnewyork.com/ny2017/system/files/presentation-slides/security_regression_06262017_2.pdf`, 6 2017.

[89] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. What eclipsers think and do about testing: A grounded theory. Technical report, Technical Report SERG-2011-010, Delft University of Technology, 2011. To appear, 2011.

[90] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Test confessions: A study of testing practices for plug-in systems. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, June 2012.

[91] M. Grindal, J. Offutt, and J. Mellin. On the testing maturity of software producing organizations. In *Testing: Academic Industrial Conference - Practice And Research Techniques (TAIC PART'06)*, pages 171–180, 2006.

[92] Lokesh Gupta. Test spring security auth with junit - howtodoinjava. `https://howtodoinjava.com/junit/how-to-unit-test-spring-security-authentication-with-junit/`, 4 2013.

[93] Phil Haack. Unit testing security example — you've been haacked. `https://haacked.com/archive/2007/09/21/unit-testing-security-example.aspx/`, 9 2007.

[94] Junxiao Han, Emad Shihab, Zhiyuan Wan, Shuiguang Deng, and Xin Xia. What do programmers discuss about deep learning frameworks. *Empirical Software Engineering*, 25(4):2694–2747, April 2020.

[95] Mubin Ul Haque, Leonardo Horn Iwaya, and M. Ali Babar. Challenges in docker development. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, New York, USA, October 2020. ACM.

[96] HashiCorp. Extending terraform - unit testing. `https://www.terraform.io/docs/extend/testing/unit-testing.html`, 7 2019.

[97] Mark Hermeling. Shift left quality and security with automated unit testing, dynamic and static analysis. `https://blogs.grammatech.com/shift-left-quality-and-security-with-automated-unit-testing-dynamic-and-static-analysis`, 7 2019.

[98] Achmad Nizar Hidayanto, Rinaldi, Putu Wuri Handayani, and Samuel Louvan. How secure your applications are?: Analysis of web developers awareness to application security. *International journal of innovation and learning*, 14(1):53–78, 2013.

[99] Michael Howard and Steve Lipner. *The security development lifecycle*, volume 8. Microsoft Press Redmond, 2006.

[100] Adrian Hsieh. Man in the mirror, apple's security flaw, and the importance of unit testing — mulesoft blog. `https://blogs.mulesoft.com/api-integration/security/mitm-automated-unit-testing/`, 3 2014.

[101] IEEE. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[102] Tanya Janca. Devsecops: Securing software in a devops world. `https://www.alldaydevops.com/blog/devsecops-securing-software-in-a-devops-world`, 9 2019.

[103] Jetbrains. The State of Developer Ecosystem 2019. `https://www.jetbrains.com/lp/devecosystem-2019/java/`, 2019.

[104] Eric Johnson. Secure devops: Faster feedback with effective security unit tests in ci / cd - sans institute. `https://www.sans.org/webcasts/secure-devops-faster-feedback-effective-security-unit-tests-ci-cd-106310/success`, 1 2018.

[105] Patricia Johnson. Security automation: Where does it fit in your secure sdlc? — whitesource. `https://www.whitesourcesoftware.com/resources/blog/security-automation/`, 7 2019.

[106] Jussi Kasurinen, Ossi Taipale, and Kari Smolander. Software test automation in practice: empirical observations. *Advances in Software Engineering*, 2010, 2010.

[107] Satyajit Ketkar. Static-dynamic analysis (unit testing) - white paper.pdf. `https://www.velentium.com/hubfs/Static-Dynamic%20Analysis%20(Unit%20Testing)%20-%20White%20Paper.pdf`, 7 2019.

[108] Vladimir Khorikov. *Unit testing: Principles, Practices, and Patterns.* Manning, Shelter Island, NY, 2020.

[109] R. Kneuper. Sixty years of software development life cycle models. *IEEE Annals of the History of Computing*, 39(3):41–54, 2017.

[110] Pavneet Singh Kochhar. Mining testing questions on stack overflow. In *Proceedings of the 5th International Workshop on Software Mining*, New York, USA, September 2016. ACM.

[111] Adam Kolawa. Reducing software security vulnerabilities through unit testing — military aerospace. `https://www.militaryaerospace.com/test/article/16708135/reducing-software-security-vulnerabilities-through-unit-testing`, 7 2005.

[112] Kati Kuusinen, Helen Petrie, Fabian Fagerholm, and Tommi Mikkonen. Flow, intrinsic motivation, and developer experience in software engineering. In *Agile Processes, in Software Engineering, and Extreme Programming*, Lecture Notes in Business Information Processing, pages 104–117. Springer International Publishing, May 2016. Volume: Proceeding volume: ; International Conference on Agile Software Development ; Conference date: 24-05-2016 Through 27-05-2016.

[113] Seth Law. https://www.blackhat.com/docs/asia-17/materials/asia-17-law-domo-arigato-mr-roboto-security-robots-a-la-unit-testing.pdf. `https://www.blackhat.com/docs/asia-17/materials/asia-17-Law-Domo-Arigato-Mr-Roboto-Security-Robots-A-La-Unit-Testing.pdf`, 3 2017.

[114] Triet Huynh Minh Le, David Hin, Roland Croft, and M. Ali Babar. PUMiner. In *Proceedings of the 17th International Conference on Mining Software Repositories*, New York, USA, June 2020. ACM.

[115] Jihyun Lee, Sungwon Kang, and Danhyung Lee. Survey on software testing practices. *IET software*, 6(3):275–282, 2012.

[116] Jiakun Liu, Xin Xia, David Lo, Haoxiang Zhang, Ying Zou, Ahmed E. Hassan, and Shanping Li. Broken external links on stack overflow. `https://arxiv.org/abs/2010.04892`, 2020.

[117] Jiakun Liu, Haoxiang Zhang, Xin Xia, David Lo, Ying Zou, Ahmed E. Hassan, and Shanping Li. An exploratory study on the repeatedly shared external links on stack overflow. `https://arxiv.org/abs/2104.03518`, 2021.

[118] Tamara Lopez, Thein Tun, Arosha Bandara, Levine Mark, Bashar Nuseibeh, and Helen Sharp. An investigation of security conversations in stack overflow. In *Proceedings of the 1st International Workshop on Security Awareness from Design to Deployment - SEAD '18*, New York, USA, May 2018. ACM.

[119] Tamara Lopez, Thein Tun, Arosha Bandara, Levine Mark, Bashar Nuseibeh, and Helen Sharp. An anatomy of security conversations in stack overflow. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, New York, USA, May 2019. IEEE.

[120] Mike Loukides. *What is DevOps?* " O'Reilly Media, Inc.", 2012.

[121] Jim Manico, Jim Bird, and Katy Anton. Owasp top 10 proactive controls (2018). `https://owasp.org/www-project-proactive-controls/`, 2018.

[122] Gary McGraaw. *Risk-Based Security Testing*, pages 187–204. Addison-Wesley Professional, 2006.

[123] Gary McGraw. *Software Security: Building Security In.* Addison-Wesley Professional, 2006.

[124] Sarah Meldrum, Sherlock A. Licorish, and Bastin Tony Roy Savarimuthu. Crowdsourced knowledge on stack overflow: A systematic mapping study. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, EASE'17, page 180–185, New York, NY, USA, 2017. Association for Computing Machinery.

[125] Marcilio Mendoncaa. Devops meets security: Security testing your aws application: Part i – unit testing — aws developer blog. `https://aws.amazon.com/blogs/developer/devops-meets-security-security-testing-your-aws-application-part-i-unit-testing/`, 8 2016.

[126] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code.* Addison-Wesley, Upper Saddle River, NJ, 2007.

[127] Gerard Meszaros. *xUnit test patterns: Refactoring test code.* Pearson Education, London, England, 2007.

[128] Mahmoud Mohammadi, Bill Chu, and Heather Richter Lipford. Detecting cross-site scripting vulnerabilities through automated unit testing. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 364–373, New York, USA, jul 2017. IEEE.

[129] Mahmoud Mohammadi, Bill Chu, Heather Richter Lipford, and Emerson Murphy-Hill. Automatic web security unit testing: Xss vulnerability detection. In *2016 IEEE/ACM 11th International Workshop in Automation of Software Test (AST)*, pages 78–84, New York, USA, 2016. IEEE.

[130] Patrick Morrison, Benjamin H Smith, and Laurie Williams. Measuring security practice use: a case study at ibm. In *2017 IEEE/ACM 5th International Workshop on Conducting Empirical Studies in Industry (CESI)*, pages 16–22. IEEE, IEEE, may 2017.

[131] Patrick Morrison, Benjamin H. Smith, and Laurie Williams. Surveying Security Practice Adherence in Software Development. In *Proceedings of the Hot Topics in Science of Security: Symposium and Bootcamp*, volume Part F1271 of *HoTSoS*, pages 85–94, New York, NY, USA, 2017. Association for Computing Machinery.

[132] Juni Mukherjee. Devsecops: Injecting security into cd pipelines — atlassian. `https://www.atlassian.com/continuous-delivery/principles/devsecops`, 2021.

[133] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, 2017.

[134] Ken Murgrage. Add security testing to your deployment pipelines — gocd blog. `https://www.gocd.org/2016/02/08/not-done-unless-its-done-security/`, 2 2016.

[135] Emerson Murphy-Hill and Stefan Wagner. *Software Productivity Through the Lens of Knowledge Work*, pages 57–65. Apress, Berkeley, CA, 2019.

[136] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. "jumping through hoops": Why do java developers struggle with cryptography apis? In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*, New York, USA, 2016. ACM.

[137] David Newman, Jey Han Lau, Karl Grieser, and Timothy Baldwin. Automatic evaluation of topic coherence. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, HLT '10, page 100–108, USA, 2010. Association for Computational Linguistics.

[138] SP Ng, Tafline Murnane, Karl Reed, D Grant, and Tsong Yueh Chen. A preliminary survey on software testing practices in australia. In *2004 Australian Software Engineering Conference. Proceedings.*, pages 116–125. IEEE, 2004.

[139] Breno B Nicolau de França, Helvio Jeronimo Junior, and Guilherme Horta Travassos. Characterizing DevOps by Hearing Multiple Voices. In *Proceedings of the 30th Brazilian Symposium on Software Engineering - SBES '16*, New York, New York, USA, 2016. ACM Press.

[140] Rajai Nuseibeh. Devops security — checkmarx application security. `https://www.checkmarx.com/glossary/devops-security/`, 6 2013.

[141] Gerardo Orellana, Gulsher Laghari, Alessandro Murgia, and Serge Demeyer. On the differences between unit and integration testing in the TravisTorrent dataset. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, New York, USA, May 2017. IEEE.

[142] Stack Overflow. Annual developer survey. `https://insights.stackoverflow.com/survey/2020`, 2020.

[143] Gerard O'Regan. Concise guide to software engineering. *Concise Guide to Software Engineering, Cork, Ireland: Springer*, pages 131–138, 2017.

[144] Sofia Parafina. Unit testing infrastructure — pulumi. `https://www.pulumi.com/blog/unit-test-infrastructure/`, 3 2020.

[145] Ron Patton. *Software Testing (2nd Edition).* Sams, USA, 2005.

[146] Olgierd Pieczul, Simon Foley, and Mary Ellen Zurko. Developer-Centered Security and the Symmetry of Ignorance. In *Proceedings of the 2017 New Security Paradigms Workshop*, NSPW 2017, pages 46–56, New York, NY, USA, 2017. Association for Computing Machinery.

[147] David Pine. Asp.net core unit testing for security attributes. `https://davidpine.net/blog/asp-net-core-security-unit-testing/`, 3 2019.

[148] Andreas Poller, Laura Kocksch, Sven Türpe, Felix Anand Epp, and Katharina Kinder-Kurlanda. Can security become a routine? a study of organizational change in an agile software development group. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*, pages 2489–2503, 2017.

[149] Bruce Potter and Gary McGraw. Software security testing. *IEEE Security & Privacy*, 2(5):81–85, 2004.

[150] R.S. Pressman and D. Bruce R. Maxim. *Software Engineering: A Practitioner's Approach.* McGraw-Hill Education, 2014.

[151] ed. R. Harrison. Lightweight directory access protocol(ldap):authentication methods and security mechanisms. RFC 4513, RFC Editor, 6 2006.

[152] A A U Rahman and L Williams. Software Security in DevOps: Synthesizing Practitioners' Perceptions and Practices. In *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)*, pages 70–76, May 2016.

[153] Arvind Rai. Spring 4 security junit test with @withmockuser and @withuserdetails annotation example using @webappconfiguration. `https://www.concretepage.com/spring-4/spring-4-security-junit-test-with-withmockuser-and-withuserdetails-annotation-example-usi` 5 2015.

[154] Austen Rainer. Using argumentation theory to analyse software practitioners' defeasible evidence, inference and belief. *Information and Software Technology*, 87:62–80, 2017.

[155] Austen Rainer, Tracy Hall, and Nathan Baddoo. Persuading developers to" buy into" software process improvement: a local opinion and empirical evidence. In *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.*, pages 326–335. IEEE, 2003.

[156] Austen Rainer and Ashley Williams. Heuristics for improving the rigour and relevance of grey literature searches for software engineering research. *Information and Software Technology*, 106:231–233, 2019.

[157] Austen Rainer and Ashley Williams. Using blog-like documents to investigate software practice: Benefits, challenges, and research directions. *Journal of Software: Evolution and Process*, 31(11), November 2019.

[158] Päivi Raulamo-Jurvanen, Mika Mäntylä, and Vahid Garousi. Choosing the right test automation tool: A grey literature review of practitioner sources. In *EASE'17: Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, volume Part F1286, pages 21–30, New York, New York, USA, June 2017. Association for Computing Machinery.

[159] Loukmen Regainia. *Assisting in secure application development and testing.* PhD thesis, Université Clermont Auvergne, 2018.

[160] Loukmen Regainia, Sebastien Salva, and Cedric Ecuhcurs. A classification methodology for security patterns to help fix software weaknesses. In *2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)*, volume 0, pages 1–8. IEEE, 11 2016.

[161] Loukmen Regainia and Sébastien Salva. A Practical Way of Testing Security Patterns. In *International Conference on Software Engineering Advances*, Nice, France, 2018.

[162] Dominique et al. Righetto. OWASP authentication cheatsheet. `https://bit.ly/353sm2E`, 2019.

[163] Leonardo Jimenez Rodriguez, Xiaoran Wang, and Jilong Kuang. Insights on apache spark usage by mining stack overflow questions. In *2018 IEEE International Congress on Big Data (BigData Congress)*, New York, USA, July 2018. IEEE.

[164] Matthias Rohr. Automatic testing for security headers - sustainable application securitysustainable application security. `https://blog.secodis.com/2014/05/04/security-unit-tests/`, 5 2014.

[165] Christoffer Rosen and Emad Shihab. What are mobile developers asking about? a large scale study using stack overflow. *Empirical Software Engineering*, 21(3):1192–1223, April 2015.

[166] Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. Deeptc-enhancer: Improving the readability of automatically generated tests. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 287–298, 2020.

[167] W Royce. The software lifecycle model (waterfall model). In *Proc. Westcon*, volume 314, 1970.

[168] P. Runeson. A survey of unit testing practices. *IEEE Software*, 23(4):22–29, 2006.

[169] Elie Saad, Matteo Meucci, and Rick Mitchell. Owasp testing guide, v4. *OWASP Foundation*, 4:66–80, 2013.

[170] Salesforce. Enhanced transaction security apex testing. `https://help.salesforce.com/articleView?id=enhanced_transaction_security_apex_testing.htm&type=5`.

[171] Sébastien Salva and Loukmen Regainia. An approach for guiding developers in the choice of security solutions and in the generation of concrete test cases. *Software Quality Journal*, 27(2):675–701, jun 2019.

[172] Sébastien Sebastien Salva and Loukmen Regainia. *Risks and Security of Internet and Systems*, volume 10694 of *Lecture Notes in Computer Science*. Springer International Publishing, Cham, 2018.

[173] Sébastien Salva and Loukmen Regainia. Using Data Integration for Security Testing. In *International Federation for Information Processing*, volume 10533 LNCS, pages 178–194. Springer International Publishing, 2017.

[174] Dave Sayer. Spring security architecture, 2019. Link: `https://spring.io/guides/topicals/spring-security-architecture`.

[175] Ina Schieferdecker, Juergen Grossmann, and Martin Schneider. Model-based security testing. *arXiv preprint arXiv:1202.6118*, 2012.

[176] C.B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, 1999.

[177] Spring Security. 11. testing method security. `https://docs.spring.io/spring-security/site/docs/4.2.x/reference/html/test-method.html`, 10 2020.

[178] Servoy. Security (non-mobile solutions) and plugin usage. `https://wiki.servoy.com/display/DOCS/Unit+Testing#UnitTesting-Security(non-mobilesolutions)andpluginusage`, 1 2017.

[179] Chris Sevilleja. The ins and outs of token based authentication. `https://scotch.io/tutorials/the-ins-and-outs-of-token-based-authentication`, 2015.

[180] Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, editors. *Guide to Advanced Empirical Software Engineering*. Springer London, 2008.

[181] Chua Bo Si. Appsec - 5 tips for a superb testing strategy — hackerone. `https://www.hackerone.com/blog/5-Tips-Effective-AppSec-Testing-Strategy`, 1 2019.

[182] Ben Smith and Laurie Ann Williams. A survey on code coverage as a stopping criterion for unit testing. Technical report, North Carolina State University. Dept. of Computer Science, 2008.

[183] Jayson Smith. What is cucumber? `https://cucumber.io/docs/gherkin/`, 2019.

[184] Larry Smith. Shift-left testing, 2001.

[185] Richard Smith. Crafting an effective security organization — etsy. `https://www.infoq.com/presentations/security-etsy/`, 9 2015.

[186] Jacopo Soldani, Damian Andrew Tamburri, Willem-Jan Van, and Den Heuvel. The pains and gains of microservices: A Systematic grey literature review. *The Journal of Systems and Software*, 146:215–232, 2018.

[187] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, USA, 9th edition, 2010.

[188] Davide Spadini, Mauricio Aniche, Magiel Bruntink, and Alberto Bacchelli. To mock or not to mock? an empirical study on mocking practices. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, New York, USA, May 2017. IEEE.

[189] Spring. Spring security testing, 2017. Link: `https://docs.spring.io/spring-security/site/docs/current/reference/html/test.html`.

[190] Spring. Spring security api documentation, 2018. Link: `https://docs.spring.io/spring-security/site/docs/5.2.1.RELEASE/api/`.

[191] Keith Stevens, Philip Kegelmeyer, David Andrzejewski, and David Buttler. Exploring topic coherence over many models and many topics. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, EMNLP-CoNLL '12, page 952–961, USA, 2012. Association for Computational Linguistics.

[192] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: A critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 120–131, New York, NY, USA, 2016. Association for Computing Machinery.

[193] Mohammad Tahaei and Kami Vaniea. A survey on developer-centred security. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, June 2019.

[194] Alexander Tarlinder. *Developer testing: Building quality into software.* Addison-Wesley Professional, Boston, MA, USA, 2016.

[195] Fangchao Tian, Peng Liang, and Muhammad Ali Babar. How developers discuss architecture smells? an exploratory study on stack overflow. In *2019 IEEE International Conference on Software Architecture (ICSA)*, New York, USA, March 2019. IEEE.

[196] Gu Tian-yang, Shi Yin-sheng, and Fang You-yuan. Research on software security testing. *International Journal of Computer and Information Engineering*, 4(9):1446–1450, 2010.

[197] Fabian Trautsch, Steffen Herbold, and Jens Grabowski. Are unit and integration test definitions still valid for modern java projects? an empirical study on open-source projects. *Journal of Systems and Software*, 159:110421, January 2020.

[198] Peleus Uhley. An overview of behavior driven development tools for ci security testing. `https://web.archive.org/web/20200813122427/https://blogs.adobe.com/security/2014/07/overview-of-behavior-driven-development.html`, 7 2014.

[199] Danny van Bruggen and Federico Tomassetti. JavaParser: Java 1-13 Parser and Abstract Syntax Tree for Java, 10 2013.

[200] Dirk van der Linden, Pauline Anthonysamy, Bashar Nuseibeh, Thein T Tun, Marian Petre, Mark Levine, John Towse, and Awais Rashid. Schrödinger's security: Opening the box on app developers' security rationale. In *42nd International Conference on Software Engineering (ICSE)*. Association for Computing Machinery (ACM), 2020.

[201] Andrew van der Stock, Brian Glas, and T Gigler. Owasp top 10 2017: The ten most critical web application security risks. *OWASP Foundation*, null(null):23, 2017.

[202] Gabriel Varaljay. Devsecops and agile: Build process in security testing - app-ray. `https://app-ray.co/2020/03/01/devsecops-and-agile-build-process-in-security-testing/`, 3 2020.

[203] Julien Vehent. *Securing DevOps: Security in the Cloud*. Manning Publications Co., New York, USA, 2018.

[204] John Viega and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way (Paperback) (Addison-Wesley Professional Computing Series)*. Addison-Wesley Professional, Boston, MA, USA, 1st edition, 2011.

[205] Isabel K. Villanes, Silvia M. Ascate, Josias Gomes, and Arilo Claudio Dias-Neto. What are software engineers asking about android testing on stack overflow? In *Proceedings of the 31st Brazilian Symposium on Software Engineering - SBES'17*, New York, USA, 2017. ACM.

[206] Daniel Jared Votipka. *A Human-Centric Approach to Software Vulnerability Discovery*. PhD thesis, University of Maryland, 2020.

[207] Matt Watson. What is devsecops? how to automate security testing – stackify. `https://stackify.com/devsecops-automate-security-testing/`, 6 2017.

[208] David Wayland. Security driven development (moving security to the left) — linkedin. `https://www.linkedin.com/pulse/security-driven-development-moving-left-david-wayland/`, 11 2017.

[209] Charles Weir, Ingolf Becker, James Noble, Lynne Blair, M. Angela Sasse, and Awais Rashid. Interventions for long-term software security: Creating a lightweight program of assurance techniques for developers. *Software: Practice and Experience*, 50(3):275–298, November 2019.

[210] Roel J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer Berlin Heidelberg, Berlin, DE, 2014.

[211] Ashley Williams and Austen Rainer. How do empirical software engineering researchers assess the credibility of practitioner-generated blog posts? In *EASE '19: Proceedings of the Evaluation and Assessment on Software Engineering*, pages 211–220, 2019.

[212] Margaret A. Wojcicki and Paul Strooper. A state-of-practice questionnaire on verification and validation for concurrent programs. In *Proceedings of the 2006 Workshop on Parallel*

and Distributed Systems: Testing and Debugging, PADTAD '06, page 1–10, New York, NY, USA, 2006. Association for Computing Machinery.

[213] Chris Wysopal, Lucas Nelson, Elfriede Dustin, and Dino Dai Zovi. *The art of software security testing: identifying software security flaws.* Pearson Education, London, UK, 2006.

[214] Xin-Li Yang, David Lo, Xin Xia, Zhi-Yuan Wan, and Jian-Ling Sun. What Security Questions Do Developers Ask? A Large-Scale Study of Stack Overflow Posts. *Journal of Computer Science and Technology*, 31(5):910–924, September 2016.

[215] Jiewen Yao and Vincent Zimmer. Security Unit Test. In *Building Secure Firmware*, chapter 21, pages 857–874. Apress, Berkeley, CA, 2020.

[216] Jiewen Yao and Vincent Zimmer. *Security Unit Test*, pages 857–874. Apress, 2020.

[217] Deheng Ye, Zhenchang Xing, and Nachiket Kapre. The structure and dynamics of knowledge network in domain-specific q&a sites: a case study of stack overflow. *Empirical Software Engineering*, 22(1):375–406, April 2016.

[218] Benwen Zhang, Emily Hill, and James Clause. Towards automatically generating descriptive names for unit tests. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 625–636, New York, NY, 2016. IEEE.

[219] Peng Zhang. What topics do developers concern? an analysis of java related posts on stack overflow. In *2019 2nd International Conference on Artificial Intelligence and Big Data (ICAIBD)*, New York, USA, May 2019. IEEE.

[220] Waleed Zogaan, Ibrahim Mujhid, Joanna C. S. Santos, Danielle Gonzalez, and Mehdi Mirakhorli. Automated training-set creation for software architecture traceability problem. *Empirical Software Engineering*, 22(3):1028–1062, November 2016.

[221] Radim Řehůřek and Michael Penkov. Documentation: Lda model tutorial. `https://radimrehurek.com/gensim/auto_examples/tutorials/run_lda.html`, 2019.

[222] Radim Řehůřek and Petr Sojka. Gensim—statistical semantics in python. `https://radimrehurek.com/gensim_4.0.0/index.html`, 2011.

Appendices

Appendix A

Grounded Theory Analysis of Security Unit Test Code

This appendix provides the **unabridged methodology** for the unit test code analysis to supplement the description given in Chapter 3.1. **Data Collection and Processing**
To study our phenomena of interest, we needed access to *authentication unit tests* from real projects. Therefore, we targeted data sources that were freely accessible to us. In this context, we focused on open source projects that implemented *authentication* and have corresponding unit tests. To facilitate the use of an automated approach to mine a large and rich dataset for our analysis, we set the following narrowing criteria for language, authentication framework, and testing framework: the projects must be implemented in **Java**, use the **Spring Security** framework for authentication, and unit tests must be written using the **JUnit** framework. Java was selected because it is a popular choice for back-end development of web applications [103]. The Spring Security framework is a very popular framework used to implement authentication in Java, and JUnit is the mainstream testing framework for Java projects [83, 103].

The final data set used for analysis consisted of **481 test methods from 125 unique test files, mined from 53 projects**. These tests were collected using the following procedure:

Data Source: We selected GitHub as a repository source. GHTorrent [87] and the REAPER tool [133] were used to automatically identify, clone, and filter 100,000 randomly selected Java projects from GitHub. The data set was filtered for empty, duplicate, and inactive (no commits or recent updates) repositories using metadata from GitHub API: size, commits, and date of last push. The test ratio metric from REAPER was used to discard projects with no JUnit tests, leaving 33,090 candidate projects. An automated script was developed to parse these projects and verify use of Spring Security, the target authentication framework.

Detecting Use of Spring Security: A package-search approach was used to identify projects that used Spring Security to implement authentication. To curate a list of relevant packages, we reviewed several forms of documentation provided by Spring: an overview of the framework's architecture [174], the comprehensive reference document [9], and the API documentation [190]. First, we identified objects *required* to implement authentication such as an `Authentication Provider`. These were used as a starting point to find the root packages providing each component in the API. Next, we reviewed *all* root packages in the API documentation to identify packages for flexible components such as the numerous authentication mechanisms (*e.g.* LDAP, SAML) Spring Security can integrate with. We excluded packages related to *sessions* to prevent extending the phenomena under study to session management. Table A.1 lists the final set of 18 root packages with descriptions.

Identifying Relevant JUnit Test *Files* Included in REAPER's repository analysis was a list of JUnit test files for each project. Our script used these lists and our authentication package list to automatically search each test file for imports of one or more of these packages. We addedan additional rule that test files importing the *Security Context Holder* or *User Details* packages should also import at least one more from the list, because these packages are also used to implement authorization. We randomly selected 50 resulting test files to ensure there were no false positives. This search identified 229 test files from 150 projects.

Duplicate Removal: A major component GitHub's contribution workflow is *forking*, or making a copy, of a repository. Forks in our data set were identified from the GHTorrent metadata. If forks were discovered, the original repository was kept and all forks were discarded. Forks containing unique and relevant tests would have been kept, but this situation did not arise.

We also discovered that several projects in the dataset were built using the JHipster development platform [199]. This platform helps developers build highly customizable web applications with a Spring-based server side. JHipster also provides a test suite for the code it generates, that includes relevant authentication tests. In our dataset, we found 81 projects that *only* included the provided test file without adding additional related tests to this file or others (as detected by our tools). To avoid adding duplicates of the same test method to our dataset, we excluded these projects but included the original file with the authentication test cases from the JHipster [199] repository.

Identifying Relevant JUnit Test *Methods*: Next, we built another tool that utilized the Java-

Table A.1: Spring Security API Packages Used To Detect JUnit Tests For Authentication

Package Name	Description From API Documentation
security.authentication	Core classes and interfaces related to user authentication
security.core.authentication	Represents tokens for authentication requests or an authenticated principal
security.saml	Allows combination of SAML 2.0 and authentication & federation mechanisms in a single application
security.kerberos	An extension for using Kerberos concepts with Spring
security.oauth	Support for using Spring Security with OAuth (1a)
security.oauth2	Support for using Spring Security with OAuth 2.0
security.config.oauth2	Core classes and interfaces providing support for OAuth 2.0
security.config.authentication	Parsing of Authentication Manager and related elements
security.openid	Authenticates standard web browser users via OpenID
security.cas.authentication	An Authentication Provider that can process CAS service & proxy tickets
security.cas.web.authentication	Authentication processing mechanisms that allow submitting credentials using CAS
security.web.authentication	Authentication processing mechanisms that allow submitting credentials using various protocols
security.web.server.authentication	Authentication mechanisms specific to server authentication
security.config.annotation.authentication	Interface for operating on a SecurityBuilder that creates a ProviderManager
security.config.annotation.web.configurers.oauth2	Contains AbstractHttpConfigurers for OAuth 2.0
security.ldap.authentication	Spring Security's LDAP module
security.core.context.SecurityContextHolder	Associates a given SecurityContext with the current execution thread
security.core.userdetails	The standard interfaces for implementing user data DAOs

Parser [59] Symbol Solver to identify *individual test methods* within these files that used objects from the imported packages. This AST parser uses the test file and the source project to identify the package of origin for an object. Our tool was configured to record all test methods that had at least one object originating from one of the 18 root packages on our list, with the additional rule for the *Security Context Holder* and *User Details* packages from test file selection. To verify the automated identification (see Section A), we performed a manual validation of 50 randomly selected results, and discarded test methods that did not *use* packages from our candidate set. Our final dataset contained 481 unique test methods.

Grounded Theory Analysis

The 7 core activities of "classic" grounded theory were performed on the unit test methods as follows. Figure 3.1 in Chapter 3.1 (Study Design) also provides an overview of this process.

Open Coding & Memoing: Two authors independently performed *open coding* [76, 192] by annotating the 481 test methods in our data set with organically-derived **codes** reflecting *concepts and themes* present in the test (*cf.* ① in Figure 3.1). Authors reviewed each test method and its setup & helper methods. Accurately identifying *themes* and *concepts* requires thorough understanding of each test method. Prior work in test summarization [218] has shown that test comprehension can be achieved by identifying a test's *context*: precondition(s) & setup; *action*: manipulation(s) of the subject-under-test; and *expected outcome*: desired state and result(s) after performing the action.

We captured these concepts in **memos** formatted as *Gherkin scenarios* [183]. Gherkin is used in behavior-driven development to capture scenarios in a non-technical & readable format. Test components are represented using three keywords: *Given* (context), *When* (action), & *Then* (expected outcome). The *And* keyword is used when a components spans multiple statements in a test. When an existing memo fully represented a new test method, no new memo was created to avoid duplicate memos. However, the test was still annotated with codes. We attached the codes as well as more data in a custom *Metadata* section in each scenario (*cf.* bottom block in Listing A.2), which is not part of the official Gherkin syntax.

Listing A.1 shows a unit test `testJWTFilterInvalidtoken()` that verifies whether a request containing an invalid JSON Web token (JWT) is filtered and no authentication occurs. Also shown are setup and helper methods from the test class that provide additional context. Listing A.2 shows the test's corresponding memo in the Gherkin syntax. The *Token-Authentication* and *Filter-Chain* codes represent core authentication concepts in the test.

Constant Comparative Analysis: In open coding, concepts are organically extracted from the data rather than assigned from an existing set. Initial codes are often broad, but additional analysis will reveal more specific patterns and concepts. To accommodate this in our analysis we incorporated *constant comparative analysis* [77] into the open coding process. Test methods & memos were constantly compared to identify reoccurring themes and note variations. New codes were constantly compared against the existing set to merge codes that were too fine-grained (not enough data points) and refine overly-broad codes. For example, one initial code was *tokens*, which are objects used to store and authenticate user credentials. As more test methods were reviewed this code was refined into 3 new codes representing these actions: *verify token properties*, *token authentication*, and *refresh token.* Another code (*verify user properties*) was merged with *verify token properties*

```java
public class JWTFilterTest {
  private TokenProvider tokenProvider;
  private JWTFilter jwtFilter;

  @Before
  public void setup() {
    JHipsterProperties jHipsterProperties = new JHipsterProperties();
    tokenProvider = new TokenProvider(jHipsterProperties);
    ReflectionTestUtils.setField(tokenProvider, "secretKey", "test␣secret");
    ReflectionTestUtils.setField(tokenProvider, "tokenValidityInMilliseconds",
      60000);
    jwtFilter = new JWTFilter(tokenProvider);
    SecurityContextHolder.getContext().setAuthentication(null);
  }

  @Test
  public void testJWTFilterInvalidToken() throws Exception {
    String jwt = "wrong_jwt";
    MockHttpServletRequest request = new MockHttpServletRequest();
    request.addHeader(JWTConfigurer.AUTHORIZATION_HEADER, "Bearer␣" + jwt);
    request.setRequestURI("/api/test");
    MockHttpServletResponse response = new MockHttpServletResponse();
    MockFilterChain filterChain = new MockFilterChain();

    jwtFilter.doFilter(request, response, filterChain);

    assertThat(response.getStatus()).isEqualTo(HttpStatus.OK.value());
    assertThat(SecurityContextHolder.getContext().getAuthentication()).isNull();
  }
  // ..
}
```

Listing A.1 An example JUnit test from the data set with corresponding setup & helper methods.

```
Scenario: JWT Filter Invalid Token
  Given A new JHipster Properties Object is initialized
  And A Token Provider is initialized with the JHipster Properties Object
  And The Token Provider's secretKey attribute is initialized
  And The Token Provider's tokenValidityInMilliseconds attribute is initialized
  And A new JWT Filter is initialized with the Token Provider
  And The Security Context's Authentication Token Object is set to null
  And A "wrong" String representation of a token is created
  And A new Mock HTTP Servlet Request is initialized and its Authorization
    header is set with the wrong String Token
  And the Mock HTTP Servlet Request's Request URI is set
  And A new Mock HTTP Servlet Response is initialized
  And A new Mock Filter Chain is initialized
  When The JWT FIlter's doFilter method is called with the Mock HTTP Servlet
    Request, Mock HTTP Servlet Response, and Mock Filter Chain
  Then The Status of the HTTP Servlet Response is 200
  And The Security Context's Authentication property is null

  Metadata:
    ProjectName: loja
    FileName: JWTFilterTest.java
    TestName: JWTFilterInvalidToken
    Codes: Token-Authentication, Filter-Chain, multiple-assertion
```

Listing A.2 Sample memo & codes for the test in Listing A.1.

because further analysis revealed that in all cases, these properties were retrieved from a token.

Selective Coding: After all the unit tests were annotated, codes were compared to review concepts and themes discovered. During *selective coding* [77] deeper analysis was conducted to refine the identified concepts (*cf.* ② in Figure 3.1). For each code, all relevant unit test methods were analyzed in-depth to capture all unique combinations of test *contexts*, *actions*, *conditions*, & *expected outcomes*) associated with the code. Constant comparative analysis was applied to modify existing codes to incorporate these combinations. For example, *verify token properties* was refined into four new codes: *input validation during token initialization*, *pre-authentication verification of token properties*, *post-authentication verification of token properties*, and *token validation fails with incorrect input.* As selective coding progressed, multiple tests sharing the same unique combination of these elements were iteratively merged into natural language test cases. The resulting collection of unique authentication test cases grouped by code form the basis for the study's conclusions.

Memo Sorting: During the selective coding phase, the memos related to authentication concepts were merged to form natural language test cases. These and the test smell memos were then *sorted* by concept to relate test cases to corresponding smells (*cf.* ③ in Figure 3.1).

Theoretical Coding: To connect related codes and construct a taxonomy of token authentication unit tests (*cf.* ④ in Figure 3.1), theoretical coding [77] was then performed. Concepts were integrated and organized using a hierarchical coding paradigm. High-level authentication concepts were structures into features such as *Token Manipulation* and *Login.* Concepts reflecting unique combinations of *context* and *action* related to each feature were structured as scenarios. For example, the *Token Authentication* feature has four scenarios: *token passed to authenticator*, *token passed to authenticating servlet filter via HTTP request*, *application supports multiple authentication types*, and *authentication succeeds.* In this paradigm, test cases within a feature can share a scenario and an expected outcome, but have a unique condition. For instance, 4 test cases in the *token passed to authenticator* scenario have the expected outcome *authentication fails.* Each test case reflects a unique condition that may cause this: (1) the token value is empty, (2) the token is expired, (3) one or more user details are missing, and (4) one or more user details are incorrect. These condition can cause failures or defects if not handled correctly, and all should be verified.This hierarchy was adapted from the concept of equivalence class partitions used in test planning [37, 145].

Write-up and Literature Review: The final two activities are the reporting of results and a formal review of related prior work. The literature review is conducted as the last step to avoid biasing the analysis. All findings from this study are reported Chapter 4 as part of a broader discussion of security unit testing in practice.

Appendix B

Mixed-Method Analysis of Developer's Q&A Posts

This appendix provides the **unabridged methodology** for the Q&A post study that was omitted from the research design overview in Chapter 3.2 for readability.

Data Collection and Processing

For this work, we mined Q&A posts from Stack Overflow (SO) and Security Stack Exchange (SSE). As both sites are on the same platform, we were able to mine all the post data using the Stack Exchange API and Stack Exchange Data Explorer.

<u>Mining Candidate Q&A Posts:</u> One challenge that studies using Q&A data must address is searching and filtering for relevant posts. The standard approach used in prior work is to perform tag and content-based filtering, where keywords for the study topic are used to identify relevant posts [8,124]. To build a candidate set of posts from Stack Overflow, we used Le *et al.* [114]'s dataset of 97,051 security-related posts as our initial search space. Their collection is the largest to-date, and was curated using a novel automated learning framework designed to mitigate limitations of keyword and supervised filtering approaches. It was not necessary to filter post content for security relevance when searching Security Stack Exchange. Instead, the criteria used to identify candidate posts from this site was relevance to *unit* and *integration testing*. A query for posts with *'unit'* and *'integrat'* in their title, tags, or body yielded a candidate set of 1,310 posts.

<u>Identifying Relevant Posts:</u> Candidate posts from both sources then underwent a second filtering. The goal was to extract the SO posts specifically related to security unit testing and confirm relevance of the SSE posts. The title, body, and tags of each post were scanned using an automated

script to look for the following keywords: *"unit-test","unittest", "unit-tests", "unittesting", "unit-testing", "unit test", "unit testing", "unit tests", "testing", "JUnit", "pyunit", "phpunit", "tdd", "bdd", "integration-test", "integration-testing", "integration test", "integration testing".* These terms were chosen after manual experimentation with tag-based search on Stack Overflow. The three 'xUnit' terms (*"JUnit", "pyunit", "phpunit"*) were included because they were recommended by Stack Overflow as popular tags related to "unit-test". Several variations of the same terms are also considered to ensure that matches in tags (which use '-' for multiple terms) *and* title/body text would be captured. Regular expressions were used for the search to ensure that sub-strings were not considered matches. When a post has one or more matches, a record was also created mapping the matches to the field (*e.g.* title) they were found in. After manually reviewing preliminary results, posts that *only* had "test" or "testing" keywords in one of the three fields were removed because they were usually not related to unit or integration testing.

Final Dataset: After discarding posts unrelated to security unit testing, the dataset consisted of 512 posts from Stack Overflow and 513 from Security Stack Exchange. As Security Stack Exchange focuses on information security, the small number of posts about testing at the unit and integration level from that source was expected. Combined, **the dataset used for this analysis consisted of 525 security unit testing Q&A posts**. Posts were created between 2008 and 2020, and 71% have an accepted answer. Table B.1 shows statistics for post scores, views, and answer counts.

Table B.1: Descriptive Statistics for Q&A Posts

	Score	***# Views***	***# Answers***
Mean	2.09	1,673.24	1.43
Median	1	688	1
Mode	0	110	1
Range	87	38,317	9
Minimum	-4	17	0
Maximum	83	38,334	9

Quantitative Analysis: Unsupervised Topic Modeling

The LDA topic modeling algorithm [32] was used to identify key discussion topics in the 525 security unit and integration testing Q&A posts mined from Stack Overflow and Security Stack Exchange.

Text Pre-Processing The following pre-processing pipeline was used to transform the textual contents of the Q&A posts for use in the topic modeling analysis:

Remove Links, Code Snippets and HTML Tags: The first step was removing these from the text, as this information is not relevant to the topic model [23]. To extract links, the text was searched for the `<a>` tag. Code snippets were identified by searching for the `<code>` tag, and similarly general HTML tags (*e.g.* `<p>`, `<a>`, `<br/>`) were also removed to reduce noise.

Remove Punctuation and Tokenize Text: After removing all punctuation characters, sentences were *tokenized* into lists of terms. The default approach is to isolate each individual word, but it is also possible to allow groups of words (*i.e.* n-grams) as a single term. During parameter tuning (described later in this Section), bigrams (*e.g.* "unit test") and trigrams (*e.g.* "Latent Dirichlet Allocation") were considered, but the optimal configuration used single terms.

Remove Stop Words: This step removes overly-common words and numbers, known as stop words, which are core elements of the text language (*e.g.*"the", "a","in") and thus do not provide topic content. Without this step, the high frequency of such terms would bias the model.

Normalize Terms: Terms were normalized to their canonical forms for consistency. This ensures that multiple forms of the same term (*e.g.* "configure","configuring", "configuration") are not considered unique terms. There are two approaches for normalization, lemmatization and stemming. Each produces a different result for the same word; for example "configuring" would be lemmatized as "configure" and stemmed to "configur". Both approaches were considered during tuning to account for influences on model performance and output readability; ultimately stemming was used.

Document Frequency Filtering: After normalization, frequency outliers were removed to reduce noise. This is based on the notion of removing the least and most frequently occurring terms, *i.e.* "extreme terms" that appear in less than X documents or in more that $Y\%$ of all documents. Several values for X and Y were tested during parameter tuning.

Final Transformation: Finally, the list of terms was transformed into a 'bag of words' vector [221].

Parameter Tuning: Like the 'number of topics' parameter, some choices made during pre-processing can influence the performance of the model. Thus, to determine the optimal configuration for this data, we compared the performance of LDA models with 246 unique value combinations for the following 'parameters': term frequency filtering, term normalization, n-grams, and number of topics. Table B.2 describes these parameters, the values tested for each, and the optimal values selected after evaluation. For this work the `gensim` [222] implementation of LDA was used; the default values for the alpha and beta (word and topic density) hyper-parameters were found to be sufficient in preliminary experiments. Standard coherence and perplexity metrics [137, 191] were used to identify the optimal configuration for the dataset; these measures indicate the performance of an LDA model in terms of its ability to identify distinct but understandable topics. The optimal configuration removed terms found in less than 50 posts or more than 60% of all posts, used stemming to normalize terms, considered only 1-grams, and generated 10 topics.

Table B.2: Parameters Tuned for Q&A Post Topic Model (LDA)

Name	**Description**	**Tested Values**	**Best**
Frequency Filter	Remove "extreme terms" that appear in below *B* documents or above *A*% of all documents	Below 10 & Above 25%, Below 20 & Above 40%, Below 20 & Above 50%, Below 20 & Above 60%, Below 50 & Above 40%, Below 50 & Above 50%, Below 50 & Above 60%, None (no filtering)	Below 50 & Above 60%
Transform	Normalize terms by trimming to their canonical form	Stemming, Lemmatization	Stemming
N-Grams	Length of multi-word (N) phrases to consider as 1 term	1, 2, 3	1
Topics	Number of topics	5, 10, 15, 20, 25	10

Topic Labeling: Topics generated by an LDA model are represented by a list of terms and their frequency (weight) within the topic. Labels were assigned for each topic manually; two authors independently reviewed terms for each topic to derive label sets. These were compared, merged and revised until full agreement was reached.

Qualitative Pain Point Analysis: Open Coding

Consistent with other studies applying qualitative methods to analyze Q&A posts [19,110,136,195], 50 posts were randomly selected from the 'top' 200 posts after sorting by number of views and score. Open coding [176] was applied for this analysis to guide the identification and synthesis of concepts related to the context, problem, and subject under test for each post. This was performed independently by two authors, who then discussed and merged the code sets collaboratively to ensure agreement. Codes were sorted, grouped, and combined as needed until a set of distinct pain points emerged that captured all of the challenges observed.