

ABSTRACT

ELDER, SARAH ELIZABETH. Applying Software Security Assessments in Realistic Contexts. (Under the direction of Laurie Williams and Marcelo D'Amorim).

CONTEXT: Security assessments are used as part of many decision-making processes, from prioritizing vulnerability detection and remediation resources to selecting third-party components to be incorporated in an organization's code.

PROBLEM: While a wide range of assessments exist, comparisons and evaluations of these assessments still have gaps.

OBJECTIVE: *The goal of this thesis is to assist practitioners with applying and interpreting security assessments through analyzing and enhancing different assessment techniques against real-world applications.*

APPROACH: We analyze vulnerability detection techniques and vulnerability exploitability assessments, to assist practitioners with prioritizing vulnerability detection and mitigation efforts. In our final work we examine how assessments can be used as part of determining whether to incorporate third-party components.

RESULTS: Our results suggest that increasing and improving activities for vulnerability detection and actively maintaining software can be beneficial in surfacing latent vulnerabilities. However, some techniques are not adopted and not expected to be adopted by all practitioners due to factors such as the ease or difficulty of applying the technique. Applying more techniques can improve the results, but practitioners often consider a lower baseline to be sufficient.

© Copyright 2024 by Sarah Elizabeth Elder

All Rights Reserved

Applying Software Security Assessments in Realistic Contexts

by
Sarah Elizabeth Elder

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2024

APPROVED BY:

William Enck

Bradley Reaves

Laurie Williams
Co-chair of Advisory Committee

Marcelo D'Amorim
Co-chair of Advisory Committee

DEDICATION

“All we have to decide is what to do with the time that is given us” – J.R.R. Tolkien

To my family.

BIOGRAPHY

Sarah Elder was born and raised in Newport News, Virginia. She graduated and earned an International Baccalaureate Diploma from Warwick High School. Sarah attended the University of Virginia for her undergraduate degree, where she double-majored in Computer Science and History. After a brief interlude working at IBM in the Washington DC area, she returned to earn her graduate degree at North Carolina State University. Her research interests include software testing and software security

ACKNOWLEDGEMENTS

Many thanks to everyone who was a part of this journey. To all the students from WSPR, SCI, and general denizens of 3228. Words cannot express my gratitude to the members of Realsearch and SWAT, past and present, who guided me in so many ways. Many thanks to Rahul, Maria, Pat, Chris, Akond, Rezvan, Nasif, Rayhanur, Nusrat, Sathvick, Ozgur, Nirav, Aishwarya, Monica, Setu, Imranur, Mahzabin, Sivana, Chris, Justin, Isaac, Abid, Ishrak, Ummay, and Uschwas.

Many thanks to Brennan, Brittany, Dr. Slankas, Dr. Trepte, and the many other research mentors I have had over the years. Thank you Dr. Heckman, for all the advice and encouragement over the years. Thanks to Dr. Enck and Dr. Reaves for being on my committee, and for their advice. Thank you to my advisors, Dr. Williams and Dr. D'Amorim. Thank you to Dr. Menzies and Dr. Singh for their help and advice. Thanks to Dr. Stolee for the reminders of the importance of being kind. Thanks to Dr. King, for the encouragement to keep going.

Thank you to my colleagues and mentors at IBM who inspired me to keep asking questions - Dave, David, Tandylyn, Carla, Matt, Andrew, Jesse, and Mary-Catherine. Additional thanks to the amazing Computer Science faculty at my alma mater, UVA, without whom I would not have returned to a STEM field since it was “all computers today, anyways”. Last but not least, I would like to acknowledge and thank my high school calculus teacher, Julyn Sheppard, who inspired countless students to reach for the stars.

The research presented in this dissertation has been supported (in part) by the National Science Foundation under Grants No. 1909516, 2026928, and 2207008. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

List of Tables	ix
List of Figures	xi
Chapter 1 INTRODUCTION	1
1.1 Software Measures & Metrology	2
1.2 Challenges in Realistic Software Systems	2
1.2.1 Size and Complexity	2
1.2.2 Software Supply Chain	3
1.3 Thesis Statement	4
1.4 Overview of Research Studies	4
1.4.1 Vulnerability Detection	5
1.4.2 Exploitability Assessment	6
1.4.3 Measurement Aggregation in Dependency Trees	6
1.5 Document Structure	7
Chapter 2 Key Concepts	8
2.1 Key Security Concepts	8
2.2 Key Software Architecture Concepts	10
2.2.1 Hypertext Transfer Protocol (HTTP)	11
2.2.2 Dependencies and Dependent Components	12
2.2.3 Package Management	13
2.3 Key Program Analysis and Testing Concepts	14
Chapter 3 Vulnerability Detection Methods	15
3.1 Previous Work by Austin et al.	18
3.2 Related Work	18
3.3 Vulnerability Detection Methods	20
3.3.1 Manual Methods	20
3.3.2 Automated Methods	21
3.4 System Under Test - OpenMRS	25
3.4.1 Why OpenMRS?	25
3.4.2 Technical Description	25
3.4.3 Security Practices at OpenMRS	26
3.5 Data Sources	26
3.5.1 Researcher Data	26
3.5.2 Student Data	26
3.5.3 Overview of Data Sources Per Research Question (RQ)	29
3.6 Methodology for RQ1.1 - Effectiveness	29
3.6.1 Data Collection	30
3.6.2 Data Analysis	40
3.7 Methodology for RQ1.2 - Efficiency	40
3.7.1 Data Collection	40

3.7.2	Data Analysis	41
3.8	Methodology for RQ1.3 - Other Factors	42
3.9	Equipment	42
3.10	Results	43
3.10.1	RQ1.1 - Method Effectiveness	43
3.10.2	RQ1.2 - Efficiency	57
3.10.3	RQ1.3 - Other Factors to Consider when Comparing Tools	59
3.11	Limitations	61
3.11.1	Conclusion Validity	61
3.11.2	Construct Validity	62
3.11.3	Internal Validity	62
3.11.4	External Validity	64
3.12	Discussion	65
3.12.1	Organizational Objectives	65
3.12.2	Resources to Consider	67
3.12.3	Implications for Evaluating Vulnerability Detection Methods	68
3.13	Conclusions	70
Chapter 4	Vulnerability Exploitability Assessments	71
4.1	Background and Related Work	73
4.1.1	Common Terms	73
4.1.2	Related Surveys	74
4.2	CVSS Components and Sub-Scores	74
4.2.1	Exploitability-related CVSS components	76
4.2.2	Differences in Exploitability-related subscores between CVSS v3 and CVSS v2	76
4.3	Methodology	77
4.3.1	Inclusion/Exclusion Criteria	78
4.3.2	Phase 1: Keyword Search with Active Learning	78
4.3.3	Phase 2: Snowballing	81
4.3.4	Organization and Categorization	81
4.4	Categories of Assessment Methods in Each Study	83
4.5	CVSS and CVSS-based metrics	84
4.5.1	How CVSS is calculated (manual/expertise-based)	84
4.5.2	Evaluations & Criticisms	87
4.5.3	Proposed Changes/Improvements	92
4.6	Deterministic	94
4.6.1	Program-State-Based	94
4.6.2	Network-System-State-Based	100
4.6.3	Attacker-Based	100
4.7	Probabilistic Assessments: Learning Models (LM)	101
4.7.1	What is used as the ground truth labels for Training / Testing (GT)?	102
4.7.2	What categories of information are commonly used as features (FT)	107
4.8	Other Probabilistic Assessments (Non-LM)	111
4.8.1	Exploit Availability Distribution Function from Frei et al.	111

4.8.2	Developing Prioritization Rules based on Exploit Likelihood Analysis . . .	113
4.9	Limitations	113
4.10	Discussion	114
4.10.1	Temporal Trends	114
4.10.2	Aligning with Industry values for vulnerability prioritization	115
4.10.3	Research Gaps and Future Directions	116
4.11	Conclusion	117
Chapter 5	OpenSSF Scorecard Aggregation	119
5.1	OpenSSF Scorecard	121
5.1.1	The Checks	121
5.1.2	Calculating the Overall Scorecard Score	125
5.1.3	Existing Aggregation Across the Dependency Tree for <i>Vulnerabilities</i> . . .	125
5.1.4	The OpenSSF Scorecard Tool	126
5.2	Go (The Programming Language)	126
5.2.1	Package Management in Go	126
5.2.2	Why Go?	126
5.3	Related Work	127
5.3.1	Understanding Third-Party Component Selection	127
5.3.2	Examining the relationship between package depth and measurement value	130
5.3.3	Prior work on OpenSSF Scorecard	131
5.3.4	Prior work on the Go Ecosystem	132
5.4	Preliminary Interviews	133
5.4.1	Interview Process	133
5.4.2	Findings	134
5.5	Data	135
5.5.1	Go Packages	135
5.5.2	OpenSSF Scorecard Data	137
5.5.3	Size Data	140
5.5.4	Vulnerability Data	140
5.6	Methodology	140
5.6.1	Data Analysis	140
5.6.2	Survey	142
5.7	Aggregation Methods	143
5.8	Results	144
5.8.1	RQ3.1 - Data Analysis	145
5.8.2	RQ3.1 - Survey	148
5.9	Limitations	151
5.9.1	Conclusion Validity	151
5.9.2	Construct Validity	152
5.9.3	Internal Validity	152
5.9.4	External Validity	153
5.10	Ethics	153
5.11	Discussion	153

5.11.1	Should Metrics aggregate information from throughout the package's dependency tree?	154
5.11.2	Improvements and Future Work	154
5.11.3	Other Implications	155
5.12	Conclusion	155
Chapter 6	Summary	157
6.1	Lessons Learned	157
6.2	Future Work	158
6.3	Conclusion	158
References	159
APPENDICES	186
Appendix A	Acronyms	187
Appendix B	Vulnerability Detection Technique Comparison - Replication Details	189
B.1	Appendix - Automated Technique CWEs	189
B.2	Appendix - Student Experience Questionnaire	199
B.3	Appendix - Student Assignments	200
B.3.1	Project Part 1	200
B.3.2	Project Part 2	202
B.3.3	Project Part 3	204
B.3.4	Project Part 4	207
B.4	Appendix - Equipment Specifications	209
Appendix C	Vulnerability Detection Technique Comparison - All CWEs Table . .	211
Appendix D	OpenSSF Measurement Method Details	217
Appendix E	Survey Example	225
Appendix F	Survey Results for Other Checks	237

LIST OF TABLES

Table 1.1	Research Study Index for this Dissertation	5
Table 3.1	Data Sources	
	R = Researchers; S = Students	30
Table 3.2	Results Summary	43
Table 3.3	Total Alerts (Tot. Alrt.), False Positives (FP), and Precision (Prec) for the Current Study Compared with Austin et al.	
	M indicates OpenMRS, E indicates OpenEMR, T indicates Tolven, and P indicates PatientOS	46
Table 3.4	DAST Alerts	48
Table 3.5	Vulnerability Counts	50
Table 3.6	Vulnerability Count Per Detection Method	52
Table 3.7	Vulnerability Type Comparison with Austin Study	55
Table 3.9	VpH Compared to Austin et al.	58
Table 3.8	Games-Howell t-test of Efficiency Scores	58
Table 4.1	Related Surveys	75
Table 4.2	Inclusion/Exclusion Criteria	79
Table 4.3	Databases Examined	80
Table 4.4	Exploitability Assessment Methods Proposed and/or Evaluated in Each Study	
	M - assessments from this category are the main focus of the study; C - assessments from this category are compared against the main category	85
Table 4.5	Studies on Automated, Rule-Based Program Analysis for Assessing Ex- ploitability “NA” = “Not Applicable”; “NP” = “Not Provided”	96
Table 4.6	Learning Model(s) Examined in Each Study	118
Table 5.1	OpenSSF Checks	121
Table 5.2	Go Dataset Summary	137
Table 5.3	OpenSSF Checks Data Summary	139
Table 5.4	Relationship of Metrics to Average Depth	146
Table 5.5	Effect of Adjusted Scores	146
Table 5.6	Linear Regression of Checks and Adjusted Values against Active Vulnera- bility Count	147
Table 5.7	Survey Results	150
Table A.1	A summary of acronyms used in alphabetical order.	187
Table B.1	CWEs covered in the rules implemented by automated techniques	189
Table C.1	CWEs associated with <i>more severe</i> Vulnerabilities	211
Table C.2	Low Severity Vulnerability CWEs	215
Table D.1	OpenSSF Check Details	218

Table F.1 Survey Results for Checks with Limited Data 237

LIST OF FIGURES

Figure 2.1	Example HTTP message	11
Figure 2.2	Example HTTP Sequence	12
Figure 2.3	Example Dependency Tree Each node “depends on” (i.e. explicitly references) the nodes that it points to. E.g. d_1 “depends on” d_3	13
Figure 3.1	Applying Manual Test Methods (based on ISO/IEC/IEEE 29119-1)	20
Figure 3.2	Example SMPT Test Case	22
Figure 3.3	Applying Tool-Based Methods	22
Figure 3.4	Message from a Malformed Input (Test Case) Produced by a DAST Tool .	24
Figure 3.5	Industry Security Experience of Students	27
Figure 3.6	Data Collection for RQ1.1	31
Figure 3.7	Data Collection for RQ1.2	40
Figure 3.8	Vulnerability Detection Method Efficiency	57
Figure 4.1	Paper Collection Process	78
Figure 4.2	Graph Based on Metric Taxonomy in Pendleton et al. [221], with relevant attributes highlighted	82
Figure 5.1	Example Dependency Tree from Interviews	133

CHAPTER

1

INTRODUCTION

Security assessment is essential to identifying and reducing security risk. The United Kingdom (UK) Office of National Statistics 2024 Cyber Security Breaches Survey [70] found that only “31% of businesses ... have undertaken cyber security risk assessments in the last year - rising to 63% of medium businesses and 72% of large businesses”. The response was lower for other forms of security assessment. For example, only 17% of businesses had “a cyber security vulnerability audit”. Although most businesses were not assessing security proactively, half of the businesses who responded to the survey reported “some form of cyber security breach or attack in the last 12 months” [70]. Lack of effective security assessment is not limited to the UK. For example, in a survey of a wide range of StackOverflow and Github Users, de la Mora and Nadi [66] found that, on average, practitioners find measures helpful for determining which libraries to select. However, studies have found that current tools for selecting third-party libraries lack flexibility in the types of measures that they allow practitioners to use to assess the libraries [189, 157, 164]. Existing tools may also be missing necessary measures [164, 189].

An assessment [134, 93, 4] is a set of criteria that can be applied to a system to make a decision, such as whether the system is ready to be deployed. In *software security assessments*, the criteria are applied to a *software* system and intended to provide information about the system’s *security* to aid in decision-making. These criteria may be qualitative or quantitative. Assessment criteria, particularly quantitative criteria, may be applied using one or more measures.

The goal of this thesis is to assist practitioners with applying and interpreting security assessments through analyzing and enhancing different assessment techniques against real-world applications.

1.1 Software Measures & Metrology

Understanding and evaluating software security measures is part of the discipline of *Software Metrology* [4]. Metrology is the study of how we map entities and their attributes in the real world to numeric or symbolic values [93, 4]. In Software Metrology, the entities and attributes being mapped pertain to software. The mappings, or *measures*, help practitioners to understand and, if possible, improve the processes and situations they encounter [93, 4].

An important distinction in Metrology is between the concept that a measure is trying to capture (sometimes referred to as a *measurand*), the measurement method, and measurement results [4]. The *concept the measure is trying to capture* is a high-level idea such as the security risk that may be posed by a particular software component. The *measurement method* is how the measurand is captured, such as the number of vulnerabilities identified using static analysis tools. The *measurement results* are the values that are output when the measurement method is applied to a particular entity, such as 20 critical vulnerabilities. Whether a particular set of *measurement results* are useful for practitioners depends on the *measurement method*.

1.2 Challenges in Realistic Software Systems

Many of the recurring challenges with applying security assessments to realistic systems involve the size and complexity of the systems being assessed. We provide an overview of this challenge in Section 1.2.1. As we will see in our detailed discussion of the work so far in Chapters 3 and 4, size and complexity of the system being assessed are important to consider when determining appropriate measurement methods. Challenges around size and complexity become further complicated by the concept of the Software Supply Chain. We provide a brief overview of the challenges around Software Supply Chain in Section 1.2.2. The Software Supply Chain is at the center of our final work in Chapter 5 .

1.2.1 Size and Complexity

Security assessment methods often perform differently when applied to larger, more complex systems than when applied to smaller, less complex systems. The United States (US) National Institute of Standards and Technology's (NIST's) most recent Static Analysis Tool Exposition

(SATE) compares and evaluates Static Analysis Security Testing (SAST) tools. Their most recent analysis reported that “low code complexity was the prominent driver for tool success” [67]. The same report also found that some tool-makers found larger systems difficult to assess as part of the evaluation, preferring systems with less than 500,000 lines of code (LoC)¹. Similarly, while vulnerability exploitability assessment systems which use trace analysis may take seconds or minutes to analyze a single vulnerability [120, 312, 285, 132, 230, 231, 232], they can take hours when applied to an entire system [232]. This hours-long process makes trace analysis-based exploitability assessments impractical for some scenarios, e.g. for routine analysis of the entire system before a major change to the source code as part of a Continuous Integration / Continuous Deployment (CI/CD) pipeline.

Where possible, evaluation of software security assessment methods should ensure that the size and complexity of the software used in the evaluation is comparable to the systems where the assessments would be deployed by practitioners. Additionally, measures, such as vulnerability density, which incorporate size into the measure itself [6] may be useful.

1.2.2 Software Supply Chain

A factor that can influence the size and complexity of software is the Software Supply Chain. In contemporary software development, developers rarely encounter a task that no one has attempted before. For many ecosystems, from Java to Go to node.js, a plethora of open-source components are available to perform a wide range of software tasks. These open-source components are also referred to as “libraries” or “packages”. Developers can use these components in lieu of writing the code themselves.

The components used by a particular piece of software are often referred to as its “dependencies” [219]. Those dependencies can have dependencies of their own, which can have dependencies of their own, etc. This entire network of dependencies is often referred to as the “dependency tree” [219].

A dependency tree is an aspect of the supply chain for a particular piece of software. NIST defines a supply chain as a “linked set of resources and processes between and among multiple tiers of organizations, each of which is an acquirer, that begins with the sourcing of products and services and extends through their life cycle.” [146]. The entity acquiring a particular resource or process is sometimes referred to as a “consumer” [209]. The entire Software Supply Chain may go well beyond the dependency tree, to include hardware and other elements of a software system. For this thesis, we focus on the software elements of the Software Supply Chain, i.e. its dependency tree.

¹For context, large and complex systems can have millions of LoC [193]

When a consumer selects an open-source component to include as a dependency, they inherently increase the size and complexity of the project. However, in many software development frameworks, the full code of software dependencies is not stored in the same repository as the dependent package. When code and other artifacts for dependencies are not packaged with a source repository, they may not be accounted for in some software assessments. For example, the CLOC tool² for measuring Lines of Code (LoC), a common software size measure, counts the lines of code in a particular file or directory which may not include the lines of code of dependencies.

Challenges, such as identifying files that may be stored separately, are primarily technical. The more fundamental challenge is understanding whether dependency information *should* be assessed alongside the dependent, and if so - how the information should be aggregated. As may be implied from the NIST definition of “supply chain”, dependencies can come from different organizations. If a problem occurs due to a dependency managed by a different organization from the consumer, the consumer may see the problem as not their responsibility [219, 287]. If consumers *do* want to fix the problem themselves, it can involve additional complications, such as creating and maintaining a new version of the original dependency [287]. Hence, following our previous example, consumers may want the LoC for their own code to be assessed separately from the LoC of their dependencies.

1.3 Thesis Statement

The thesis statement of this work is the following:

Combining information from multiple software assessment methods and from throughout a project’s dependency tree can assist practitioners in making security decisions.

1.4 Overview of Research Studies

In this section, we briefly summarize the three studies supporting our thesis. We begin each summary with the high-level research question the study addresses. The mapping between the summaries in this section and the chapters containing additional details is shown in Table 1.1.

²<https://github.com/AlDanial/cloc>

Table 1.1: Research Study Index for this Dissertation

	<i>Study</i>	<i>Summary</i>	<i>Details</i>
Vulnerability Detection		Sec. 1.4.1	Ch. 3
Exploitability Assessment		Sec. 1.4.2	Ch. 4
Dependency Analysis		Sec. 1.4.3	Ch. 5

1.4.1 Vulnerability Detection

What are the relative advantages and disadvantages of vulnerability detection methods, in terms of effectiveness and efficiency?

Determining the number, location, and other attributes of the software vulnerabilities is key to triaging vulnerabilities appropriately and reducing software security risk. However, applying vulnerability detection methods to measure number, location, and other attributes is one of many tasks using the limited resources of a software project. A decade ago, a comparison of vulnerability detection methods by Austin et al. [23, 22] illustrated how that “One technique is not enough”. Different methods will produce different results. Vulnerability detection approaches, such as static application security testing (SAST), have improved over the past 10 years [68]. However, prior evaluations of vulnerability detection methods have primarily used smaller systems for analysis, and not compared across multiple types of methods. A new comparison was needed to understand the relative performance of modern vulnerability detection methods on a realistic system.

We examine the four different categories of vulnerability detection methods used by Austin et al.: systematic manual penetration testing (SMPT), exploratory manual penetration testing (EMPT), dynamic application security testing (DAST), and static application security testing (SAST). We applied each method to an open-source medical records system, OpenMRS [205]. As a result, we identified 329 vulnerabilities of medium or high severity. At the time we began our analysis in 2020, the NVD contained less than 10 vulnerabilities in OpenMRS [198]. In our study, we found the most vulnerabilities using SAST. However, EMPT found more severe vulnerabilities. With each method, we found unique vulnerabilities not found using the other methods. These widely differing vulnerability counts illustrate the necessity of ensuring that the measurement method for vulnerability-count-based measures reflects the entity it is intended to represent. For example, if comparing vulnerability counts across multiple systems - practitioners should ensure that the vulnerability counts were gathered using the same method to ensure they are capturing the intended differences between target systems rather than differences in collection method (e.g. SAST or DAST). We discuss this study in greater detail in Chapter 3.

1.4.2 Exploitability Assessment

How is the exploitability of software vulnerabilities assessed?

Knowing the exploitability and severity of software vulnerabilities helps practitioners prioritize vulnerability mitigation efforts. One frequently used definition of exploitability comes from the Common Vulnerability Scoring System (CVSS) [101, 11, 304, 302, 300, 246, 315, 80, 113], a vulnerability severity assessment framework used by organizations such as the U.S. National Vulnerability Database (NVD). CVSS defines exploitability as the “ease and technical means” by which an attacker can use a vulnerability to achieve their goals [59]. However, researchers have proposed and evaluated many different exploitability assessment methods, some of which do not agree with CVSS [33]. In our second study, *The goal of this research is to assist practitioners and researchers in understanding existing methods for assessing vulnerability exploitability through a survey of exploitability assessment literature*. In our survey, we identify three approaches for exploitability assessment that have been examined in the literature: CVSS-based, Deterministic, and Probabilistic Systems.

The applicability of different exploitability assessment methods depends on the resources and priorities of the organization using the method. As noted in Section 1.2.1, methods relying on program analysis tools for control and data-flow tracing may not be usable for some scenarios, such as continuous integration /continuous deployment (CI/CD) pipelines, due to the length of time required to run [232] and their use was limited to less frequent, in-depth analyses. However, some of the data required to build learning models which are common in *Probabilistic* approaches may be difficult for smaller organizations to collect [259, 34]. The same, smaller organizations may have greater ability able to run a program analysis tool against their own code, even if the assessment takes hours to complete. We discuss our survey in greater detail in Chapter 4.

1.4.3 Measurement Aggregation in Dependency Trees

How should security scores from throughout a package's dependency tree be aggregated?

Practitioners rely on high-level measures that are readily accessible to evaluate the risk of incorporating an open-source software component [287]. However, incorporating a software component does not always involve just one component. Each component may depend on other components, which in turn depend on other components, creating a network of dependencies often referred to as a dependency tree. While some work has been done to examine how vulnerability-based measures from throughout the dependency tree should be aggregated [310, 219], less work has been done to examine how other measures should be aggregated. Given the rarity of software vulnerabilities and the reactive (not proactive) nature of

vulnerability-based security assessments, it is important to understand how other assessments, such as maintenance activity, should be aggregated.

The goal of this research is to aid developers and software architects in making good component choices through an analysis of the relevance and utility of security measures (OpenSSF metrics) of the entire dependency tree associated with a package.

To address this goal, we developed a set of aggregation approaches for OpenSSF Scorecard checks based on a preliminary set of semi-structured interviews. We then validated those approaches in a practitioner survey. We also performed a statistical analysis of the Go ecosystem to understand the effect of aggregating measures from throughout a project's dependency tree, and how aggregation would impact the score's relationship to the number of vulnerabilities in a package. We discuss this work in greater detail in Chapter 5.

We found that for some checks, such as the check determining the extent to which Github commits are Code Reviewed, aggregating information may not be helpful or desirable for practitioners. For other checks, such as the check analyzing the level of Branch Protection on the package's Github Repository, aggregating information from throughout the package's dependency tree may surface information that would not otherwise be accessible and *is* preferred by practitioners. Additionally, the survey responses suggest it may be preferable to present information from the package's dependency tree alongside the original score for the root node.

1.5 Document Structure

The remainder of this document is organized as follows. In Chapter 2, we provide background on key concepts and terms which will be used throughout the document. In Chapter 3, we present our study of vulnerability detection methods. In Chapter 4, we present our exploitability assessment survey. In Chapter 5, we discuss the final work on aggregating information from throughout the dependency tree of a third-party library.

CHAPTER

2

KEY CONCEPTS

In this section, we explain key, cross-cutting concepts that may be useful in understanding the research described in this dissertation. In Section 2.1 we explain key concepts from Software Security. In Section 2.2 we explain key concepts from Software Architecture. Finally, in Section 2.3 we explain key concepts that are frequently used to characterize Program Analysis methods.

2.1 Key Security Concepts

In this section we define and describe key security concepts which may be referenced throughout the document.

Vulnerability

We use the definition of vulnerability from the U.S. National Vulnerability Database¹. A vulnerability is “*A weakness in the computational logic (e.g., code) found in software and hardware components that, when exploited, results in a negative impact to confidentiality, integrity, or availability.*” [198].

¹<https://nvd.nist.gov/vuln>

Common Vulnerability Scoring System (CVSS)

CVSS is a standardized vulnerability assessment framework in which an overall severity score can be computed based on sub-scores from three metric groups: Base, Temporal, and Environmental [59]. The U.S. National Vulnerability Database (NVD) [197] and vendors, such as Oracle², use the Base CVSS score to communicate the severity of a vulnerability in a software system. CVSS is owned and managed by the Forum of Incident Response and Security Teams (FIRST) [59], who organize the CVSS Special Interest Group (SIG) to maintain and improve the CVSS specification. CVSS has gone through multiple iterations and versions.

Common Vulnerability Enumeration (CVE)

The acronym “CVE” is commonly used to refer to vulnerabilities that are part of the “Common Vulnerability Enumeration” (CVE) List [56]. The CVE List is currently managed and maintained through the CVE program. The CVE list is used by a wide range of tools and processes, including the U.S. National Vulnerability Database [198, 34].

U.S. National Vulnerability Database (NVD)

The U.S. National Vulnerability Database (NVD) is “the U.S. government repository of standards based vulnerability management data” [194]. The NVD adds CVSS scores to each vulnerability in the CVE list, and provides a public API for accessing the CVE list [194]. The NVD is managed by the U.S. National Institute for Standards and Technology (NIST).

Common Weakness Enumeration (CWE)

Per the CWE website, “CWE is a community-developed list of software and hardware weakness types.”[184]. Many security tools, such as the OWASP Application Security Verification Standard (ASVS) and most vulnerability detection tools, use CWEs to identify the types of vulnerabilities relevant to a security requirement, test case, or tool alert. We use the CWE list in this paper to standardize and compare the vulnerability types found by different vulnerability detection method.

OWASP Top Ten

The OWASP Top Ten is a regularly updated list of “the most critical security risks to web applications.”[215]. The OWASP Top Ten categories and ranking are developed by security

²<https://www.oracle.com/security-alerts/cvssscoringsystem.html>

experts based on the incidence and severity of vulnerabilities associated with different CWEs. A mapping [182] from CWE to OWASP Top Ten allows vulnerabilities to be mapped to the OWASP Top Ten categories via CWEs. We use the OWASP Top Ten in this paper to summarize the types of vulnerability found and to understand the relative severity of the vulnerabilities found. The latest (2021) Top Ten, which were used in our analysis, are: A01 - Broken Access Control, A02 - Cryptographic Failures, A03 - Injection, A04 - Insecure Design, A05 - Security Misconfiguration, A06 - Vulnerable and Outdated Components, A07 - Identification and Authentication Failures, A08 - Software and Data Integrity Failures, A09 - Security Logging and Monitoring Failures, and A10 - Server-Side Request Forgery (SSRF). Additional information on the OWASP Top Ten may be found at <https://owasp.org/Top10/>.

OWASP Application Security Verification Standard (ASVS)

OWASP ASVS is an open standard for web application security verification. ASVS provides a high-level set of “*requirements or tests that can be used by architects, developers, testers, security professionals, tool vendors, and consumers to define, build, test and verify secure applications*” [278]. In ASVS, each requirement or test is referred to as a “control” which is not specific to a particular SUT. Each ASVS control is mapped to a CWE type. We used OWASP ASVS version 4.0.1 released in March 2019³, which was the current version when we began collecting data in Spring 2020. ASVS has three levels of requirements. If a requirement falls within a level, it also falls within higher levels. ASVS describes Level 1 as “*the bare minimum that any application should strive for*” [278].

2.2 Key Software Architecture Concepts

In this section, we provide key concepts and information around three topics. First, in Section 2.2.1, we explain some technical details of HTTP are helpful in understanding different ways of assessing web applications. We then explain concepts around Software Dependencies and the Dependency Tree in Section 2.2.2, which are key to understanding our work in Chapter 5, but are broadly applicable to all software systems. Similarly, we explain key concepts relating to package management which may be useful in understanding the system under test in Chapter 3, as well as our work with package dependency trees in Chapter 5.

³<https://github.com/OWASP/ASVS/tree/v4.0.1>

2.2.1 Hypertext Transfer Protocol (HTTP)

HTTP is the set of rules used to communicate with web applications, such as the SUT for our case study. When a user interacts with the web application through a browser HTTP messages are created by the web browser [186] and sent to the application. Each HTTP message requests that some action be applied to a particular resource in the application [94]. Resources are identified through a Uniform Resource Identifier (URI). The action and URI are indicated in the header of an HTTP message. The HTTP request in Figure 2.1 is a message sent to the OpenMRS application to log into the application. In Figure 2.1, the browser is requesting that the information in the message be POSTed to the URI `http://127.0.0.1:8080/openmrs/login.htm`. The remainder of the HTTP message contains a *representation* [94] of the requested resource. The application server software then responds to the request with an HTTP message.

```
POST http://127.0.0.1:8080/openmrs/login.htm HTTP/1.1
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:99.0) Firefox/99.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Content-Type: application/x-www-form-urlencoded
Content-Length: 170
Origin: https://127.0.0.1:8800
Connection: keep-alive
Referer: https://127.0.0.1:8800/openmrs/login.htm
Cookie: JSESSIONID=owczzkvskvc8n4580psmvbb
Host: 127.0.0.1:8800

username=admin&password=Admin123&sessionLocation=0&redirectUrl=/openmrs/referenceapplication/home.page
```

Figure 2.1: Example HTTP message

As an example of how HTTP messages are passed between the browser and the application server is shown in Figure 2.2. In this example, the user logs into the system. The user begins by opening a browser and enters the application login URL `http://127.0.0.1:8080/openmrs/login.htm`. The browser creates and sends an HTTP GET request for the initial login screen (shown in line 01 in Figure 2.2). The server's response to the GET request contains information about the application page (the description and size of the response are also indicated in line 01). In a simple application, the entire web page may be included in the first response. In a more complex web application, the response may indicate that the browser needs to request additional resources, and the browser will use more GET requests to obtain the additional resources (lines 02-14 in Figure 2.2). Once all the resources have been received and the login page rendered, the user enters required information such as the username and password and submits the information to the browser. The browser then creates and sends a POST request (line 15 in

Figure 2.2), sending the login information to the server. The server's response to the POST request tells the browser where to start accessing resources based on whether the login was successful, and the browser creates one or more GET requests (lines 16-17 in Figure 2.2) to obtain the necessary resources. Although the user has only performed two actions - entering the URL and submitting login information, this series of interactions with the OpenMRS application results in 17 different requests being sent from the browser to the server.

	<i>Request</i>		<i>Response</i>	
	Method	URL - http://127.0.0.1:8080/openmrs	Desc.	Size
01	GET	/login.htm	OK	7924
02	GET	/ms/uiframework/resource/uicommons/scripts/jquery-1.12.4.min.js	OK	97163
03	GET	/ms/uiframework/resource/uicommons/scripts/jquery.simplemodal.1.4.4.min.js	OK	9769
04	GET	/ms/uiframework/resource/uicommons/styles/styleguide/jquery-ui-1.9.2.custom.min.css	OK	68264
05	GET	/ms/uiframework/resource/uicommons/scripts/jquery.toastmessage.js	OK	6390
06	GET	/ms/uiframework/resource/uicommons/scripts/emr.js	OK	16365
07	GET	/ms/uiframework/resource/referenceapplication/styles/login.css	OK	210
08	GET	/ms/uiframework/resource/uicommons/styles/styleguide/jquery.toastmessage.css	OK	4194
09	GET	/ms/uiframework/resource/uicommons/scripts/underscore-min.js	OK	13450
10	GET	/ms/uiframework/resource/uicommons/scripts/knockout-2.2.1.js	OK	90242
11	GET	/ms/uiframework/resource/referenceapplication/styles/referenceapplication.css	OK	164423
12	GET	/ms/uiframework/resource/uicommons/scripts/jquery-ui-1.9.2.custom.min.js	OK	236825
13	GET	/ms/uiframework/resource/referenceapplication/fonts/opensans-regular-webfont.ttf	OK	42596
14	GET	/ms/uiframework/resource/referenceapplication/fonts/fontawesome-webfont.woff?v=3.0.1	OK	29380
15	POST	/login.htm	Found	0
16	GET	/referenceapplication/home.page	OK	13326
17	GET	/ms/uiframework/resource/appui/styles/header.css	OK	289

Highlighted lines are directly triggered by user input

Figure 2.2: Example HTTP Sequence

2.2.2 Dependencies and Dependent Components

A software **dependency** is a piece of code which can be re-used as part of another piece of code, e.g. via an explicit reference to an identifier for the code being re-used [264, 219, 187]. The term “dependency” is typically used to describe the independent components or libraries which are required for a particular piece of software to function [264]. A **dependent** component, on the other hand, is a package that depends on the particular dependency. Other key terms relating to software dependencies include:

Direct Dependency A direct dependency is explicitly referenced from the main source code of the software being evaluated. [220, 219, 314, 63]

Transitive Dependency A transitive dependency is referenced either by a direct dependency or by another transitive dependency, but *not* by the main source code of the component being evaluated [219, 36, 63]

Dependency Tree the hierarchical graph of all components connected to a piece of software via dependency relationships is referred to as the “Dependency Tree” for that software [200].

Figure 2.3 illustrates the relationships that comprise a dependency tree. This example is abbreviated for illustration purposes. In production software, dependency trees can include dozens, or even hundreds of dependencies [314]. In the example in Figure 2.3, d_1 and d_2 are *direct* dependencies of the main source code r , while $d_3, d_4, d_5, d_6, d_7, d_8,$ and d_9 are *transitive* dependencies of r . Similarly, d_3 is a *direct* dependency of d_1 . Figure 2.3 shows no *transitive* dependencies for d_1 .

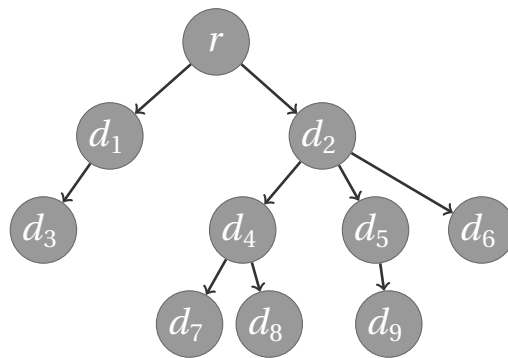


Figure 2.3: Example Dependency Tree

Each node “depends on” (i.e. explicitly references) the nodes that it points to. E.g. d_1 “depends on” d_3

2.2.3 Package Management

The distribution of software components is sometimes referred to as *package management*. Key terms around package management include:

Package Code, documentation, and metadata that is logically grouped together for distribution, e.g. as a third party component, is referred to as a *package* [127].

Package Manager Software that is used to automatically install, update, and configure packages that are used as dependencies is referred to as a *package manager* [187, 158].

Registry In the context of package management, the location that stores authoritative information about a package, such as where the package may be downloaded from and which version is the latest, is referred to as the *registry* [187].

2.3 Key Program Analysis and Testing Concepts

In this section, we explain common characteristics of program analysis including whether the methods use *automated* vs *manual* analysis, *systematic* vs *exploratory* analysis, *dynamic* vs *static* analysis, and *source code* analysis. In Chapter 3 we use these characteristics to explain the vulnerability detection approaches. This terminology will again be useful in Chapter 4 in understanding the program-analysis-based approaches for assessing exploitability.

Automated vs Manual analysis: Some methods are based on *automated* analysis performed by a tool. Manual effort may be required to use automated program analysis tools, such as to review and remove false positives from the tool results [263]. However, in this document we will refer to a method as a *manual* method if no automated tool is needed, to distinguish them from *automated* tool-based methods

Systematic vs Exploratory analysis: *Systematic* analysis is performed in a very prescriptive, methodical manner; in contrast with *exploratory* analysis which is less formally planned. For example, ISO 29119 [135] defines exploratory testing, a form of *exploratory* analysis, as “experience-based testing in which the [analyst] spontaneously designs and executes tests based on the [analyst]’s existing relevant knowledge, prior exploration of the test item ..., and heuristic ‘rules of thumb’ regarding common software behaviours and types of failure”. The concepts of *systematic* and *exploratory* analysis primarily apply to *manual* analysis. Whether an automated tool has knowledge and experience is outside the scope of this paper.

Dynamic vs Static analysis: Dynamic analysis is performed against actively running software [135]. Static analysis is performed on static artifacts such as source code or binaries, where the software is not actively running [135].

Source code analysis: Source code analysis is any form of analysis that reviews the source code of the SUT. While source code analysis is sometimes used as a synonym of static analysis [172, 23], static analysis can include analyzing binaries and other artifacts that are not source code. Analysis that does not have access to source code is sometimes referred to as “black box” analysis.

CHAPTER

3

VULNERABILITY DETECTION METHODS

Detecting software vulnerabilities efficiently and effectively is necessary to reduce the risk that hackers will exploit vulnerabilities before developers can find and patch them. However, as noted by Alomar et al. [15], security teams often struggle to justify the costs of vulnerability detection and other vulnerability management activities. The need to improve vulnerability detection efforts while not expending unnecessary resources is highlighted in Section 7 of U.S. Presidential Executive Order 14028, which begins “*The Federal Government shall employ all appropriate resources and authorities to maximize the early detection of cybersecurity vulnerabilities...*” [86]. The executive order also emphasizes the need for improved evaluation of security practices, including vulnerability detection.

The goal of this research is to assist managers and other decision-makers in making informed choices about the use of software vulnerability detection methods through an empirical study of the efficiency and effectiveness of four methods on a Java-based web application. We perform a theoretical replication¹ of work done by Austin et al. [22, 23]. Since the original Austin et al. work in 2011, the vulnerability detection landscape has changed - from the applications being tested to the vulnerabilities found [214, 213, 212, 211]. For example, the number of vulnerabilities in the United States National Vulnerability Database assigned to Cross-Site Scripting (XSS) has

¹A theoretical replication seeks to investigate the scope of the underlying theory, e.g. by redesigning the study for a different target population, or by testing a variant of the original hypothesis [169]

increased faster than the prevalence of other vulnerability types such as Code Injection [199]. Our methodology and findings may also be useful to future evaluations of new vulnerability detection methods.

We examined the 4 vulnerability detection methods from Austin et al. [22, 23].

- **Systematic Manual Penetration Testing (SMPT)**: the analyst manually and systematically develops, documents, then executes test cases which verify the security objectives of the System Under Test (SUT) [261, 22, 260, 23]
- **Exploratory Manual Penetration Testing (EMPT)**: the analyst “spontaneously designs and executes tests” [135], searching for vulnerabilities.
- **Dynamic Application Security Testing (DAST)**: automatic tools generate and run tests based on security principles, without access to source code[249].
- **Static Application Security Testing (SAST)**: automatic tools scan source code for patterns that indicate vulnerabilities [54, 118, 248].

These four methods can be applied during and after software implementation. Applying vulnerability detection during and after software implementation is more common in many industry settings [54] compared to methods which focus on earlier phases of software development such as requirements and design. We used an industry standard, the Open Web Application Security Project’s Application Security Verification Standard (OWASP ASVS), to systematically develop test cases for SMPT. The two DAST tools and three SAST tools are currently used in industry settings. Two of these tools, the OWASP Zed Attack Proxy (OWASP ZAP)² DAST tool and the Sonarqube³ SAST tool, are open source. The other tools, which we will refer to as DAST-2, SAST-2, and SAST-3 are proprietary.

We applied each method to OpenMRS (<https://openmrs.org/>), a large open-source medical records system used in medical research and clinical settings throughout the world. OpenMRS is a web application written in Java and JavaScript, containing 3,985,596 lines of code⁴. We consider our work to be a case study since we only examine a single System Under Test (SUT).

Although OpenMRS is comparable in size to other industry systems [277, 85] we know of no other study applying multiple different vulnerability detection methods to a system this large. The only previous work comparing as many different types of vulnerability detection methods that we are aware of is the original study by Austin et al. [22, 23]. The systems in Austin et al.’s

²<https://owasp.org/www-project-zap/>

³<https://www.sonarqube.org/>

⁴as measured by CLOC v1.74 (<https://github.com/AlDanial/cloc>)

work were less than 500,000 lines of code. Collecting data for our current study on a system with millions of lines of code required a team of four graduate students a combined eleven months of full-time work and twenty months of part-time work; four months part-time work from an undergraduate student; and the results of assignments from a large graduate-level software security course. Our experiences in structuring the software security course have been reported previously in Elder et al.[83].

Our overall research question is *What are the relative advantages and disadvantages of vulnerability detection methods, in terms of effectiveness and efficiency?* which we break down into the following research sub-questions:

- RQ1.1: What is the effectiveness, in terms of number and type of vulnerabilities, for each method?
- RQ1.2: How does the reported efficiency in terms of vulnerabilities per hour differ across method?

As part of the software security course, students were asked to discuss and compare the four methods. Two researchers performed qualitative analysis on the answers, addressing the following research question:

- RQ1.3: What other factors should we consider when comparing methods?

Our research makes the following contributions:

- Analysis from our comparison of the efficiency and effectiveness of the four vulnerability detection methods.
- A detailed description of the methodology and related findings, which may be useful for future comparisons of vulnerability detection methods

We are releasing our vulnerability dataset once the vulnerabilities are safely mitigated at <https://github.com/RealsearchGroup/vulnerability-detection-20>. We are working with OpenMRS to minimize the risk of disclosing information about vulnerabilities that would endanger OpenMRS users, since medical systems are popular targets for malicious actors.

The rest of this chapter is structured as follows. In Section 3.1 we provide a brief overview of the previous work. We discuss other related work in Section 3.2. In Section 3.3 we describe the vulnerability detection methods used in this paper. In Section 3.4 we discuss the SUT used in our Case Study, OpenMRS. In Section 3.5 we discuss the sources of data for the Case Study. In Sections 3.6, 3.7, and 3.8 we outline our research methodology for RQ1.1, RQ1.2, and RQ1.3. We discuss the equipment used in Section 3.9. We report our results in Section 3.10. We discuss our findings in Section 3.12. We discuss limitations of our study in Section 3.11. We discuss the findings in Section 3.12 and conclude with Section 3.13.

3.1 Previous Work by Austin et al.

Our study is a theoretical replication⁵ of previous work done by Austin et al. [22, 23]. The goals of the previous work are “*to improve vulnerability detection by comparing the effectiveness of vulnerability discovery methods and to provide specific recommendations to improve vulnerability discovery with these methods*”[22]. In their first study [22] Austin et al. applied SMPT, EMPT, DAST, and SAST to two electronic medical records systems, Tolven Electronic Clinician Health Record (eCHR), a Java-based application with 466,538 lines of code, and OpenEMR, a PHP-based application with 277,702 lines of code. The second publication [23] added another SUT, PatientOS, a Java-based mobile application with 487,437 lines of code. In both studies, the authors used one tool for each automated method. The DAST tool used by Austin et al. was only applicable to web applications. Consequently, they only applied SMPT, EMPT, and SAST to PatientOS.

For each SUT, Austin et al. compare the number and types of vulnerabilities found by each method, as well as the rate of vulnerabilities per hour for each method. In the current study, we examine these same metrics, referring to the number and types of vulnerabilities as “effectiveness” and the vulnerabilities per hour as “efficiency”. We discuss the results of Austin et al in comparison with our study for effectiveness in Section 3.10.1.5, and for efficiency in Section 3.10.2.2.

3.2 Related Work

Many studies have focused on a single category of methods, such as comparisons of DAST tools or comparisons of SAST tools[17, 28]. We highlight three notable examples. In 2010, Doupé et al. [74] compared eleven “point and click” DAST tools. The authors found that while some types of vulnerabilities could be found reliably, other types of vulnerabilities could not. More recently, Klees et al. [151] examined 32 papers on fuzz testing and performed an experiment comparing two tools against five benchmark applications. Klees et al identified several best practices for comparing DAST tools, particularly fuzzers, such as avoiding arbitrary timeouts and carefully selecting the sample inputs to the tool. The U.S. National Institute of Standards and Technology (NIST) Software Assurance Metrics and Tool Evaluation (SAMATE) program has performed a series of Static Analysis Tool Expositions (SATE) [68, 204, 203, 202, 201]. On a regular basis, the SAMATE program establishes a set of trials. Participants in each trial, which include multiple organizations from industry, are required to run their tools against one or

⁵A theoretical replication seeks to investigate the scope of the underlying theory, for example by redesigning the study for a different target population, or by testing a variant of the original hypothesis of the work [169]

more benchmarks. The results are reviewed by SAMATE organizers. These studies, particularly the experiments run by Klees et al. and the SAMATE program, inform our methodology for SAST and DAST methods, but do not compare across methods and do not examine manual methods.

Another common comparison between vulnerability detection methods is between static methods that primarily analyze source code, and dynamic methods that run tests against an active software system. Scandariato et al. [248] conducted an experiment in which nine participants performed vulnerability detection tasks. The authors examine the user experience for SAST and DAST tools, and analyze the efficiency of using SAST and DAST. Scandariato et al. found that although participants found DAST tools more “fun” to use, the participants were more efficient with SAST tools and considered SAST tools a better starting point for new security teams. Similar to our study, Antunes and Viera [20] found that different tools within the same method found different vulnerabilities, and that SAST found more vulnerabilities than DAST. In contrast to Scandariato et al. and Antunes and Viera, we further subdivide dynamic analyses into SMPT, EMPT, and DAST, exploring each of these methods separately and noting differences between manual and automated methods.

Many surveys and comparisons focus on a single type of vulnerability. For example, Chaim et al. perform a survey of Buffer Overflow Detection methods. They note that existing vulnerability detection methods are impractical or produce too many false positives, but that emerging hybrid methods are “promising”. Liu et al. [165] survey automated, state-of-the-art methods for finding and exploiting Cross-Site-Scripting (XSS) vulnerabilities, categorizing them as “static”, “dynamic”, or “hybrid”. They note that the increasing size of web applications may hinder the effectiveness of these automated methods, but do not perform an empirical analysis. Fonseca et al. perform an empirical comparison of the effectiveness of different DAST tools [97] for finding XSS vulnerabilities. Practitioners may prioritize some types of vulnerabilities over others and these studies assist practitioners in understanding how vulnerability detection methods compare for a single type of vulnerability. However, applications are rarely threatened by a single type of vulnerability. Our study, which examines the effectiveness of methods across a range of vulnerability types; gains insight from and provides additional insight into results from studies which focus on a single type of vulnerability.

An additional area of related work is the development and application of benchmarks for security testing tools, such as the 2010 work by Antunes and Viera [21] on developing a benchmark for SAST and DAST tools. As noted in the SATE V report, benchmark studies have an important role in evaluating security testing methods [68]. However, the use of vulnerability detection methods in benchmark studies may differ from how security vulnerability detection methods would be applied in practice. For example, the three web infrastructure performance

benchmark systems used by Antunes and Viera[21] to develop security benchmarks contained a combined 2,654 lines of code which could be manually reviewed by security experts in a reasonable amount of time. The results of our study on OpenMRS, which has over 3,000,000 lines of code, may not generalize to smaller systems such as those examined by Antunes and Viera.

3.3 Vulnerability Detection Methods

In this section describe the four *vulnerability detection methods* from our case study. We will rely on the terminology for Program Analysis and Testing approaches described in Chapter 2 Section 2.3. We distinguish between the *analysis type* and the *vulnerability detection method* since confusion may arise due to common names for vulnerability detection methods which are derived from their analysis types. For example, Dynamic Application Security Testing (DAST), Exploratory Manual Penetration Testing (EMPT), and Systematic Manual Penetration Testing (SMPT) all use *dynamic* analysis, even though only DAST has “dynamic” in the name.

3.3.1 Manual Methods

Both *manual* methods examined in this study are *dynamic* methods that *do not require access to source code*. Manually examining the entire source code for system as large as OpenMRS is infeasible. A high-level overview of how manual dynamic testing methods, particularly systematic methods, are applied is shown in Figure 3.1. This figure is based on the process for dynamic methods presented in ISO/IEC/IEEE 29119-1 [135].

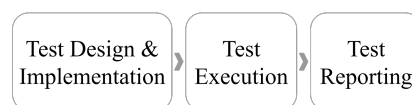


Figure 3.1: Applying Manual Test Methods (based on ISO/IEC/IEEE 29119-1)

3.3.1.1 Systematic Manual Penetration Testing (SMPT)

Specifically, SMPT is a form of scripted testing defined by ISO 29119-1 [135] as “dynamic testing in which the [analyst]’s actions are prescribed by written instructions in a test case”. SMPT involves *dynamic*, *manual*, and *systematic* analysis. SMPT *does not require access to source code*.

In SMPT, the analyst begins by writing a set of test cases and planning how the test suite will be run for a particular test execution in what ISO 29119-1 refers to as the *Test Design & Implementation* stage, as shown in Figure 3.1. The tests are then executed. In the final stage, the test results are documented and reported.

Figure 3.2 shows an example SMPT test case from our case study. As can be seen in Figure 3.2, the steps recorded in an SMPT test case are the actions a person would take when interacting with the system. SMPT test cases can be developed in a variety of ways [260]. As we will discuss further in the methodology under Section 3.6.1.2.1, for our case study we used the the OWASP Application Security Verification Standard (ASVS) as the basis for our test cases. ASVS provides a set of security controls which can be tailored to develop specific test cases for a software system. The test case in Figure 3.2, is based on ASVS Control 2.1.7 - *Verify that passwords submitted during account registration, login, and password change are checked against a set of breached passwords either locally (such as the top 1,000 or 10,000 most common passwords which match the system's password policy) or using an external API*. In the test case, the administrator attempts to create a user with the common password “Passw0rd”.

3.3.1.2 Exploratory Manual Penetration Testing (EMPT)

Exploratory Manual Penetration Testing is a *manual*, unscripted, *exploratory*, *dynamic* method that *does not require access to source code*. Previous studies of functional exploratory testing have suggested that knowledge and experience may play a significant role in exploratory testing [137, 226].

The process for EMPT is similar to the process shown in Figure 3.1. As found by Votipka et al. [280], security analysts who perform exploratory testing spend time learning about the system prior to beginning exploration. The analyst then moves on to activities such as “exploration” and “vulnerability recognition” [280]. The analyst also still documents and reports all vulnerabilities found. However, the process is less formal and more iterative for EMPT as compared with SMPT.

3.3.2 Automated Methods

We examine two categories of *automated* vulnerability detection methods, Dynamic Application Security Testing (DAST) and Static Application Security Testing (SAST). Figure 3.3 provides an overview of how automated, i.e. automatic tool-based, methods are applied. The tools must first be setup, which includes installing, configuring, and customizing the tool. The analyst then runs the tool. Once the automated portion of the analysis is complete, the analyst must review the tool output to remove false positives and prepare the report.

Test Case ID: XXX
ASVS Control: 2.1.7

Steps:

- 01) Open the OpenMRS web app to the login screen
- 02) Type ‘‘admin’’ as the username and ‘‘Admin123’’ as the password
- 03) Select ‘‘Inpatient Ward’’ as the location
- 04) Click ‘‘login’’
- 05) Select ‘‘System Administration’’, then select ‘‘Manage Accounts’’.
- 06) Click ‘‘Add New Account’’
- 07) Enter the following information:
 - Family Name: Potter
 - Given Name: Harry
 - Gender: Male
- 08) Select ‘‘Add User Account?’’
- 09) Enter the following information:
 - Username: Hedwig
 - Privilege Level: Full
 - Password: Passw0rd
 - Confirm Password: Passw0rd
- 10) Leave all other defaults as they are
- 11) Click ‘‘Save’’

Expected Results: The password, ‘‘Passw0rd’’, should be rejected as it is on the list of the 10,000 most commonly-used passwords.

Actual Results:

Figure 3.2: Example SMPT Test Case

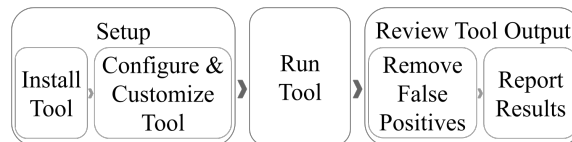


Figure 3.3: Applying Tool-Based Methods

3.3.2.1 Dynamic Application Security Testing (DAST)

DAST uses *automated* tools to perform *dynamic* analysis. We only include methods that *do not have access to source code* in the DAST category. DAST is sometimes referred to as Automated Penetration Testing [21, 22, 23], Black-Box Web Vulnerability Scanning [74], Fuzzing [151], or Dynamic Analysis [54]. In our case study, we examined two general-purpose DAST tools. One tool, the Open Web Application Security Project’s Zed Attack Proxy version 2.8.1 (OWASP ZAP,

further abbreviated as ZAP in tables)⁶, is a free, open-source, dynamic analysis tool which describes itself as “the world’s most widely used web app scanner”[216]. The second DAST tool, which we will refer to as DAST-2 (further abbreviated as DA-2 in tables), is a proprietary tool.

DAST tools automatically generate a set of malformed inputs to the SUT based on sample inputs provided by the analyst. For web applications, the sample inputs provided by the analyst and the malformed inputs generated by the DAST tool are represented as a sequence of HTTP messages such as the one shown in Figure 2.2. Background on HTTP is provided in Chapter 2, Section 2.2. Some DAST tools, such as OWASP ZAP, provide built-in ways to record HTTP messages. Other DAST tools require a standard file format such as HTTP Archive (.har)⁷ which can be generated by most web browsers.

Some DAST tools for web applications, including OWASP ZAP but not DAST-2, incorporate a web crawler [218, 74, 248], also referred to as a spider. Analysts can use the web crawler to automatically find additional resources that may not have been included in the original sample inputs. For example, if the analyst does not know that a particular resource exists or is accessible, the resource is unlikely to be included in the sample input the analyst provides.

Using the sample inputs and any additional information that may have been found using a web crawler, the DAST tool applies a set of security rules to create a new set of malformed inputs. If we consider the example HTTP message in Figure 2.1 as a potential message from a sample input, Figure 3.4 shows an HTTP message that could be created by DAST tools as part of a malformed input. In this example, the `sessionLocation` parameter is changed from a number, 0, to a script designed to find Cross-Site Scripting (XSS) vulnerabilities, `<script>alert(1);</script>`. The same XSS-focused rule could be applied to the `username` parameter instead of the `sessionLocation` parameter to generate a different malformed input. A different rule could ignore parameters entirely and search the `http` header for sensitive information or could re-order the HTTP messages in the sequence.

The rules used to generate different malformed inputs are typically associated with one or more CWEs. The CWEs covered by the rules in the tools we used are discussed in Appendix B.1. ZAP rules were associated with 33 CWEs while DAST-2 covered 44 CWEs for a combined 68 CWEs covered by DAST. ZAP covered 6 of the OWASP Top Ten while DAST-2 also covered 6 of the OWASP Top Ten. Five (5) of the Top Ten categories were covered by both tools for a combined 7 of the Top Ten covered between the two DAST tools.

With many combinations of rules and ways to apply them, DAST tools can create and run hundreds or thousands of malformed inputs. The malformed inputs generated by a DAST tool

⁶<https://www.zaproxy.org/>

⁷e.g. <https://docs.rapid7.com/insightappsec/scan-scope/>;
<https://www.netsparker.com/support/scanning-restful-api-web-service/>;
https://docs.gitlab.com/ee/user/application_security/api_fuzzing/create_har_files.html

```
POST http://127.0.0.1:8080/openmrs/login.htm HTTP/1.1
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:99.0) Firefox/99.0
Accept: text/html, application/xhtml+xml, application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Content-Type: application/x-www-form-urlencoded
Content-Length: 170
Origin: https://127.0.0.1:8800
Connection: keep-alive
Referer: https://127.0.0.1:8800/openmrs/login.htm
Cookie: JSESSIONID=owczzkvskvc8n4580psmvbb
Host: 127.0.0.1:8800

username=admin&password=Admin123&sessionLocation=<script>alert(1);</script>&redirectUrl=/openmrs/
referenceapplication/home.page
```

Figure 3.4: Message from a Malformed Input (Test Case) Produced by a DAST Tool

are sometimes referred to as *test cases*. DAST tools execute the “test cases” (malformed inputs) automatically, suggesting that DAST tools may be able to perform more testing in less time compared with manual methods such as SMPT [5]. One of the motivations of our study is to understand whether this promise of “more” inputs executed “faster” by automated methods produces equivalent or better results.

3.3.2.2 Static Application Security Testing (SAST)

We use the term SAST to refer to methods that use *automated* tools to perform *static, source code* analysis. SAST tools are a common way to comprehensively apply source code analysis, as manual source code analysis can be tedious and time-consuming [172, 143, 248, 262, 54]. In practice, SAST tools are less likely to be applied by security analysts [54, 118, 280] and more likely to be applied by the developers themselves [54]. In this study, we used three SAST tools from industry. First, we used the open-source community edition of Sonarqube version 8.2, which we refer to as *Sonarqube* (abbreviated as Sonar in tables). The two other tools examined, SAST-2 and SAST-3 (abbreviated as SA-2 and SA-3 in tables) are proprietary tools and cannot be named due to license restrictions. Sonarqube and SAST-2 were used to answer RQ1.1, while SAST-2 and SAST-3 were used to answer RQ1.2.

All SAST tools used in this study perform static analysis by first parsing the source code to build a tree representation of the code, known as a *syntax tree*. The tool then applies a set of rules to the syntax tree, where each rule describes a pattern within the syntax tree that may indicate a vulnerability [172]. Since the original work by Austin et al [22, 23], SAST tools have evolved to include additional features. For example, Sonarqube uses symbolic execution, as well as more traditional methods such as parsing the code using regular expressions, to identify vulnerabilities [171]. Similarly, both SAST tools used in this study employ taint analysis [35], although it is not clear whether taint analysis is available in the free / open-source version of

Sonarqube used in this study.

3.4 System Under Test - OpenMRS

The SUT for our case study was OpenMRS, an open-source medical records system. OpenMRS is a “Java-based web application capable of running on laptops in small clinics or large servers for nation-wide use”[205].

3.4.1 Why OpenMRS?

We selected OpenMRS as the SUT because OpenMRS is a “real” system that is actively used and actively under development. The 2018 U.S. National Institute of Standards and Technology (NIST) Static Analysis Tool Exposition (SATE) report [68], provides the following criterion for “real, existing software”: “their development should follow industry practices. Their size should align with similar software. Their programming language should be widely used for their purpose.” . OpenMRS follows common development practices for open-source systems, as discussed in their Developer Guide[205]. With over 3 million lines of code, OpenMRS is comparable to other modern medical records systems, such as the VistA system used by the US Department of Veteran Affairs[277] and Epic[85], which involve millions of lines of code. The languages and frameworks used by OpenMRS including Java, Javascript, Node.js, and SQL, consistently appear on lists of the most commonly used software technologies [269, 108, 37, 38]. Furthermore, as of July 2021[206], OpenMRS is actively used many contexts including clinics, hospitals, and health networks in Mexico, Haiti, Tanzania, Pakistan, and Bangladesh.

Additionally, we selected OpenMRS due to its domain. The three SUT examined by Austin et al [22, 23] were medical records systems. Hence the SUT for the current study should also come from the medical domain. Although OpenMRS was not examined by Austin et al., OpenMRS has also been used in other research on software testing and security analysis[275, 234]. The security of medical records systems is, if anything, a more important issue in 2021 than in 2011, with healthcare systems an increasingly popular target for hackers [237, 276, 48, 26].

3.4.2 Technical Description

OpenMRS contains 3,985,596 lines of code as measured by CLOC v1.74⁸ including 476,139 coding lines, i.e. not comments, of Java as well as 1,884,233 coding lines of Javascript. The OpenMRS architecture is modular. In this study, we examined the 43 modules that compose

⁸<https://github.com/AlDanial/cloc>

the basic reference application for OpenMRS Version 2.9. The source code for each module is available on github⁹. We compiled and ran OpenMRS using Maven and Jetty as described in the Developer's Manual[205].

3.4.3 Security Practices at OpenMRS

OpenMRS is open-source software. The OpenMRS team has received vulnerability reports from both volunteers and independent researchers in the past, based on SAST and other vulnerability detection methods. When we reached out to OpenMRS with our results, our understanding was that SAST and DAST tools were not being used at the organizational level. Since then, OpenMRS's security posture has continued to mature, including more vulnerability detection efforts.

3.5 Data Sources

The data for the case study came from two sources: 1) a team of five (5) researchers and 2) sixty-three (63) students from a graduate-level security course. In this section, we provide background on these two data sources.

3.5.1 Researcher Data

Three Ph.D. student researchers, one Master's student researcher, and one undergraduate student researcher applied SMPT, DAST, and SAST; and reviewed the alerts or other failures output by all four methods as part of data collection for RQ1.1. The researchers also reviewed all student information used in this study to remove incorrect answers. All graduate-level researchers had participated in a graduate-level software security course. The undergraduate student researcher had taken two security-related undergraduate courses.

3.5.2 Student Data

Sixty-three (63) of 70 students in a graduate level software security course gave signed consent for their data to be used for this study. Student data was collected following North Carolina State University (NCSU) Institutional Review Board Protocol 20569. Students worked in teams of 3-4 students, for a total of 19 teams. Where data could only be collected at the team level, we used data from the 13 teams in which all team members consented to the use of their data. Where data is available at the student level, we used data from all 63 students who consented.

⁹<https://github.com/openmrs>

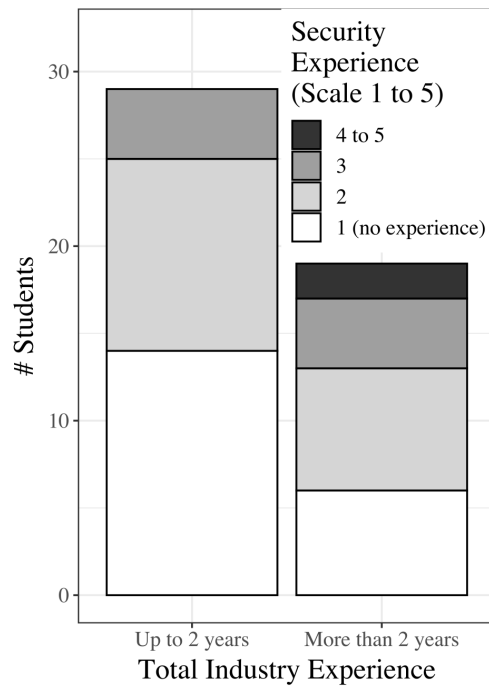


Figure 3.5: Industry Security Experience of Students

Student EMPT and SMPT data was used as part of data collection for RQ1.1. Researchers then analyzed students' reported efficiency scores to answer RQ1.2.

3.5.2.1 Students' Prior Experience

Students were asked to fill out a survey about their relevant experience including industry experience and related coursework. The full survey is available in Appendix B.2. First, the survey asked students about the amount of time they worked in industry. Second, the survey asked students to note how much of their time in industry involved cybersecurity on a scale from 1 (none) to 5 (fully). Seven (7) of the 55 students had no industry experience. The distribution of security experience for the 48 students with industry experience is shown in Figure 3.5. Fifty-five (55) of the 63 students whose data was used in this study provided valid survey responses. The x-axis of Figure 3.5 indicates students who had up to 2 years of industry experience as compared with more than 2 years of industry experience. The y-axis indicates the number of students. The shading within the bar chart indicates security experience. Darker shades indicate more security experience.

Students had a range of industry experience, but most students had little experience in cybersecurity at the start of the course. At the lower end were 7 students with no experience, while the maximum experience was approximately 10 years. The median and average industry experience of students, including students with no experience, was 1 year and 1 year 8 months, respectively. In addition to industry experience, 7 students had previously taken a course in security or privacy. Eight (8) students were currently taking a course in security or privacy in

addition to the course from which we collected data.

3.5.2.2 Course Assignments

The data used in this study that comes from student assignment responses is taken from the Course Project. The course project had four parts which were distributed over the semester. The verbatim text from the course project assignments is provided in Appendix B.3. A summary of the assignments relevant to our study is as follows:

- **SMPT Assignments:** In Project Part 1, students were required to write and execute a set of 15 systematic manual penetration test cases. Each test case mapped to at least one ASVS control. In Project Part 3, students were required to write and execute ten additional test cases for logging, and five additional test cases to increase the ASVS coverage of their test suite. Correct, unique test cases and their results were used as part of Data Collection for RQ1.1 (Effectiveness). The test cases were re-run and supplemented with additional test cases by researchers, as we will discuss in Section 3.6.1.2.1. Student performance and experience with SMPT as part of these assignments also informed their response to the Comparison Assignment listed below, which was used to collect data for RQ1.2 (Efficiency) and RQ1.3 (Other Factors).
- **EMPT Assignment:** In Project Part 4, students spent three hours individually performing exploratory penetration testing. EMPT was assigned at the end of the course when students were familiar with the SUT and with many security concepts. Students produced a video recording of their three-hour session, speaking out loud about any vulnerabilities found; and created black-box test cases to enable replication of each vulnerability found. The vulnerabilities found by students were used as part of the Data Collection for RQ1.1 (Effectiveness). Student performance and experience with EMPT informed their responses to the Comparison Assignment, which were used as part of Data Collection for RQ1.2 (Efficiency) and RQ1.3 (Other Factors).
- **DAST Assignment:** In Project Part 2 students used two DAST tools (OWASP ZAP and DAST-2), using 5 test cases from their SMPT assignments to provide the sample inputs for the DAST tool. Students reported the number of true positive vulnerabilities and the amount of time spent reviewing the output. Students' performance and experience with the DAST assignment contributed to their response to the Comparison Assignment, which was used as part of the Data Collection for RQ1.2 (Efficiency) and RQ1.3 (Other Factors).

- **SAST Assignment:** In Project Part 1, each student team ran two SAST tools (SAST-2 and SAST-3) on a subset of the SUT. Due to the length of SAST reports, students were only required to review at least 10 of the alerts from each tool to determine whether the alerts were true or false positives. Students had to identify at least 5 false positives even if it required reviewing more than 10 alerts to ensure that students were not incentivized to focus on trivial false positives. The students reported the number of true positive vulnerabilities found, as well as the amount of time spent reviewing the alerts. The SAST assignment contributed to the students' response to the Comparison Assignment, which in turn was used as part of the Data Collection for RQ1.2 (Efficiency) and RQ1.3 (Other Factors).
- **Comparison Assignment:** At the end of the course in Part 3 and Part 4 of the project, each team created a table showing the number of vulnerabilities found by each activity, the amount of time it took to discover these vulnerabilities, and the resulting VpH. The students reflected on their experience with the different vulnerability detection methods in a free-response format. The numeric responses in the table were used to answer RQ1.2. Two researchers applied qualitative analysis to the students' free-response answers for RQ1.3.

3.5.3 Overview of Data Sources Per Research Question (RQ)

Table 3.1 provides an overview of the data sources used in Data Collection for each research question. All data analysis was performed by researchers and is therefore not included in the table. In Table 3.1, student data is indicated with an S. Researcher data is indicated with an R. As can be seen in the table, RQ1.1 relied primarily on researcher efforts, although student data was used with SMPT and EMPT. Student data was used more extensively in RQ1.2, and was the source of the documents used in qualitative analysis for RQ1.3. The detailed methodology for each Research Question will be further explained in Sections 3.6, 3.7, and 3.8.

3.6 Methodology for RQ1.1 - Effectiveness

Our first research question is: *What is the effectiveness, in terms of number and type of vulnerabilities, for each method?* To answer RQ1.1, we need a comparable set of vulnerabilities found by each method. Ensuring the vulnerability counts were comparable required an extensive Data Collection process described in Section 3.6.1 which is split into two phases. In the first phase, *Applying the Method* we applied each vulnerability detection method described in Section 3.6.1.2 to our SUT. The initial outputs of each method, which we will refer to as the list

Table 3.1: Data Sources

R = Researchers; S = Students

		Method			
		SMPT	EMPT	DAST	SAST
RQ1.1	Applying Method	S ^a & R	S ^b	R	R
	Review Failures	R	R	R	R
RQ1.2	Recorded Efficiency	S	S	S ^c	S ^c
	Data Cleaning	R	R	R	R
RQ1.3	Document Source	S	S	S	S
	Qualitative Coding	R	R	R	R

^aStudent SMPT results for RQ1.1 were reviewed and replicated by researchers

^bStudent EMPT results for RQ1.1 were reviewed by researchers

^cTo enable fair comparison, we use the performance of Students for all methods in RQ1.2.

of *failures*, are not comparable. For example, an analyst performing EMPT might document a single vulnerability where a malicious input script such as `<script>alert(123)</script>` is saved in one part of the application due to an input validation vulnerability and executed by the application due to lack of output sanitization. A SAST tool, on the other hand, may scan for input validation and output sanitization using different rules, resulting in two alerts for the same issue documented using EMPT. To reduce possible biases introduced by different vulnerability counting approaches or different vulnerability type classification approaches, we review the failures from each method in the second phase of Data Collection described in Section 3.6.1.3. Once the data has been collected, we analyze the results as described in Section 3.6.2.

3.6.1 Data Collection

Figure 3.6 provides an overview of the Data Collection process. As seen previously in Table 3.1, we subdivide our Data Collection process for RQ1.1 into two phases - *Applying the Method* and *Reviewing the List of Failures* output by each method. In the first phase, we collect a list of true positive failures. The first phase varies widely by method. For the second phase, to enable empirical analysis, the list of failures is further reviewed to ensure the vulnerability count, type, and severity are comparable across the methods. We begin this section with a set of guidelines used across methods. We then go into the details of how data collection was performed for each phase for each method.

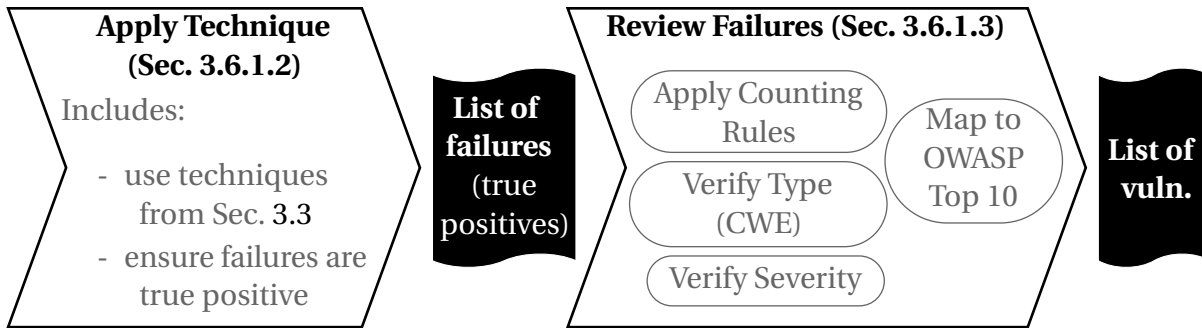


Figure 3.6: Data Collection for RQ1.1

3.6.1.1 General Guidelines

This section provides key guidelines for the Data Collection process which we will refer to for the remainder of Section 3.6.1. True / False Positive Classification guidelines are used when applying automated methods (DAST and SAST). As described in Section 3.6.1.3, the list of failures from each method was assessed to ensure that the vulnerability count, type, and severity are consistently evaluated using the guidelines provided in Sections 3.6.1.1.2, 3.6.1.1.3, and 3.6.1.1.4

3.6.1.1.1. True/False Positive Classification Guidelines

A true positive alert or vulnerability is one that meets the definition of a vulnerability from Chapter 2, Section 2.1. We follow a conservative policy towards true and false positive classification based on the principle of Defense-in-Depth [145]. We considered an alert or other finding to be a vulnerability if it could potentially lead to a security breach. For example, an alert is raised due to a particular malicious input. Upon review, we note that the input is stored in the database without encoding or other protection. We would classify the alert as a true positive even if we have not yet found another vulnerability where the malicious input is executed, e.g. by the application as part of an XSS attack. There may be vulnerabilities yet to be found, and changes to the code could make the input validation vulnerability more exploitable in the future. We also consider an alert to be a “true positive” even if the vulnerability found does not have the same CWE type as the original failure. CWE type is reviewed separately using the Vulnerability Type Guidelines in Section 3.6.1.1.3.

3.6.1.1.2. Counting Guidelines

The CVE program, which is the source for vulnerabilities in the NVD [196], also provides a set of guidelines[180] for CVE Numbering Authorities (CNAs) to help CNAs identify and remove

false positives, as well as consolidate duplicate vulnerability reports. We based our counting process for determining the number of vulnerabilities identified using each method on the CVE Counting Rules[180]¹⁰. The counting rules used in our analysis were:

- True/False Positive: The failure report must provide evidence of negative impact or that the security policy of the system is violated; and
- Independence: Each unique vulnerability must be independently fixable.

We applied these counting rules to the list of failures output by each method. For example, we applied the counting rules to the alerts produced by a tool. When one alert pointed to the same vulnerability as another alert, we marked one of the alerts to be a “duplicate” of the other.

Where we were unsure of the independent fixability of different failures, we assumed that the initial count was correct and the failures represented independent vulnerabilities. For example, a vulnerability detection tool could raise two alerts for the same type of vulnerability, where each alert was triggered by a different checkbox in the same form. We counted each alert as a distinct vulnerability unless we knew that the checkboxes relied on the same server-side code.

3.6.1.1.3. Vulnerability Type Guidelines

The vulnerability types assigned to each vulnerability are based on two systems - CWE and OWASP Top Ten - which are described in Chapter 2 Section 2.1. How each vulnerability was initially assigned a CWE varied by method. the initial CWE was assigned in SMPT based on the test case; in EMPT, by the student who found the vulnerability; and in DAST and SAST by the tool.

Researchers reviewed the CWE type assigned to each vulnerability and corrected the CWE assignment when the CWE was missing, inaccurate, or inconsistent with other vulnerabilities in our dataset. For example, a DAST tool creates a malformed input (test case) designed to trigger XSS as described in Section 3.3.2.1. When the malformed input is executed against the SUT, it triggers an error message revealing sensitive information (CWE-209¹¹). The error message is unexpected behavior which may result in an alert being flagged by the DAST tool. However, the alert will be assigned CWE-79 (XSS¹²) since that was the rule used to create the test case. In our study, we would consider this alert to be “true positive” since the alert points to a vulnerability. However, the CWE type would need to be reclassified - i.e. we would

¹⁰The CVE Counting rules have been updated since our original study. In future work, the authors may follow the updated rules: <https://cve.mitre.org/cve/cna/rules.html>

¹¹<https://cwe.mitre.org/data/definitions/209.html>

¹²<https://cwe.mitre.org/data/definitions/79.html>

consider the error message containing sensitive information to be a CWE-209 vulnerability even though the alert indicates CWE-79. When a classification is incorrect, if there are already similar vulnerabilities in our dataset we reclassify the alert to the same CWE as the similar vulnerabilities. In the previous example of an alert reclassified as CWE-209, there were many CWE-209 vulnerabilities flagged by SMPT and EMPT prior to running the DAST tool. Otherwise, the analyst may need to perform a keyword search of the CWE database [184] to find an appropriate CWE. The CWE mapping to the OWASP Top Ten [182] as well as more general guidelines such as the list of “Weaknesses for Simplified Mapping of Published Vulnerabilities”[183] and relationships between CWEs provided by the CWE system [181] were also used to identify the most appropriate CWE. When multiple CWEs were equally applicable, multiple CWEs could be assigned to the same vulnerability. Fifty-six (56) CWE types were found in our experiment.

We mapped the vulnerabilities found to the OWASP Top Ten through their assigned CWE values using the mapping provided by CWE[182]. The OWASP Top Ten provides a more readable summary of the types of vulnerabilities found, requiring 10 categories instead of 56. The OWASP Top Ten, as described in Chapter 2 Section 2.1, categorizes and ranks vulnerability types based on their severity and how frequently they are seen in software systems, providing additional insight into the vulnerabilities found.

3.6.1.1.4. Severity Guidelines

We examine two different perspectives for the severity of the vulnerabilities found. Our first perspective on severity is through the lens of the OWASP Top Ten. The OWASP Top Ten are ranked in a “risk-based order” suggesting that the first category of the OWASP Top Ten, A01 - Broken Access control, is considered highest risk and therefore more severe than vulnerabilities associated with lower-ranked categories.

We also examine severity based on severity classifications provided by tools, supplemented by analysis of high-frequency vulnerability types and discussions with OpenMRS. As discussed in Section 3.6.1.2, we excluded alerts that were labeled insignificant or inconsequential by the tools themselves. We then further split the vulnerabilities between those that are “less severe” and those that are “more severe”. Different tools have different labels for the different levels. We consider the lowest severity level for each tool to be “Low”. Vulnerabilities classified as “less severe” include all vulnerabilities where at least one tool indicated the vulnerability was of “Low” severity. Once vulnerabilities were detected, we reviewed “more severe” vulnerabilities where more than 20 vulnerabilities associated with the same CWE were found by the same tool or method. In our experience, if a tool or method flags large numbers of vulnerabilities associated with the same vulnerability type, it is unlikely that those vulnerabilities are more

severe. Additionally, large quantities of incorrectly classified vulnerabilities may skew the results. Finally, we adjusted severity level based on discussions with OpenMRS. Since the tool-based severity was adjusted depending on the results, we discuss the vulnerability types where severity was updated in Section 3.10.1.3.

3.6.1.2 Applying the Method

The process of applying of each method is slightly different, as described in Section 3.3. We discuss details of how each method was applied for our Case Study for SMPT in Section 3.6.1.2.1, for EMPT in Section 3.6.1.2.2, for DAST in Section 3.6.1.2.3, and for SAST in Section 3.6.1.2.4.

3.6.1.2.1. SMPT

To apply SMPT as part of RQ1.1, 131 test cases were manually written and executed by a combination of *students* (S) and *researchers* (R), as shown in Table 3.1. The test cases were based on the OWASP Application Security Verification Standard (ASVS) [278]. As described in Chapter 2 Section 2.1, ASVS has three levels of controls. However, “*Level 1 is the only level that is completely penetration testable using humans*” [278]. In addition to the ASVS Level 1 controls, students and researchers used knowledge of OpenMRS and documentation available on the OpenMRS wiki¹³ to develop test cases specific to OpenMRS. We excluded 44 controls that were not applicable to the application. For example, ASVS control 5.2.3 [278] states “*Verify that the application sanitizes user input before passing to mail systems to protect against SMTP or IMAP injection*” and is not applicable since the SUT did not include a mail server.

The 131 test cases in the test suite covered 63 of the remaining 87 controls. We used 86 test cases developed by *students*, as well as 45 test cases developed by *researchers* to increase ASVS coverage. Students originally wrote over 390 test cases as part of their course assignments described in Section 3.5.2.2. However, many of the students’ test cases were duplicates of each other since the 13 teams worked independently and generally wrote test cases for easier security concepts. Additionally, test cases were removed due to quality concerns with the test case or the results recorded.

Each test case was executed by two independent analysts to reduce bias and inaccuracy due to subjectivity and human error. For the 86 test cases developed by students, the first test case execution was performed by the students and the second execution was performed by researchers. For the 45 test cases developed by researchers, two different researchers each executed the test case. When the two executions of the test case produced different results, an

¹³<https://wiki.openmrs.org/>

additional researcher executed the test case and the result (pass or fail) given by two of the three test case executions was recorded as the actual result.

3.6.1.2.2. *EMPT*

As shown in Table 3.1, for RQ1.1, EMPT was applied by *students* (S), according to the assignment outlined in Section 3.5.2.2. Data from 62 students was used as part of data collection for EMPT in RQ1.1, since 1 of the 63 students in the study did not complete the EMPT assignment. Students were required to spend three hours performing EMPT and record the results via video. Students documented their results as test cases to enable verification of the results. Extensive review of the student results was needed, which we will discuss in Section 3.6.1.3.2. We therefore distinguish between Student Reported Vulnerabilities (SRVs) from the first phase of data collection, and the final set of vulnerabilities from EMPT used to answer RQ1.1.

An important factor in exploratory testing is ensuring that individuals have sufficient knowledge and experience [137, 136, 280] because EMPT is based on knowledge and experience. The students had limited security experience at the start of the course, as discussed in Section 3.5.2. However, our results suggest that the students had sufficient experience by the time of the EMPT assignment to be effective.

3.6.1.2.3. *DAST*

As shown in Table 3.1, researchers (R) applied DAST for RQ1.1. As shown in Figure 3.3, individuals applying DAST tools must first setup and run the tools, then review the output for false positives. We include the review of the tool output to remove false positives in the first phase since it is an integral part of applying automated methods.

Setting Up and Running the DAST Tools: As discussed in Section 3.3.2.1, DAST tools require a set of *sample inputs* to the application, to which the DAST tool applies a set of rules to create a set of *malformed inputs*. The DAST tool runs the malformed inputs against the SUT and determines whether a security alert should be raised based on how the SUT responds. To better understand whether DAST tools would find the same vulnerabilities as SMPT and other methods, the researchers based the sample inputs on 6 test cases from the SMPT suite. The test cases were selected to maximize coverage of the SUT. Due to the complexity and resources required by the DAST-2 tool, we were unable to run additional test cases, a limitation discussed further in Section 3.11. Based on the sample inputs based on the SMPT test cases, each tool generated and tested a set of malformed inputs against the system using the default set of rules. The CWEs covered by each ruleset are shown in Table B.1 of Appendix B.1.

Two researchers performed multiple trials of each tool in different configurations to deter-

mine how different choices would impact the alerts produced. For example, DAST-2 by default only used a randomized subset of the test cases. We trialed DAST-2 with both the full set of test cases and a randomized subset. Due to the repetitive nature of the test cases generated by DAST-2, the alerts produced by the random subset were similar to the alerts produced by the full set, and the alerts produced by the full set did not seem to point to any additional vulnerabilities. Once we were satisfied with our configuration, we performed a final run. We used the list of failures produced by the final run. Tool-specific details for the final configuration and run of each DAST tool are as follows:

OWASP ZAP Final Configuration and Run: For OWASP ZAP, we used the proxy included in the tool to record our interactions. We then ran the spider before running the security scan. Since OWASP ZAP was run after DAST-2, we limited the OWASP ZAP input to the 6 SMPT test cases used for DAST-2. With OWASP ZAP we were able to cover all 6 test cases in a single run of the tool, which took less than 2 hours to execute.

DAST-2 Final Configuration and Run: DAST-2 was more resource-intensive and required more configuration by design. For DAST-2, we recorded the interactions for each of the 6 SMPT test cases in an HTTP Archive (.har) file and uploaded it to the tool as described in Section 3.3.2.1. As shown previously in Figure 2.2 and discussed in Chapter 2, for every explicit interaction with the application there could be 10 or more messages sent between the browser and the application server. In longer test cases such as the 11-step example in Figure 3.2, hundreds of http messages could be exchanged. Unfortunately, on our equipment the DAST-2 tool would crash due to memory constraints if more than approximately 6 HTTP messages were included in the initial sample. Hence each test case had to be recorded and run as a separate *model* within the DAST tool. For each model we removed all HTTP messages from the input sequence other than the messages that were key to the test case. For example, to apply DAST-2 based on the test case in Figure 3.2, we would include the HTTP messages for the GET request that loaded the login page in step 01, the POST request which performed the login at step 04 using the login details from steps 02-03, the GET request for the “Add New Account” page in Step 06, and the POST request for saving the user at step 11 containing the account information from steps 07-10. We would remove all other non-essential HTTP messages from the sequence, such the requests for the “System Administration” and “Manage Accounts” pages in step 05, and GET requests for Cascading Style Sheets (CSS). For the final run of DAST-2, 7 separate models were setup to cover the 6 test cases selected from SMPT. Based on our trial runs we used a randomized subset of malformed inputs to further reduce load. The final run required between 2 hours and 3 days for each of the 7 models.

Reviewing DAST Tool Output for False Positives: We reviewed the alerts output by the tools for true and false positives using the guidelines in Section 3.6.1.1.1. Unless otherwise noted, when we refer to the alerts from a DAST tool we exclude alerts marked as insignificant or inconsequential by the tools themselves. For example, with OWASP ZAP we exclude alerts where the severity level was “Informational” [217]. With OWASP ZAP, two reviewers independently examined all alerts. DAST-2 produced over one thousand alerts, and two researchers reviewing all alerts would be inefficient. For each model for DAST-2, if the model produced less than 40 alerts, two researchers each reviewed every alert. For models that produced more than 40 alerts, both reviewed at least 40 alerts to compare their agreement and determine whether continued review by two independent researchers was necessary for consistency. If their classification was consistent, the remaining alerts were divided between the researchers for review.

For both tools, the researchers calculated their inter-rater reliability using Cohen’s Kappa using R to determine if they were consistently classifying the results as true or false positive. For DAST-2 if the inter-rater reliability was at least 0.70 [168, 280] for the initial subset of alerts, the reviewers split the remaining alerts such that each reviewer only examined half of the remaining alerts. The inter-rater reliability for the classification of DAST alerts as true or false positive and the precision of the tools based on the final set of True/False Positives is reported in Section 3.10.1.1.

3.6.1.2.4. SAST

As seen in Table 3.1, applying SAST began with researchers (R) running the SAST tools on the SUT using the default security rules. As shown in Figure 3.3 and described in Section 3.3.2.2, SAST tools were first setup and run by the analyst. The tool output was then reviewed to remove false positives, producing the list of true positive failures needed for the next phase of data collection.

Setting Up and Running the SAST Tools: For each SAST tool, one researcher initiated automated scans for each of the 43 modules in the OpenMRS Reference Application, using the default security ruleset. The CWEs covered by each tool’s ruleset are shown in Table B.1 in Appendix B.1. No other configuration was needed.

Reviewing SAST Tool Output for False Positives: None of the SAST tools produced over 1000 results; therefore all alerts were reviewed by two researchers to identify and remove false positives using the guidelines in Section 3.6.1.1.1. We computed Cohen’s Kappa[46] to determine whether the true / false positive classification process was consistent and reproducible. A third researcher resolved disagreements. Similar to DAST results, unless otherwise noted, when we refer to the alerts from a SAST tool we exclude alerts marked as insignificant or inconsequential

by the tools themselves. For example, we exclude Sonarqube’s “Security Hotspots” which were for “security protections that have no direct impact on the overall application’s security”[265].

3.6.1.3 Reviewing the List of Failures

Each method produces different outputs, which we refer to as “failures”. For example, systematic, dynamic methods such as SMPT and DAST produce a set of failing test cases. In contrast, SAST finds specific weaknesses in the codebase that should be changed to improve the security of the system. Multiple failing test cases may be due to the same weakness in the codebase, or a single failing test case may be due to multiple weaknesses in the codebase. Consequently, the raw count of failures may be higher or lower for one method even though it is no more effective than another method. To resolve these potential counting differences, we take the list of failures from each method and apply the Counting Rules described in Section 3.6.1.1.2 to determine the number of vulnerabilities found by each method. While most of the failures are already assigned a CWE type by the tool or by the ASVS control, the CWE type may not be correctly assigned. We also reviewed the CWE assignments as part of reviewing the method output. As shown in Table 3.1, researchers (R) reviewed the list of failures for all methods.

3.6.1.3.1. SMPT

Two researchers (R) independently reviewed all failing test cases from SMPT to determine how many vulnerabilities were found using the counting rules outlined in Section 3.6.1.1.2. The researchers discussed their differences with a third researcher, as needed, to determine the final vulnerability count. The researchers also reviewed the CWE assigned to the vulnerabilities, as described in Section 3.6.1.1.3. Each test case was linked to an ASVS control, which was associated with a CWE. However, a test case failure may have been due to a violation of a different security principle than the original CWE associated with the test case and require correction. Finally, after discussing the final set of vulnerabilities with OpenMRS, we separated “less severe” vulnerabilities from more critical vulnerabilities as discussed in Section 3.6.1.1.4.

3.6.1.3.2. EMPT

For EMPT, one researcher (R) reviewed each Student Reported Vulnerability (SRV) while a second researcher audited 100 randomly sampled SRV as well as 2 additional SRV at the request of the first reviewer. A third researcher performed additional auditing. The first reviewer examined each of the 484 SRV to determine if the SRV was reproducible. The researcher removed SRV if the researcher could not understand the students’ documentation, if the researcher was unable to observe the result reported, or if the report was clearly a duplicate

of another report. The researcher determined if the SRV was correct using the True/False Positive guidelines described in Section 3.6. Researchers used the counting rules specified in Section 3.6.1.1.2 to remove duplicate SRV that had already been reported by other students and to split SRV into multiple vulnerabilities when students had incorrectly applied the counting rules. The researchers reviewed the CWE values assigned to each vulnerability, as described in Section 3.6.1.1.3, removing inaccuracies due to typos and other errors. Finally, after discussing the final set of vulnerabilities with OpenMRS, we distinguished less severe vulnerabilities from more severe vulnerabilities following the guidelines in Section 3.6.1.1.4.

After reporting our results to OpenMRS, feedback from the OpenMRS team resulted in the removal of five additional EMPT vulnerabilities that were determined to be not reproducible or not applicable. A team of Master's and Undergraduate students at NCSU working with OpenMRS to assist in fixing the vulnerabilities also provided feedback, which resulted in consolidating three EMPT vulnerabilities which had been found on three different pages of the application but were due to an error in shared search functionality.

3.6.1.3.3. DAST

, Once the true and false positive alerts were determined for each DAST tool two researchers (R) determined how many unique vulnerabilities were indicated by the alerts using the counting rules from Section 3.6.1.1.2. Researchers marked alerts that were triggered by the same vulnerability as “duplicates” of each other. If the researchers could not determine whether alerts were duplicates based on experience and analysis, the alerts were assumed to be unique unless the alerts shared the same CWE type, URL, and targeted parameter; in which case the alerts were assumed to be duplicate. It is unlikely, for example, that the “sessionLocation” parameter for the “/openmrs/login.htm” URL shown in Figure 3.4 would contain two distinct XSS vulnerabilities. Discussions of duplication and de-duplication continued in subsequent steps of the review if new information was uncovered by the analysis.

The CWE value of each vulnerability found by DAST was based on the CWE value assigned by the DAST tool to the alerts associated with the vulnerability. Researchers reviewed the CWE values using the guidelines in Section 3.6.1.1.3. The severity measures provided by the DAST tools were then used to distinguish less severe vulnerabilities from more severe vulnerabilities as described in Section 3.6.1.1.4.

3.6.1.3.4. SAST

Researchers (R) determined the number of distinct vulnerabilities indicated by the SAST alerts using the counting rules from Section 3.6.1.1.2. The researchers reviewed the vulnerability

CWE assignments provided by the SAST tools to ensure their accuracy following the guidelines from Section 3.6.1.1.3. Additionally, severity measures provided by the SAST tools were used to distinguish less severe vulnerabilities from more severe vulnerabilities as described in Section 3.6.1.1.4.

3.6.2 Data Analysis

Once we had a comparable set of vulnerabilities, we calculated the number of vulnerabilities found by each method for each type of vulnerability, using the CWE numbers and associated OWASP Top Ten categories. Vulnerability count is commonly used in both academia and industry as a measure of security risk [185]. We used vulnerability counts and types to answer RQ1.1.

3.7 Methodology for RQ1.2 - Efficiency

For RQ1.2, we address the question *How does the reported efficiency in terms of vulnerabilities per hour differ across method?*. To reduce the bias that could be introduced by a high-performing or low-performing participant, we cannot rely on results from a single individual or team [148]. Using data from a graduate level security course worked well for three reasons. First, we have a wide participant pool. Second, the students are all required to perform exactly the same tasks, reducing external factors that could influence our results. Third, graduate students can be assumed to have some existing knowledge in computer science.

3.7.1 Data Collection

We collected efficiency information recorded by students (S), which we discuss in Section 3.7.1. Researchers (R) then performed data cleaning, as we discuss in Section 3.7.1.2 before the data could be analyzed. An overview of the data collection process for RQ1.2 is shown in in Figure 3.7

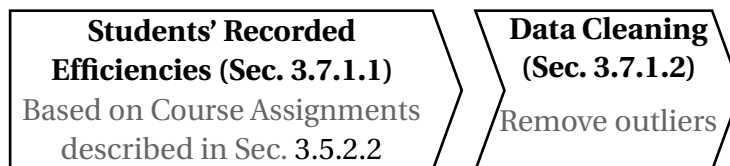


Figure 3.7: Data Collection for RQ1.2

3.7.1.1 Recorded Efficiency

To quantify efficiency, we started with information provided by the students (S) as shown in Table 3.1. As discussed in Section 3.5.2, the students worked in Teams of 3-4 to apply SMPT, EMPT, DAST, and SAST to OpenMRS as part of their course project. The students were given the assignments described in Section 3.5.2.2 which appear verbatim in Appendix B.3. We do not have efficiency information at the student level for SMPT, DAST, and SAST since these assignments were only reported at the team level. However, students were allowed to work independently for EMPT. Hence we use the average VpH across all participating members of each team for EMPT. For RQ1.2 we exclude data from students whose team members did not participate in the study as discussed in Section 3.5.2.

3.7.1.2 Data Cleaning

Once we have collected efficiency data, as shown in Table 3.1 the researchers (R) performed data cleaning as needed. We formally identified outliers for each method using the median absolute deviation and median (MADN)[288, 149], applying MADN to the VpH scores for each method. We removed outliers where the MADN was higher than 2.24, the threshold recommended in the literature[288, 150]. This data cleaning was needed to systematically identify and remove cases where students did not correctly follow the assignment to the extent that it impacted our analysis. For example, one team reported spending only 32 minutes on the SAST assignment described in Section 3.5.2.2 in contrast with the second-fastest team who spent 7.5 hours on the assignment. We detected and removed only four outliers, one in each method.

3.7.2 Data Analysis

With outliers removed, we retained 12 efficiency scores for each method. We performed a statistical comparison to determine if the average efficiency across the groups for each method was higher or lower than the average efficiency for other methods. We first applied the Shapiro-Wilk test [242] to the data for each method to assess normality. Based on the output of the Shapiro-Wilk test, we used Bartlett's test for homogeneity of variance [27]. Our case study data was normal, but the variance differed across methods. Based on the results of the normality and homogeneity of variance tests, we chose to apply the Games-Howell test [104, 148] to perform pairwise comparison across the different vulnerability detection methods and determine which methods were different. The Games-Howell test adjusts the p-value for multiple comparisons.

3.8 Methodology for RQ1.3 - Other Factors

Once the students had experience with the four vulnerability detection methods, the students (S) were asked to reflect on their experiences with each method and to compare the methods as part of the “comparison assignment” described in Section 3.5.2.2. Students were instructed to discuss tradeoffs between the methods, and “Based upon your experience with these methods, compare their ability to efficiently and effectively detect a wide range of types of exploitable vulnerabilities” as shown in Appendix B.3. The student responses to the comparison assignment were the source documents for RQ1.3 as shown in Table 3.1. The comparison assignment was answered at the team-level, and so the data used for RQ1.3 excludes students whose teammates did not agree to participate in the study.

Two researchers (R) performed qualitative analysis on the student responses to understand what other factors may distinguish the different methods. One researcher segmented the text by sentence, but left the sentences in order, to retain key contextual information. Both researchers independently coded each segment using “open coding”[50]. The researchers found that more than one code could apply to the same sentence. The researchers then compared and discussed their results. One researcher further standardized the codes and determined which codes were mentioned by more than one response. The resulting information was used to understand the results of RQ1.1 and RQ1.2, and may be informative for future work.

3.9 Equipment

We faced several equipment constraints. OpenMRS could be run with relatively low resources such as CPU, memory, and disk space. However, the tools used for SAST and DAST were more resource intensive. Additionally, for the course from which we collected student data, all 70 students needed independent access to the SUT as part of their coursework. Student access further needed to be setup such that students could not accidentally interfere with each others’ systems as they attempted to hack into the SUT. We used the Virtual Computing Lab¹⁴ (VCL) at our university, North Carolina State University (NCSU)¹⁵. VCL provided virtual machine (VM) instances. Researchers created a system image including the SUT (OpenMRS) as well as SAST and DAST tools. An instance of the image could be checked out by students or researchers and accessed remotely. Any data collected from students, e.g. all data for RQ1.2 and RQ1.3, leveraged the VCL images. Additional resources were needed when answering RQ1.1 to improve system coverage, including larger VCL instances, Virtualbox VMs based on the VCL images, and

¹⁴<https://vcl.apache.org>

¹⁵<https://vcl.ncsu.edu>

a large desktop machine with 24 CPUs, 32G RAM, and 500G disk space. Additional information on the systems is in Appendix B.4.

3.10 Results

In this section, we describe our results. Table 3.2 provides a high-level summary of the numeric results for RQ1.1 - *What is the effectiveness, in terms of number and type of vulnerabilities, for each method?* and RQ1.2 - *How does the reported efficiency in terms of vulnerabilities per hour differ across method?*. Detailed results for RQ1.1 and RQ1.2 are provided in Sections 3.10.1 and 3.10.2 respectively. We provide our qualitative results for RQ1.3 - *What other factors should we consider when comparing methods?* in Section 3.10.3.

Table 3.2: Results Summary

		SMPT	EMPT	DAST	SAST
Effectiveness:	# vulnerabilities: <i>more severe</i> (total) ^d	32 (37)	165 (185)	17 (23)	142 (823)
	# OWASP Top Ten Covered ^e	9	7	7	7
Efficiency:	Average VpH	0.69 ^f	2.22	0.55 ^f	1.17

^dThe total vulnerability count includes both “more severe” and “less severe” vulnerabilities as described in Section 3.6.1.1.4

^eOne category within the OWASP Top Ten is outside the scope of this study. Maximum possible coverage is 9.

^fThe difference in efficiency between SMPT and DAST is not statistically significant.

3.10.1 RQ1.1 - Method Effectiveness

In this section, we discuss the results for our question *What is the effectiveness, in terms of number and type of vulnerabilities, for each method?*. First, we go over information specific to automated, i.e. tool-based, methods: the agreement of researchers reviewing the output of vulnerability detection tools for true and false positives, and the number of false positives for each tool. We then provide the number of vulnerabilities discovered by each method, and the types of vulnerabilities identified by each method based on the CWE and OWASP Top Ten. We also include the severity of the vulnerabilities found.

3.10.1.1 True and False Positive Tool Alerts

We examined two tool-based methods in this study, SAST and DAST, for which we could calculate the precision of the tools. As noted in Section 3.6.1.2, for tool-based methods, two researchers classified the alerts produced by the tool as true or false positive. We calculated their inter-rater reliability and present the results in Section 3.10.1.1.1. The reviewers discussed the results with a third reviewer, who assisted in resolving disagreements, to create a final set of true and false positive counts which could be used to determine the tool precision as presented in Section 3.10.1.1.2 and shown in Table 3.3.

3.10.1.1.1. Reviewer Agreement

We calculated the inter-rater reliability of the reviewers for SAST and DAST using Cohen's Kappa [46]. Cohen's Kappa measures the extent to which reviewers agree beyond whatever agreement would be expected due to chance.

In a classification of two ratings such as true and false positive, if one of the ratings applies to an extremely high percentage of cases (e.g. 98%) and the other rating applies to an extremely small percentage of cases (e.g. 2%), the probability of agreement due to chance is estimated to be very high. The high estimated probability of agreement can lead to a paradox where reviewers who have high observed agreement, in other words - they apply the same rating to most of the objects being rated, but have low inter-rater reliability [90, 42, 92]. We observe this paradox of high observed agreement but low inter-rater reliability for both SAST tools (Sonarqube and SAST-2). Of the 698 alerts for Sonarqube, we calculated Cohen's Kappa on 693 alerts that were independently reviewed. One of the two reviewers found 12 of the 693 reports to be false positives, while the other reviewer did not consider any of the reports to be false positives. A third researcher reviewed the results and resolved disagreements for a final set of 4 false positives and 694 true positives out of the original 698 alerts. Based on the true/false positive classifications of the first two reviewers, the expected agreement, as estimated when calculating Cohen's Kappa, is 98.3% which is identical to the observed agreement of 98.3% with a resulting Cohen's Kappa of 0 (95% confidence interval ± 0). Similarly, Cohen's Kappa for SAST-2 is 0.22 (95% confidence interval ± 0.40), in spite of a high observed agreement of 93.1%. For SAST-2, the two reviewers met to discuss and resolve disagreements, while the third researcher participated in the discussion with a final false positive count of 16 out of 264 total alerts. These Kappa scores are low. However, given that our final false positive count for Sonarqube was only 4 of 698 total alerts and our final false positive count for SAST-2 was 16 of 264 alerts, even with dozens of reviewers, we may not be able to increase the inter-rater reliability statistics to the point where the observed agreement is statistically higher than 98.3%.

Another way to consider these results is that the reviewers agreed with the tool as frequently as they agreed with each other. While the reviewers had low inter-rater agreement as analyzed using the Cohen's Kappa statistic, they had high agreement in terms of the percentage of alerts on which the two reviewers agreed upon the classification, with 98.3% inter-rater agreement for Sonarqube and 93.1% for SAST-2. In both cases, the reviewer's observed agreement with the tool was as high with their agreement with each other, with observed agreement for each of the Sonarqube reviewers and the tool at 98.3% and 100%. Observed agreement between the SAST-2 reviewers and the tool was 94.3% and 96.6% for each of the two reviewers, respectively.

The inter-rater reliability for DAST tools was much higher. The inter-rater reliability for OWASP ZAP alerts was 0.97 (95% confidence interval ± 0.28). The inter-rater reliability for the 288 DAST-2 alerts reviewed by two individuals was 0.78 (95% confidence interval ± 0.16), which was above the recommended minimum cutoff of 0.70 [168, 280]. Therefore the remaining alerts were divided between the researchers to review as described in Section 3.6.1.2. The list of failures from OWASP ZAP was reviewed after the list of failures from DAST-2, and the researchers may have been more familiar with the process which may explain the higher reliability score for OWASP ZAP.

3.10.1.1.2. True / False Positives and Precision

Table 3.3 shows the Total Alerts (Tot. Alrt.), False Positives (FP), and Precision (Prec.) for automated, i.e. tool-based, methods. Table 3.3 includes information from the current study with OpenMRS (M) which we will discuss in this section, as well as results from Austin et al. [22, 23] which we will compare with our results in Section 3.10.1.1.3.

Subtable 3.3a provides the Tot. Alrt., FP, and Prec. for DAST, while Subtable 3.3b provides the Tot. Alrt., FP, and Prec. for SAST. In the columns for the current study, the Total (M) column for Tot. Alrt. and FP is the sum of the alerts and false positives, respectively, from both tools in each category. The Prec. row of the Total (M) column is the precision calculated based on the Tot. Alrt. and FP in the previous rows. The precision of Sonarqube and SAST-2 was 0.99 and 0.94, respectively, and the precision across the combined alerts for Sonarqube and SAST-2 at 0.98. The precision of OWASP ZAP was also high at 0.95. However, the precision of DAST-2 was 0.09, resulting in 0.23 precision across all DAST alerts.

We examined possible reasons for the low precision of DAST-2. Table 3.4 shows the DAST alert counts for each CWE type originally assigned by the tool based on the test case or check that triggered the alert. In Table 3.4 we use the abbreviation Tot. Alrt. for total alerts, TP Alrt. for true positive alerts, FP Alrt. for false positive alerts, and # Vuln. for number of vulnerabilities.

When an alert correctly provided an indicator of a vulnerability, but CWE provided by the

Table 3.3: Total Alerts (Tot. Alrt.), False Positives (FP), and Precision (Prec) for the Current Study Compared with Austin et al.

M indicates OpenMRS, E indicates OpenEMR, T indicates Tolven, and P indicates PatientOS

(a) DAST						(b) SAST						
	Current Study			Austin et al.			Current Study			Austin et al.		
	Total (M)	ZAP (M)	DA2 (M)	Total (E)	Total (T)		Total (M)	Sonar (M)	SA2 (M)	Total (E)	Total (T)	Total (P)
Tot. Alrt.	3412	550	2862	735	37	Tot. Alrt.	962	698	264	5036	2315	1789
FP	2625	28	2597	25	15	FP	20	4	16	3715	2265	1644
Prec.	0.23	0.95	0.09	0.97	0.59	Prec.	0.98	0.99	0.94	0.26	0.02	0.08

tool did not match the type of vulnerability found, we classified the alert as True Positive and reassigned the CWE type as described in Section 3.6.1.1.3. The Tool-Assigned CWE, shown in the first column of Table 3.4 is the CWE value provided by the tool. The final CWE types of the vulnerabilities found, reviewed and reassigned if necessary, are listed in the “Final CWE” column. For example, DAST-2 produced 2 TP alerts originally assigned to CWE-89 *SQL Injection*¹⁶. The alerts revealed an http message where sensitive patient information was visible in the URL. However, the researchers could not perform SQL injection based on the information in the alerts. Consequently, the vulnerability was reassigned CWE-598 *Use of GET Request Method With Sensitive Query Strings*.

More than one alert can point to the same vulnerability as discussed in Section 3.6.1.3. Using the same example of the 2 true positive alerts (TP Alrt.) found by DAST-2 with Tool-Assigned CWE-89; one of the alerts was a “duplicate” of the other, pointing to the same vulnerability, resulting in a vulnerability count of 1 in the # Vuln column. An alert could also be a duplicate of another alert with a different Tool-Assigned CWE. For example, for DAST-2 both of the of the true positive Tool-Assigned CWE-352 alerts were duplicates of vulnerabilities with different Tool-Assigned CWEs, therefore # Vuln column in Table 3.4 is 0. For rows where there were no true positive alerts, we leave the # Vuln. column blank, consistent with other tables.

The Tool-Assigned CWEs were based on the rules used for each alert as discussed in Section 3.3.2.1, and false positives tended to be associated with specific rules. As we can see in Table 3.4, for OWASP ZAP, the alerts that had been assigned a given CWE were either all true positive or all false positive, except for CWE-79. Of the Tool-Assigned CWEs for OWASP ZAP, only CWE-16 and CWE-79 were associated with more than one rule; and within CWE-79, the 4 true positive alerts were all associated with one rule, while the 3 False Positive alerts were

¹⁶<https://cwe.mitre.org/data/definitions/89.html>

associated with another rule.

While DAST-2 has more variance, some rules seem to produce more false positive alerts than others. For DAST-2 we can see that the most frequently occurring Tool-Assigned CWE type, and therefore rules, is CWE-35 *Path Traversal* which accounted for 2537 of all DAST-2 alerts. 2484 of the Path Traversal alerts are false positives, and the remaining 53 alerts were all reclassified as other CWE types. If we exclude alerts for CWE-35, the precision of DAST-2 goes from 0.09 to 0.69. The improved precision without Tool-Assigned CWE-35 alerts suggests that further customization such as updating or removing rules that do not accurately model the SUT may be able to improve the performance of DAST-2.

3.10.1.1.3. Tool False Positives and Precision Comparison with Austin et al.

Table 3.3 also shows the tool precision reported by Austin et al. [22, 23] for comparison with the current study. As described in Section 3.1, the SUT examined by Austin et al. were OpenEMR (E), Tolven eCHR (T), and PatientOS (P). Austin et al. used a single tool for each method. PatientOS is not included in the DAST results, since the DAST tool used by Austin et al. was not applicable to PatientOS.

As seen in Table 3.3, the SAST tool used by Austin et al. had much lower precision than the tools examined in the current study. The highest precision in the previous study for SAST was 0.26, as compared with the lowest precision of 0.94 in the current study. The high precision we observed may be part of greater trends in SAST tools as seen in the recent NIST SAMATE project's regular Static Analysis Tool Expositions (SATE) [68] where the precision and recall of SAST tools for Java was far higher than the precision and recall reported in previous work [22, 23].

Austin et al. had similar results to our current study using DAST. When applied to OpenEMR, Austin et al.'s DAST tool had a precision of 0.97. When applied to Tolven eCHR, the DAST tool only had a precision of 0.59. Austin et al. [22, 23] also found that entire categories of alerts could be labeled true or false positive, similar to the results shown in Table 3.4. The impact of not customizing tool rules on performance measures such as precision may be more apparent as tools become more advanced and precise.

In the current section (3.10.1.1.3) we have compared our work to Austin et al. on tool-based measures. Comparison with Austin et al. on effectiveness measures applicable to all four methods may be found in Section 3.10.1.5. Comparison with Austin et al. on efficiency measures may be found in Section 3.10.2.2.

Table 3.4: DAST Alerts

Tool-Assigned CWE	OWASP ZAP					DAST-2				
	Tot. Alrt.	TP Alrt.	FP Alrt.	Final CWE	# Vuln.	Tot. Alrt.	TP Alrt.	FP Alrt.	Final CWE	# Vuln.
Total	550	522	28	N/A	12	2862	265	2597	N/A	13
16 - Configuration	262	262		16	3					
35 - Path Traversal						2537	53	2484	79, 209, 613	1
77 - Command Injection						2	1	1	20	1
79 - Cross-site Scripting	7	4	3	79	3	307	207	100	79, 20, 209, 598	10
89 - SQL Injection						8	2	6	598	1
120 - Buffer Overflow	21		21							
134 - Use of Externally-Ctrl. Format String	1		1							
200 - Exposure of Sensitive Info. to an Unauth. Actor	68	68		7, 548	1					
326 - Inadequate Encryption Strength	14	14		326	1					
345 - Insuf. Verification of Data Authenticity	11	11		345	1					
352 - Cross-Site Req. Forgery (CSRF)	51	51		352	1	8	2	6	20, 79	0
472 - External Ctrl. of Assumed-Immutable Web Param.	3		3							
548 - Exposure of Info. Through Dir. Listing	1	1		548	1					
933 - Security Misconfiguration	111	111		933	1					

3.10.1.2 Number of Vulnerabilities

Overall, SAST found the most vulnerabilities, at 823 vulnerabilities. EMPT found the second most vulnerabilities, with 185 vulnerabilities. We provide further information on the number of vulnerabilities found using each method in Table 3.5. The main results for each method are shaded gray, white-shaded columns indicate the results for each of the DAST and SAST tools.

In the first row of Table 3.5, we provide the number of “True Positive (TP) Failures”. For

SMPT, these are failing test cases, while for DAST and SAST these are true positive alerts. We do not have a true positive failure count for EMPT comparable to the failing test cases from SMPT or true positive alerts from DAST and SAST. Unlike SMPT where failing test cases could be assumed to be true positive since poorly written test cases had been removed, EMPT results required additional quality review. We mark the number of true positive failures for EMPT to be Not Applicable (N/A). The “Total” column of Table 3.5 for DAST and SAST “True Positive Failures” is the sum of all true positive alerts for the method.

The second row of Table 3.5 shows the total number of vulnerabilities indicated by the failures. The vulnerability counts are determined by applying our counting rules described in Section 3.6.1.1.2. The same vulnerability could be found by both SAST tools or both DAST tools. The “Total” column for SAST and DAST vulnerabilities accounts for the overlapping vulnerabilities and is the number of vulnerabilities from the method, *not* the sum of the vulnerabilities from the tools. We found the most vulnerabilities using SAST with 823 total vulnerabilities, followed by EMPT with 185 total vulnerabilities. We found 37 vulnerabilities using SMPT, and 23 vulnerabilities using DAST.

The third row of Table 3.5 is the ratio between the number of TP Failures (row 1) and the total number of vulnerabilities (row 2). The ratio of TP Failures to Vulnerabilities is higher for DAST (32.08) than for SMPT (1.58) and SAST (1.12). We discuss the implications of this ratio in Section 3.12.

The fourth row of of Table 3.5, labeled “Vuln. Unique to Method/Tool”, shows the number of vulnerabilities that were only found by each method or tool. It may be helpful to consider the “Vuln. Unique to Method/Tool” as *the number of vulnerabilities we would have missed if we had not used the method or tool*. Similar to the Total Vulnerabilities for SAST and DAST in row 2, the Total columns for SAST and DAST in row 3 indicate the count of vulnerabilities found only by the method. One (1) vulnerability was found by all methods, including both DAST tools and one of the two SAST tools. Specifically, the fact that our instance of OpenMRS was configured such that the default server errors, e.g. 500 errors, revealed sensitive information about the system, which was associated with CWE-7 *J2EE Misconfiguration: Missing Custom Error Page*¹⁷. Ten (10) vulnerabilities were found by both of the SAST tools, but by no other method. The 10 vulnerabilities are included in the 822 Vuln. Unique for SAST (Total), but not in the Vuln Unique to Sonarqube or SAST-2. Similarly, 1 vulnerability was found by both DAST tools but not by other methods. Each method and tool found vulnerabilities that were not found using other methods and tools.

¹⁷<https://cwe.mitre.org/data/definitions/7.html>

Table 3.5: Vulnerability Counts

	SMPT	EMPT	DAST (Total)	ZAP	DA2	SAST (Total)	Sonar	SA2
True Positive (TP) Failures	60	N/A	787	522	265	948	694	254
Total Vulnerabilities	37	185	23	12	13	823	598	235
Ratio: $\frac{TP\ Failures}{Total\ Vulnerabilities}$	1.58	N/A	34.22	43.50	20.38	1.12	1.16	1.05
Vuln. Unique to Method/Tool	11	157	13	8	4	822	588	225

3.10.1.3 Vulnerability Severity

As discussed in Section 3.6.1.1.4, we reviewed vulnerabilities where the same tool or method found more than 20 vulnerabilities associated with the same CWE, and the vulnerabilities were not already labeled as “Low” severity by the tool, i.e. they would otherwise be labeled “more severe”. We also adjusted severity for certain vulnerabilities based on feedback from OpenMRS based on our results. In this section, we describe the results-dependent severity analysis and adjustments.

3.10.1.3.1. Frequently-Occurring Vulnerabilities

Three groups of vulnerabilities were analyzed due to being both frequently-occurring and not otherwise noted as “less severe”. First, 233 vulnerabilities were found using SAST and associated with CWE-52 *Cross-Site Request Forgery*¹⁸. The 233 vulnerabilities were all functions which mapped to HTTP Requests where input parameters were not sufficiently restricted. For example, 220 of these functions used an `@RequestParam` mapping but did not specify which methods (POST, GET, etc) could be used to call the function. Not specifying which types of requests can be used can result in access being granted unintentionally; and the lack of a method parameter can be particularly problematic if the application is using CSRF protection mechanisms¹⁹. The base OpenMRS application did not employ CSRF protection. Although the OpenMRS team is working to employ better CSRF protection which might raise the severity, the “High” or higher severity assigned by SAST tools contrasts with the single vulnerability associated by the DAST tools with CWE-352 *Cross-Site Request Forgery*²⁰ which was a similar vulnerability but was labeled as “Low” severity by the DAST tool. Therefore, we classified the 233 CSRF vulnerabilities found by the SAST tools as “less severe”.

Second, 100 vulnerabilities found using EMPT associated with CWE-79 *Cross-Site Script-*

¹⁸<https://cwe.mitre.org/data/definitions/52.html>

¹⁹<https://docs.spring.io/spring-security/site/docs/5.0.x/reference/html/csrf.html>

²⁰<https://cwe.mitre.org/data/definitions/352.html>

ing²¹. An example XSS vulnerability would be if a field in a patient intake form accepts and saves the value `<script>alert(1);</script>`, then the script is executed when the user navigates to a page where information from the intake form is displayed. The XSS vulnerabilities found via EMPT were all found within a short period of time by students. We therefore consider the risk of exploitability to be high and leave the classification of the vulnerabilities associated with CWE-79 as “more severe”.

Third, SAST-2 found 56 vulnerabilities associated with CWE-404 *Improper Resource Shutdown or Release*²². Of these 56 vulnerabilities, 39 were considered higher severity by the tools while 17 were considered “Low” severity by the tools. All 56 vulnerabilities were instances where a database connection or other resource could potentially be left open for certain executions of the code. The 17 issues the tool considered “Low” severity were on an “exceptional” execution path the tool considered less likely to be identified, e.g. if an secondary failure happened on an unusual path within nested try-catch-finally blocks. The 39 more severe vulnerabilities were considered more likely to be executed system, e.g., if a connection was not inside a try-catch block at all in a function where an error is explicitly thrown under certain conditions. Given the distinctions indicated by the tools themselves that the higher severity vulnerabilities may be easier to exploit, we did not adjust the severity classification.

3.10.1.3.2. Feedback from OpenMRS

After discussion with OpenMRS, we determined that some types of vulnerabilities were low priority for their organization in the context of the application. Specifically, a number of vulnerabilities involved errors which revealed potentially sensitive information about application source code. Since the tool is open-source, the threat posed by these vulnerabilities is minimal. Vulnerabilities associated with error messages that reveal too much information about the system are also classified as “less severe”.

3.10.1.4 Vulnerability Type (OWASP Top Ten)

Table 3.6 shows the distribution of the vulnerabilities found by each method according to the OWASP Top Ten 2021 categories. The Top Ten category assignments are based on the CWEs of the vulnerabilities, using the mapping to the OWASP Top Ten provided by CWE [182], as discussed in Section 3.6.1.1.3. The vulnerability counts for the specific CWE types within each Top Ten category are available in Appendix C.

The leftmost column of Table 3.6 indicates the OWASP Top Ten category. Columns two

²¹<https://cwe.mitre.org/data/definitions/79.html>

²²<https://cwe.mitre.org/data/definitions/404.html>

through five indicate the vulnerabilities that were found for each method. Column six of Table 3.6 shows the total vulnerabilities found within each Top Ten category across all methods. Within each cell, the first value indicates the number of more severe vulnerabilities that were found in the OWASP Top Ten Category. The second value (in parentheses) indicates the total number of vulnerabilities, including both more severe and less severe vulnerabilities.

Table 3.6: Vulnerability Count Per Detection Method

more severe vuln.
(# all vuln., i.e. more and less severe)

OWASP Top Ten (2021) Category	SMPT	EMPT	DAST	SAST	Total
A01:2021 - Broken Access Control	2 ^g (2)	15 (15)	(1)	28 (261)	58 ^g (292)
A02:2021 - Cryptographic Failures	1 (1)	1 (1)	1 (1)	2 (4)	3 (6)
A03:2021 - Injection	5 (5)	119 (119)	11 (11)	24 (58)	150 (184)
A04:2021 - Insecure Design	5 ^g (7)	8 (26)	1 (2)	8 (36)	27 ^g (73)
A05:2021 - Security Misconfiguration	2 (5)	2 (4)	2 (6)	14 (15)	19 (23)
A06:2021 - Vulnerable and Outdated Components					
A07:2021 - Identification and Authentication Failures	13 (13)	10 (10)	1 (1)	2 (2)	17 (17)
A08:2021 - Software and Data Integrity Failures	1 ^h (1)		(1)	10 (11)	11 ^h (13)
A09:2021 - Security Logging and Monitoring Failures	3 (3)	9 (9)			12 (12)
A10:2021 - Server-Side Request Forgery (SSRF)	1 ^h (1)				1 ^h (1)
No Mapping to OWASP Top Ten	1 (1)	1 (1)		54 ^h (436)	56 ^h (438)
Total for Method	32 (37)	165 (185)	17 (23)	142 (823)	329 (1033)

^gOne more severe vulnerability found using SMPT mapped to both A01 and A04 through two different CWEs.

^hOne more severe vulnerability found using SMPT mapped to both A08 and A10 through two different CWEs.

This study is an evaluation of methods to find vulnerabilities which occur due to errors in software code. Tools for finding vulnerabilities in third-party components, such as Software

Composition Analysis (SCA) tools, were excluded from our study. As can be seen in Table 3.6 of the tools examined found vulnerabilities in the OWASP Top Ten Category for Vulnerable and Outdated Components (A06), further suggesting that different methods and categories of methods are useful for finding different types of vulnerabilities.

SMPT was more effective with finding more Identification and Authentication (A07) failures than any other vulnerability type. We found as many Identification and Authentication failures with SMPT as with EMPT. While SMPT found fewer vulnerabilities than EMPT or SAST, most of the vulnerabilities found were more severe. SMPT identified at least one vulnerability in every Top Ten category within scope of the tools in this study, providing better coverage of the Top Ten than other methods.

EMPT was one of the most effective methods for severe vulnerabilities, particularly in the Broken Access Control (A01), Injection (A03), Insecure Design (A04), Identification and Authentication Failures (A07), and Security Logging and Monitoring Failures (A09) categories. Notably, with EMPT we found 119 of the 150 more severe Injection vulnerabilities detected in this study.

DAST was most effective at finding Injection (A03) vulnerabilities relative to other categories of vulnerability. However, DAST found fewer injection vulnerabilities than EMPT and SAST, finding the least number of vulnerabilities overall.

SAST was the most effective method for finding vulnerabilities associated with Security Misconfiguration (A05). Only one vulnerability found by other methods was found by SAST in the entire dataset. Hence SAST should not be seen as something that can substitute for other methods, or be substituted for by other methods. While many of the SAST vulnerabilities were marked as “less severe”, all of the less severe SAST vulnerabilities except the CSRF vulnerabilities were marked as low severity by the tools. Between the two SAST tools there were also differences in the types vulnerabilities found. As can be seen in Appendix C, all 58 Injection vulnerabilities found using SAST, 24 of which were more severe, were found by SAST-2. Sonarqube did not find any Injection vulnerabilities.

3.10.1.5 Effectiveness Comparison with Austin et. al.

A comparison with the previous study by Austin et al. [22, 23] of vulnerability counts for each vulnerability type is shown in Table 3.7. The first column of Table 3.7 indicates the Method, the second column of Table 3.7 indicates whether the data is from the current study or Austin et al. The third column indicates the SUT. As with previous tables, M indicates OpenMRS, the SUT from the current study. E indicates OpenEMR, T indicates Tolven, and P indicates PatientOS; the three SUT from Austin et al. The total vulnerability count calculated for each row is provided in the final *TOTAL (TOT)* column. The remaining columns indicate the vulnerability counts for

each of the OWASP Top Ten categories. In Table 3.7, the row for the current study is shaded in the darkest gray, the total row from the Austin et al. study is in the medium gray color, and the rows for the individual SUT from Austin et al. are in the lightest gray color. For the current study, the total is equivalent to the results from OpenMRS. For Austin et al., the total is the sum of the vulnerabilities found across all three SUT. Austin et al. did not specify whether severity was evaluated in their study, and vulnerabilities such as error messages containing sensitive information about the system (CWE-209) which would have been classified as “less severe” in our current study were included in their vulnerability counts. We therefore assume that the Austin et al. counts reported[22, 23] include less severe vulnerabilities; and the vulnerability counts from the current study in Table 3.7 also include those that are less severe.

Our effectiveness with SMPT in the current study was similar to Austin et al.[22, 23]. Austin et al. found more vulnerabilities in OpenEMR using SMPT compared to the current study, but a similar number of vulnerabilities in Tolven and PatientOS. The distribution of the vulnerabilities across the OWASP Top Ten categories differs between studies. One possible explanation is differences in the test suite. We are using the 2021 Top Ten and the first study by Austin et al. was published in 2011. Some vulnerability types were less prevalent and some vulnerability types may have been considered less severe in 2011, and therefore less well-covered by vulnerability detection methods of the time. For example, Security Misconfiguration (A05) which had no vulnerabilities found by Austin et al., but five vulnerabilities found by the current study, was not included in the OWASP Top Ten until 2013. Similarly, in the Austin et al. test suite, 58 of the 137 test cases (i.e. 42% of the test suite) were targeted towards logging and auditing security controls. The ASVS standard around which our test suite was built only has 2 level 1 controls relating to logging, and only 5 test cases out of 131 (i.e. 4% of the test suite) were related to auditing and logging. The higher number of logging related test cases used by Austin et al. may help explain why Austin et al. were more effective at finding vulnerabilities associated with Security Logging and Monitoring Failures (A09).

Comparing our results against Austin et al.[23, 22] for EMPT is more complicated for methodological reasons. For DAST and SAST the analysis for RQ1.1 was done by a small team of researchers in both the current study and Austin et al.’s work. For SMPT, the procedure was also comparable as indicated by the size of the test suite: 131 test cases in the current study, compared with 137 per SUT for Austin et al.[23, 22]. The procedure for EMPT differed between studies to take full advantage of the data generated by students. As we note in Section 3.6.1, the 229 vulnerabilities in Table 3.5 for EMPT are the result of efforts by 62 students, with additional effort for researcher review. Large numbers of individuals involved in EMPT is not uncommon, for example with bug bounty programs [95]. However, the use of smaller, internal teams such the 6-person team used to apply EMPT to OpenEMR in Austin et al. [23, 22], or even individual

Table 3.7: Vulnerability Type Comparison with Austin Study

M indicates OpenMRS, E indicates OpenEMR, T indicates Tolven, and P indicates PatientOS

Method	Study	SUT	OWASP Top Ten										Not Mapped to Top Ten	TOTAL
			Broken Access Control	Cryptographic Failures	Injection	Insecure Design	Security Misconfiguration	Vulnerable and Outdated Components	Identification and Authentication Failures	Software and Data Integrity Failures	Security Logging and Monitoring Failures	Server-Side Request Forgery (SSRF)		
			A01	A02	A03	A04	A05	A06	A07	A08	A09	A10	NM	TOT
SMPT	Curr. Study	M (Total)	2	1	5	8	5		13	1	3	1	1	37
		Total	4	3	17	4			9		99			136
	Austin et al.	E	3	2	16	2			3		37			63
		T	1			2			4		29			36
		P		1	1				2		33			37
EMPT	Curr. Study	M (Total)	15	1	119	26	4		10		9		2	185
		Total		1	8	1			1				2	13
	Austin et al.	E			8	1			1				2	12
		T												
		P		1										1
DAST	Curr. Study	M (Total)	1	2	11	2	6		1	1				23
		Total	502		221		9							732
	Austin et al.	E	485		221		4							710
		T	17				5							22
SAST	Curr. Study	M (Total)	261	4	58	36	15		2	11			436	822
		Total	93		1190	124	1				22		86	1516
	Austin et al.	E	36		1155	122	1						7	1321
		T	13		35	2								50
		P	44								22		79	145

hackers working alone on EMPT as was done for Tolven and PatientOS is also not uncommon [15, 280]. The high number of students who applied EMPT for RQ1.1 in the current study should not impact the distribution of vulnerabilities across types. However, more participants may have increased the number of vulnerabilities found and to enable comparison between the studies we analyze individual effectiveness for EMPT.

Our results suggest that even at the individual level, the average individual applying EMPT found more vulnerabilities in the current study as compared with Austin et al. [23, 22]. In Austin et al. for OpenEMR a team of 6 individuals spent a combined 30 hours performing EMPT. The team found 8 vulnerabilities in total for a per-person average of 1.33 vulnerabilities. For both Tolven and PatientOS, a single individual applied EMPT for 15 and 14 hours, respectively. Austin et al. found no vulnerabilities using EMPT against Tolven and only 1 vulnerability using EMPT against PatientOS, as shown in Table 3.7 for per-person averages of 0 and 1, respectively. In the current study, EMPT was applied by 62 students and reviewed by 3 researchers, for a total of 65 people involved in collecting EMPT data. We found 185 unique vulnerabilities, of which 165 were more severe. Even including researchers, the average vulnerabilities per-person was 2.85 for all vulnerabilities and 2.54 for more-severe vulnerabilities in the current study. The higher number and per-person average vulnerabilities with EMPT in the current study, as compared to Austin et al., may partially explain the differences in efficiency we will discuss in Section 3.10.2.2.

Austin et al. were more effective with DAST, particularly against OpenEMR, when compared with the current study. Austin et al. found 710 vulnerabilities using DAST against OpenEMR and 22 vulnerabilities in Tolven, as compared with 23 vulnerabilities found in OpenMRS in the current study. We suspect that this difference may be due to differences in how counting rules are applied. The number of true positive alerts appears to be the same or close to the total number of vulnerabilities reported by Austin et al. [23, 22] and the terms “alert” and “vulnerability” appear to be used interchangeably in the prior work. While SAST would also be impacted by any differences in counting rules, as reported in Table 3.5, in the current study the ratio of alerts to vulnerabilities for DAST tools was 34.26 to 1. In contrast, the ratio of alerts to vulnerabilities for SAST tools is 1.12 to 1. The lower ratio for SAST may help explain why the effectiveness of SAST in Austin et al.’s work is more similar to the effectiveness of SAST in the current study, as compared with the DAST results from each study. Austin et al. do not provide their counting rules, and so our hypotheses that counting rules may contribute to the differences between the studies cannot be confirmed. We provide our current counting rules as well as references to how they were derived to assist in future evaluations of vulnerability detection methods.

RQ1.1 - What is the effectiveness, in terms of number and type of vulnerabilities, for each method?

Answer: SAST found the largest number of vulnerabilities overall. However, over half of the vulnerabilities identified by SAST were of low severity. EMPT found the highest number of “more severe” vulnerabilities. Furthermore, if any particular tool or method had been excluded from the analysis, at least 4 and up to 588 vulnerabilities would have been missed.

3.10.2 RQ1.2 - Efficiency

In this section, we discuss the results for our question *How does the reported efficiency in terms of vulnerabilities per hour differ across method?* The data was collected from students, as described in Section 3.7.

3.10.2.1 Efficiency Results

Boxplots for each method’s efficiency in terms of Vulnerabilities per Hour (VpH) are shown in Figure 3.8. EMPT had the highest efficiency (median 2.43 VpH, average 2.22 VpH). SAST had the second highest efficiency (median 1.18 VpH, average 1.17 VpH). SMPT (median 0.63 VpH, average 0.69 VpH) and DAST (median 0.53 VpH, average 0.55 VpH) were least efficient.

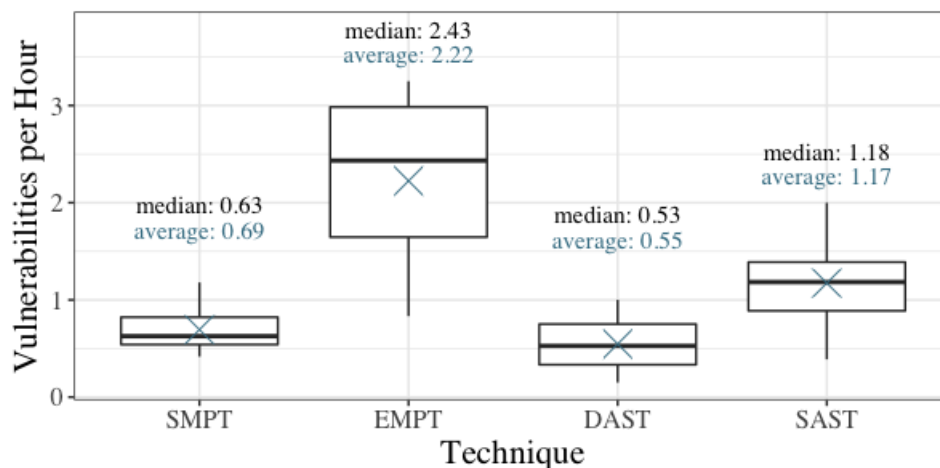


Figure 3.8: Vulnerability Detection Method Efficiency

Table 3.9: VpH Compared to Austin et al.

Study	SUT	SMPT	EMPT	DAST	SAST
Current	OpenMRS	0.63 (median)	2.43 (median)	0.53 (median)	1.18 (median)
		0.69 (avg)	2.22 (avg)	0.55 (avg)	1.17 (avg)
Austin et al.	OpenEMR	0.55	0.40	71.00	32.40
	Tolven	0.94	0.00	22.00	2.78
	PatientOS	0.55	0.07	N/A	11.15

Table 3.8: Games-Howell t-test of Efficiency Scores

Methods	t-value	p-value
DAST-SMPT	0.15(±0.29)	0.499
DAST-SAST	0.63(±0.46)	0.006*
DAST-EMPT	1.68(±0.76)	<0.001*
SAST-SMPT	-0.48(±0.45)	0.034*
EMPT-SAST	-1.05(±0.81)	0.009*
EMPT-SMPT	-1.53(±0.75)	<0.001*

* Statistically significant ($p < 0.05$)

Table 3.8 shows the Games-Howell test results for comparing each pair of methods. As can be seen in the table, EMPT is significantly more efficient than every other method ($p < 0.05$ for all comparisons). SAST is the second-most-efficient method ($p < 0.05$ when compared against both SMPT and DAST). We observed no statistically significant difference in the efficiency of SMPT compared to the efficiency of DAST. In Table 3.8, we round the p-values to the thousandths position. The p-value for the comparisons between DAST and EMPT and between EMPT and SMPT was less than one thousandth.

3.10.2.2 Efficiency Comparison with Austin et al.

Table 3.9 compares the efficiency in the current study with the efficiency reported by Austin et al. Austin et al. reported the time it took their group of researchers, on average, to find each vulnerability. To present similar values for our study, Table 3.9 indicates the median and average (avg) across the groups performing the task.

The differences in efficiency are not only due to differences between student performance and researcher performance. When applying SAST for RQ1.1 in the current study, researchers' efficiency was estimated at 18-32 VpH, comparable to Austin et al. However, researcher efficiency with ZAP, the more efficient of the two DAST tools, was 1.8 VpH - far below the minimum VpH (22.00) reported in Austin et al. One possible cause of the discrepancy, particularly with DAST efficiency, could be our focus in unique vulnerabilities as defined by our counting rules in Section 3.6.1.1.2 compared with alerts or true positive failures. As shown previously in Table 3.5, DAST had the highest ratio of true positive failures. Similarly, the differences in efficiency for

EMPT may be driven more by vulnerability count than the length of time to apply the method.

RQ1.2 - How does the reported efficiency in terms of vulnerabilities per hour differ across method?

Answer: EMPT was the most efficient (2.22 VpH), followed by SAST (1.17 VpH). SMPT (0.69 VpH) and DAST (0.55 VpH) were least efficient.

3.10.3 RQ1.3 - Other Factors to Consider when Comparing Tools

We performed qualitative analysis on students' free-form responses to answer our research question *What other factors should we consider when comparing methods?*. We discarded the response from 1 of the 13 teams where both reviewers considered the text to be confusing and self-contradictory. The results below are from the responses of the remaining 12 teams. We use "at least" to emphasize that our counts are conservative; we did not include instances where the author's intent was unclear. Some teams discussed "automated" and "manual" categories of methods rather than the further subdivision used in the rest of this paper (i.e., EMPT and SMPT are "manual" methods, while SAST and DAST are "automated" methods). We discuss our results for RQ1.3 in terms of automated and manual methods when those terms are used by one or more teams. Most teams included some discussion of efficiency and effectiveness, which we do not include here; focusing, instead, on concepts that were not covered previously.

3.10.3.1 Effort

Summary: *Effort was one of the most discussed topics by students. People do not like to do any more work than necessary. Effort was seen as a disadvantage with manual methods, discussions of effort for automated methods were mixed.*

Every response mentioned human effort beyond VpH. Effort was perceived as a disadvantage for manual methods (SMPT and EMPT) more than automated ones (SAST and DAST). Automation itself is seen as an advantage by at least two teams, one of which explicitly stated "*Dynamic application security testing is better than manual blackbox testing because you can automate the tests*". Eight (8) teams mentioned effort as a disadvantage of one or both of the manual methods, while 0 teams mentioned advantages relating to effort for either of the manual methods. In contrast, for one or both automated methods, 4 teams mentioned effort as a disadvantage, 3 teams mentioned effort as an advantage, and 2 teams mentioned both advantages and disadvantages.

3.10.3.2 Time

Summary: *Although the amount of time spent on an activity was a component of our efficiency metric (VpH), time was discussed separately from efficiency. Manual methods were seen as requiring more time. For automated methods, some teams considered time an advantage while others considered time a disadvantage. Additionally, students conjectured that the efficiency of each method may change if the methods were applied over a longer timeframe.*

Similarly to effort, time was frequently seen as a disadvantage for manual methods, particularly SMPT. At least 10 of the 12 teams mentioned time in some way. Eight (8) teams explicitly mentioned time spent on manual tasks, while 5 of those teams also explicitly discussed the time for tools to run, even though tool running time is not active time for the analyst. Additionally, 8 teams mentioned time as a disadvantage for manual methods, with 4 teams specifically mentioning time as a disadvantage for SMPT and 3 teams mentioning time as a disadvantage for EMPT. One of the teams who considered time a disadvantage for SMPT considered time an advantage for EMPT. No other team noted time as an advantage for any manual method. Responses for automated methods were more mixed, with time seen as an advantage for SAST by 4 teams and for DAST by 1 of the 4. However, 2 teams considered time a disadvantage for SAST and 3 teams considered time a disadvantage for DAST.

At least 3 teams also noted that they anticipated that a method's effectiveness and efficiency would change if the methods were applied over a longer period of time. For example, one team noted that for SMPT "*our guess is with time it will get even more difficult to come up with black box test cases manually thus giving lower efficiency eventually*". In contrast, one team claimed that with DAST "*... if time and memory are not an issue you can run a local instance of the application and fuzz it for years*".

3.10.3.3 Expertise

Summary: *Many types of expertise are needed to apply vulnerability detection methods, particularly EMPT. Similar to findings from other works [137, 136, 280], students noted that EMPT in particular requires different types of expertise including technical expertise, security expertise, and expertise with the SUT.*

Overall, at least 8 teams commented on the role of expertise. Of the 8 teams who mentioned expertise, only 3 mentioned expertise in the context of tool-based methods, while 6 mentioned expertise when discussing EMPT, and one team mentioned expertise when discussing SMPT and manual methods generally. Expertise was not clearly an advantage or disadvantage, with only 3 of the 8 teams who mentioned expertise suggesting that the expertise required to use a method was a disadvantage, with the remaining 5 teams not clearly noting expertise as an

advantage or disadvantage. Several specific types of expertise were discussed, three of which, technical, security, and SUT expertise, are similar to the types of expertise highlighted in related work [137, 136, 280]. At least 1 team commented on the role of technical expertise in applying EMPT, 1 team commented on security expertise required for EMPT, 4 teams commented on the role of SUT expertise for EMPT, and 1 team commented on the role of SUT expertise for SMPT.

RQ1.3 - What other factors should we consider when comparing methods?

Answer: The three most frequently discussed factors (other than Effectiveness and Efficiency) were Effort, Time, and Expertise. Effort and time were seen as a disadvantage of manual methods. Some teams considering effort or time an advantage of automated methods, while others considered them a disadvantage. Expertise was associated with manual methods, particularly EMPT, more than automated methods.

3.11 Limitations

We discuss the limitations to our approach in this Section. We group these limitations as threats to Conclusion Validity, External Validity, Internal Validity, and Construct Validity [49, 91, 289]. Threats to validity frequently involve the treatments and outcome measures used in the study as well as the higher level constructs the treatments and outcomes represent [49, 289, 241]. In our study, the two primary outcome constructs we intended to observe were effectiveness (RQ1.1) and efficiency (RQ1.2). The specific proxy measures we use determine the outcome are the number and type of vulnerabilities (RQ1.1) and Vulnerabilities per Hour (RQ1.2). The cause construct (independent variable) is the vulnerability detection method being used. Our treatments to represent this cause construct are the four categories of vulnerability detection methods. These outcomes and treatments were previously used by Austin et al. [22, 23].

3.11.1 Conclusion Validity

Conclusion Validity is about whether conclusions are based on statistical evidence [49, 289]. While we have empirical results for RQ1.1, a single case study is insufficient to draw *statistically significant* conclusions for effectiveness. The measures used to evaluate effectiveness in RQ1.1 are based on the number of vulnerabilities found by applying each method thoroughly and systematically. Measuring effectiveness with statistical significance would require the application of all four methods to at least 10-20 additional applications [148]. Applying all methods

to 10-20 similarly-sized SUT is impractical given the effort required to apply these methods to a single application. To mitigate this threat to validity, we performed extensive review of the vulnerability counts, using the guidelines in Section 3.6.1.1, and at least two individuals were involved in the review process for each method to verify the accuracy of the results. For efficiency (RQ1.2) the measure used, VpH, was evaluated by having more individuals apply the method to a subset of the application, enabling us to evaluate the results with statistical significance.

3.11.2 Construct Validity

Construct Validity concerns the extent to which the treatments and outcome measures used in the study reflect the higher level constructs we wish to examine [49, 289, 241]. Our measures for RQ1.1, the number and type of vulnerabilities, are commonly used measures of (in)security in academia and industry [151, 68, 204, 203, 202, 201], including by the U.S. National Institute of Standards and Technology (NIST) Software Assurance Metrics and Tool Evaluation (SAMATE) program discussed in Section 3.2. Raw vulnerability counts do not fully capture the construct of effectiveness, since some vulnerabilities may be considered more important than others. As mentioned in Section 3.6.1.2, we excluded tool alerts which were marked as insignificant or inconsequential, assuming that these alerts would not be of interest to practitioners. Additionally, we use the OWASP Top Ten categorization to summarize our data, and indicate the severity of vulnerabilities found as described in Section 3.6.1.1.4.

Efficiency was measured in terms of vulnerabilities per hour. This measure allowed us to run a controlled experiment in which teams of students performed each method on a subset of the application, and we could determine whether differences in efficiency were statistically significant. As discussed in Section 3.10.3.2, factors such as the length of time required to apply a method may also be helpful in understanding efficiency. Applying SAST and DAST more comprehensively as part of RQ1.1 required over 20 hours per tool for both SAST tools as well as DAST-2. In a class where vulnerability detection was only part of the curriculum, it was not reasonable to expect students to spend 20 hours on each method. Adding RQ1.3, which was not included in Austin et al.'s work [22, 23], allowed us to better understand how other factors such as time spent applying a method were perceived by students, helping to mitigate this threat to validity.

3.11.3 Internal Validity

Internal Validity concerns whether the observed outcomes are due to the treatment applied, or whether other factors may have influenced the outcome [49, 91]. Running DAST based on more

test cases may have found more vulnerabilities. However, the resources available to our team were not significantly less than other small organizations, suggesting that resource limitations may be a factor to consider when using DAST. Additionally, with OWASP ZAP we leveraged the tool's spider capability to expand on the 6 inputs, which helped mitigate this threat to validity by increasing system coverage. Other comparisons of vulnerability detection methods have also used a spider [74, 248].

Another threat to internal validity for this study is that the student data used for RQ1.2 is self-reported. The student data aligns with the experiences of the research team, but self-reported estimates of the time to complete a task are not necessarily representative of the actual time to complete a task. However, perceived time to complete a task must also be considered when making decisions on which vulnerability detection methods are used. Additionally, as shown by RQ1.3, students' reported numeric efficiency was not necessarily indicative of the time and effort the students perceived was required for each method.

Another limitation with RQ1.2 is posed by equipment constraints for the graduate level class. Additional equipment was used to mitigate this risk for RQ1.1. To mitigate the risk that memory-related processing issues would negatively impact student efficiency (RQ1.2), the first author performed a SAST scan of all modules in advance and directed students to modules which would not be impacted by equipment constraints. Furthermore, students were instructed to only report time spent reviewing results, not time spent trying to get the scan to run.

The researchers performing qualitative analysis for RQ1.3 may have had biases which present threats to internal validity. Researcher biases may also have impacted the vulnerability review processes, and analysis of true and false positives from the results of vulnerability detection tools for RQ1.1. For this reason, two individuals jointly performed the qualitative analysis, and the vulnerability review was either performed by two independent individuals or performed by one individual and audited by a second individual depending on method.

Finally, although both DAST tools examined in RQ1.1 and RQ1.2 were the same, we did not use the Sonarqube SAST tool for RQ1.2, using a proprietary tool (SAST-3) that had been used in the course previously. Sonarqube may have been more or less efficient or effective as compared with SAST-3, which would influence our results. Student data was reported in aggregate and we only have the average efficiency of SAST-2 and SAST-3 combined. However, estimated researcher efficiency using Sonarqube was 22 VpH while estimated researcher efficiency using SAST-2 was 18 VpH, suggesting that tool differences may play less of a role in SAST compared with other factors such as expertise. In RQ1.2 we control for expertise by comparing efficiency scores from the same group of individuals.

3.11.4 External Validity

External Validity concerns the generalizability of our results [49, 91, 289]. Our results may not generalize to software that is not similar to the SUT and the results may not generalize to other systems. For example, we know that a strongly-typed, memory-safe language such as Java, by design is likely to have fewer memory-allocation vulnerabilities, such as buffer overflow, when compared with code in a non-memory-safe language such as C[52, 190]. As discussed in Section 3.4.1, OpenMRS is built with commonly-used languages (e.g. Java, Javascript) and frameworks (e.g. Spring) and comparable in terms of size and development practices to other systems in its domain. Additionally, many of our results are similar to other studies. For example, in recent SATE comparisons [68], the highest precision rates for SAST tools were 78-94% in tests against Java applications, similar to those for the study and higher than we expected based on other prior work as we will discuss in Section 3.12.3.2.

The tools used in this study may also not be representative of DAST and SAST tools generally. Our results as well as those of the SATE reports[68, 204, 203] suggest that the effectiveness of SAST tools may vary. As noted both in our own experience and by students, the two DAST tools were very different in terms of ease of use. We used two tools that are in prevalent use in industry when performing each method to mitigate and understand possible biases introduced by tool selection.

A related threat to external validity is that we are performing a scientific experiment in an academic setting, rather than in industry. We do not think the differences between our experiment and industry would impact our results and have worked to minimize differences. For example, the assignments were designed to mitigate the risk that differences from industry practice would impact the efficiency scores. When applying SAST tools in industry, once alerts are classified as true or false positive, practitioners are more concerned with resolving the true positive alerts than with handling false positive alerts, as supported by studies such as Imtiaz et al [133]. However, true positive vulnerabilities require more analysis since the alert must be resolved, while false positives can be ignored. In the SAST assignment²³, students were required to analyze at least 10 alerts. To avoid incentivizing students to reduce their workload by classifying alerts as false positives, students were instructed “If you have more than 5 false positives, keep choosing alerts until you have 5 true positives while still reporting the false positives”. Similarly, in our experience²⁴, a cursory review of test cases developed by less experienced testers is necessary in some industry contexts to ensure the resulting test suite can be run efficiently and effectively. While our review of student test cases described in Section 3.6.1.2.1 was more extensive, reviewing the test cases for RQ1.1 was intended to ensure

²³the full text of the assignment is available under Project Part 1 in Appendix B.3

²⁴the first author has over 2 years of industry testing experience

a more accurate test process and resulting vulnerability count. Since time was not considered in RQ1.1, the additional time spent on reviewing test cases for RQ1.1 would not impact the results.

3.12 Discussion

Section 3.12.1 and Section 3.12.2 should be considered together. In Section 3.12.1, we provide examples of how our results may help inform practitioners' decisions based on their *objectives*, particularly for projects in a similar domain to OpenMRS. In Section 3.12.2 we discuss how the availability or limitation of *resources*, specifically Expertise, Time, and Equipment, may also impact which method should be used. There may be tradeoffs between methods when practitioners focus on one *objective* over another. A manager may want to avoid manual methods in order to reduce the perceived effort for their team (Section 3.12.1.4). However, in our context automated methods were less effective in terms of the coverage of different vulnerability types and the severity of vulnerabilities found (Section 3.12.1.3). There may also be tradeoffs between *objectives* and *resources*. For example, as we note in Section 3.12.1, we found EMPT to be very effective at finding Injection vulnerabilities. However, if an organization does not have enough individuals with sufficient expertise to apply EMPT effectively as described in Section 3.12.2, practitioners may need to look to other methods.

In Section 3.12.3 we go over findings that have additional implications relevant to research and other evaluations of vulnerability detection methods. We would encourage any researcher comparing vulnerability detection methods to also be aware of our findings in Sections 3.12.1 and 3.12.2.

3.12.1 Organizational Objectives

We provide four examples of how our results might inform practitioner decisions on which vulnerability detection methods to use.

3.12.1.1 Specific Vulnerability Types (Effectiveness)

Practitioners may be more concerned about a certain type or class of vulnerabilities. For example, as seen in Table 3.6, EMPT would be a good choice for someone working on an open-source Java-based medical application such as OpenMRS where Injection vulnerabilities such as XSS are a concern. As noted by other comparisons of vulnerability detection tools [68, 28], which tool is most effective may vary across domains. Practitioners should look to vulnerability detection method evaluations in their domain.

If an organization is trying to target a specific type of vulnerability, they should focus their vulnerability detection efforts on methods that are effective at finding that type of vulnerability in systems from their domain. For example, a project similar to OpenMRS looking to find Injection (A03) vulnerabilities may benefit from EMPT as seen in Table 3.6.

3.12.1.2 Coverage (Effectiveness)

While EMPT performed particularly well in our context, SMPT provided higher coverage across the OWASP Top Ten 2021 categories, as shown in Table 3.6. On the other hand, as seen in the comparison with the prior work by Austin et al. [22] shown in Table 3.7, if the goal of the practitioner is to thoroughly cover Logging and Monitoring concerns, a test suite based on Level 1 of the ASVS may not be preferable since the first level of the ASVS only contains 2 controls for logging.

Our results suggest that if a practitioner needs a vulnerability detection method that effectively covers important types of vulnerabilities, a more systematic method, such as SMPT, may be more effective.

3.12.1.3 Automation (Efficiency)

When considering automated tools, practitioners should note that automated methods may not inherently be more efficient than manual methods. An organization may choose to use automated tools for other reasons such as the need to integrate automated tools with continuous deployment pipelines [238]. Our results suggest that manual methods are comparable to or better than automated methods in terms of efficiency.

Our results suggest that, if an organization is considering whether to use an automated method over a manual one, they should not assume that the automated method will be more efficient.

3.12.1.4 Perceived Effort and Ease of Use (Other Factors)

Two of the most-frequently-mentioned concepts in the students' free-form responses, *effort* and *expertise*, are associated with the broader concept of Perceived Ease of Use [64]. In the same assignment where students reported spending more time to find fewer vulnerabilities with SAST and DAST as compared with EMPT and, to a lesser extent, SMPT; students also claimed they considered time and effort to be a disadvantage of manual methods. As we discuss in

Section 3.10.3, *time* and *effort* was predominantly seen as negative for manual methods but views of time and effort were mixed for automated methods. Similarly, *expertise* was associated with manual methods (SMPT and EMPT) more than automated methods (DAST and SAST). The actual times recorded for manual methods were, on average, no longer than the times recorded for automated methods. Hence our findings suggest that time was *perceived* as a disadvantage of manual methods even if they actually required no more time than automated methods. Pfahl et al's [226] interviews of practitioners also found that exploratory testing was perceived as being less easy-to-use and requiring more skill. However, studies of SAST tools have also found Ease-of-Use concerns with SAST [263]. As noted by Gonçalves et al.[114], there is insufficient empirical research on the cognitive load of review-related tasks such as software testing. While we cannot make universal claims about all automated tools, practitioners looking for the “easiest” solution may wish to minimize their use of manual methods.

Our results suggest that if an organization is looking for a method that will be perceived as requiring less effort, they may want to avoid manual methods (SMPT and EMPT), regardless of actual efficiency.

3.12.2 Resources to Consider

The resources below represent factors that should be considered when selecting a vulnerability detection method for a system such as OpenMRS. Two of the three resources were highlighted by student responses in RQ1.3, while the third resource provided a much more severe limitation on our experiment than anticipated.

3.12.2.1 Expertise

As noted by the students in RQ1.3 and supported by prior work [137, 136], expertise plays a role in vulnerability detection, particularly EMPT. The effectiveness of the students as shown in Table 3.6; as well as their efficiency shown in Figure 3.8 is promising. Students with an introductory knowledge of Security were efficient and effective with EMPT. Anecdotally based on our experience with RQ1.1 as well as in student responses in RQ1.3, experience may also impact the efficiency and effectiveness of automated methods more than we expected.

Our findings support related work suggesting that the availability of analysts with security expertise should be considered when selecting a method. EMPT is particularly known to be influenced by analyst expertise.

3.12.2.2 Time

As noted by the students in RQ1.3, the amount of time an analyst has available to apply a method may influence the efficiency and effectiveness of the method. While EMPT requires more expertise, little or no preparation is needed. SMPT, EMPT, and DAST take more time to setup. As noted by the students, as discussed in Section 3.10.3.2, some methods such as DAST may perform better if practitioners have an extended timeframe in which to apply the method. In contrast, as discussed in our comparison with Austin et al. in Section 3.10.1.5, a single individual performing EMPT for a longer period of time did not find more vulnerabilities than were found in a shorter timeframe.

The amount of time available to apply a method should be considered when selecting the method. DAST, in particular, may benefit from a longer timeframe.

3.12.2.3 Equipment

We found that evaluation of a “large-scale” system required more computing resources than expected for both SAST and DAST. As discussed in Section 3.11.3, students were only able to run SAST on smaller modules of OpenMRS when using the base VMs allocated for the class. Equipment constraints also played a role in determining how researchers could run DAST-2 systematically for RQ1.1. Austin et al., as well as other studies of industry tools [17, 248, 28] do not mention equipment constraints. Where the equipment used in the experiment is mentioned [17], it is implied that the tools were able to be run on machines similar to the VMs used by students of the graduate class. While OpenMRS is “large” for an evaluation of vulnerability detection methods, OpenMRS is less than 4 million lines of code - smaller than many industry software systems [71, 19]. Equipment constraints should be considered by practitioners when considering DAST and SAST.

Equipment constraints may influence the effectiveness and efficiency of DAST and SAST on realistic systems. In some cases, applying DAST and SAST may not even be feasible due to equipment constraints.

3.12.3 Implications for Evaluating Vulnerability Detection Methods

The resource concerns highlighted in Section 3.12.2 should be considered not only by practitioners but by researchers evaluating vulnerability detection methods. Our results also have several implications specific to future evaluations of vulnerability detection methods

3.12.3.1 True Positive Failure Count vs Vulnerability Count (Ratio)

We start with an observation that is not discussed in much of the related work, but which may impact the results of any study comparing vulnerability detection methods. The ratio between the number of tool alerts or failing test cases, i.e. “true positive failures”, and the number of vulnerabilities varies across tools and methods. As can be seen in Table 3.5, particularly for DAST tools, the number of alerts was many times the number of vulnerabilities found. The high ratio of alerts to vulnerabilities is consistent with the finding from Klees et al.’s [151] work with fuzzers that “ *‘unique crashes’ massively overcount[s] the number of true bugs*”. SMPT and SAST, on the other hand, had a much lower ratio of True Positive Failures to Vulnerabilities when compared with DAST.

More research is needed to fully understand the impact of having a higher or lower number of failures per vulnerability. Similarly, for a developer using a SAST tool built into their IDE while writing code, the actual vulnerability count and consequently the difference between the SAST alert count and final vulnerability count may not have much impact at all. If additional alerts or other failures present more information about the vulnerability itself, having more failures per vulnerability may be helpful in triaging and fixing the vulnerability. However, reviewing and analyzing additional failures takes time and may reduce efficiency. In another example, if a practitioner is using the overall alert count or vulnerability count to determine the cybersecurity risk of an application, such as for insurance estimates [62], the difference between alert count and vulnerability count may have a significant impact. Researchers should also be cautious when using alerts, failing test cases, or similar true positive failures as a proxy for the number of vulnerabilities found in a system.

A consistent set of counting rules should be used when comparing the effectiveness of different tools or methods. It should not be assumed that tools or methods use the same counting rules.

3.12.3.2 SAST tools had fewer False Positives than expected.

High False Positive counts have historically been considered a drawback of SAST tools[133, 143, 248, 262, 118]. Our results suggest that, at least for our context, SAST produced few false positives. The high precision of SAST tools for this study is similar to results from recent SATE events [68]. More research is needed to better understand the circumstances under which a lower false positive count may generalize and the relationship between the perception that SAST tools produce large numbers of false positives and the actual false positives produced by tools.

Our research supports other evaluations that indicate some SAST tools have low false positive counts when applied to Java applications. The lower false positive count opens up new questions about why the percentage of false positives is perceived as a problem for SAST methods, and whether false positive counts have improved in other contexts.

3.13 Conclusions

The motivation for this paper came from practitioner questions about which vulnerability detection methods they should use and whether vulnerability detection could be fully automated. After ten years, with a changing vulnerability landscape, and many improvements in vulnerability detection methods such as the more common use of symbolic execution and taint tracing in SAST tools[171, 35] results from previous work by Austin et al. were no longer assured to hold true. We replicated the previous work, this time examining at least two tools for each category of method. The main finding of Austin et al. still holds - each approach to vulnerability detection found vulnerabilities NOT found by the other methods. If the goal of an organization is to find “all” vulnerabilities in their system, they need to use as many methods as their resources allow.

Our work also highlights how security assessments that rely on number of vulnerabilities in a system can be heavily influenced by how vulnerabilities are collected and counted. As shown in Table 3.5, the number of “True Positives” produced by each different vulnerability detection method differed from the number of actual vulnerabilities based on our standardized counting method. Furthermore, each method found different numbers of vulnerabilities. Hence any one approach for gathering data without a method for standardizing vulnerability count would have produced different results. These differences when applying techniques within the same system suggest that when using vulnerability-count-based measures as part of an experiment or assessment, researchers and practitioners must ensure that bias is not introduced via subtle or significant differences in vulnerability detection methods.

CHAPTER

4

VULNERABILITY EXPLOITABILITY ASSESSMENTS

The U.S. National Vulnerability Database (NVD) has reported annual increases in the number of reported vulnerabilities every year since 2016¹. As found in our own work in Chapter 3 even a single system can have a large number of vulnerabilities that need to be triaged. Due to the volume of vulnerabilities that need to be addressed in software systems; practitioners prioritize their efforts by addressing the most pressing security risks first [15].

Understanding and assessing the *exploitability* of vulnerabilities is a key component in risk-based vulnerability prioritization [142, 10, 34]. A commonly-referenced definition of “exploitability” is the definition from the Common Vulnerability Scoring System (CVSS) where exploitability is defined as “the ease and technical means by which the vulnerability can be exploited” [59]. However, researchers have proposed and evaluated a wide range of methods for assessing the “exploitability” of vulnerabilities. Exploitability assessment can include a variety of methods, from manual/expertise-based assessment of the Base Exploitability CVSS score, to program analysis tools and machine learning models designed to automatically analyze software vulnerabilities and determine their exploitability.

¹<https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time>

Literature surveys to date (e.g. [162, 167, 153, 266, 258]) have focused on specific categories of tools or techniques, such as surveys of machine learning or binary analysis tools for vulnerability detection and assessment, within which “exploitability assessment” is just one of the tasks that such tools are used for. Instead of focusing on the technical details of the techniques and tools, which have already been thoroughly covered, we look at the task of exploitability assessment itself.

The goal of this research is to assist practitioners and researchers in understanding existing methods for assessing vulnerability exploitability through a survey of exploitability assessment literature.

Our survey is based on two key concepts: *vulnerabilities* and *exploits*. As noted in Chapter 2, we use the definition of *vulnerability* from the NVD: “a weakness in the computational logic (e.g., code) found in software and hardware components that, when exploited, results in a negative impact to confidentiality, integrity, or availability” [198]. In software security, the term *exploit* often refers to the series of steps used to exploit a system, particularly when these steps are documented as computer code [75, 31, 281, 43, 272, 98]. Documentation of exploits typically includes the inputs required to trigger specific program states, which results in negative security consequences. In practice, exploit documentation frequently does not include explicit information about the internal states of the vulnerable program. Exploits may take advantage of one or more *vulnerabilities*. More than one exploit may be possible using the same vulnerability, and more than one vulnerability may be used in an exploit.

In this survey, we focus on *software* vulnerabilities. Assessments of hardware, user, or other vulnerability categories [221] are outside the scope. We address the following research question:

- **RQ: How is the *exploitability* of software vulnerabilities assessed?**

We performed a methodical search of the academic literature to identify papers on exploitability assessments for software vulnerabilities. We then reviewed the papers and identified characteristics shared between different software vulnerability assessment methods and characteristics that can be used to differentiate between these methods. We then categorized the research based on those characteristics.

The contributions of this work include:

- an list of academic literature focusing on exploitability assessment for software vulnerabilities
- an analysis of characteristics of exploitability assessment of software vulnerabilities that illustrate similarities and differences between methods

In Section 4.1, we explain key terms and discuss other surveys that look at similar topics. In Section 4.2 we provide an overview of CVSS, a standard assessment method commonly referenced across multiple categories within the literature. In Section 4.3, we provide the methodology for our search for relevant papers. In Section 4.4 we provide an overview of our findings. Sections 4.5 through 4.8 discuss the groups of exploitability assessment methods in greater detail., we discuss our findings. In Section 4.9, we discuss the study’s limitations, followed by a discussion in Section 4.10. We conclude with Section 4.11.

4.1 Background and Related Work

In Section 4.1.1 we provide definitions for common terms that will be reused throughout the paper. Then, in Section 4.1.2 we provide an summary of recent surveys which partially overlap with our work and highlight the differences between the studies.

4.1.1 Common Terms

In this section we include brief definitions of common terms which are frequently used throughout the chapter and not defined previously

Exploit Signature Exploit signatures are a way of identifying specific exploits, i.e. of identifying a specific set of steps that can be used by an attacker to achieve their goals [9, 139]. Exploit signatures are commonly used by intrusion detection systems which scan network traffic, and by anti-malware programs (also referred to as anti-virus programs) which scan computer systems to find signs that a malicious actor may have attacked the system [9, 10, 192]. For network analysis, an example signature would be a particular pattern found within network messages that are always sent as part of the exploit. For anti-malware programs, an example signature would be a particular pattern in bytecode that indicates a program being used by the exploit.

ExploitDB ExploitDB is a publicly-available repository of exploits submitted and maintained by penetration testers and security professionals [87].

OSVDB The Open Source Vulnerability Database (OSVDB) was a publicly accessible vulnerability database active between 2002 and 2016 [155, 65, 112]. The OSVDB included information such as CVSS scores from the NVD, as well as information from other databases and information specifically curated for OSVDB [144]

4.1.2 Related Surveys

Researchers have performed surveys of the literature on vulnerability assessment and related themes. Table 4.1 summarizes six related works whose survey partially overlaps with ours in at least one of the five major categories we identify. The first two columns of Table 4.1 indicate the reference and year. The next five columns indicate the five major categories in this survey, with a “P” indicating that the survey includes some of the same literature as our survey for that category. We provide an overview of these categories in Section 4.4. The “Summary” column (column 9) provides a brief overview of the survey, while the last column includes a brief explanation of how we expand upon and differ from the prior work, often within the same category as we overlap. As can be seen in Table 4.1 the greatest overlap between our study and prior work is the study by Pendleton et al. Our own categorization of the studies in our survey is heavily influenced by Pendleton et al. [221], which we discuss further in Section 4.3.4. However, there are also many key differences between our study and theirs. Over half (49 out of 76) of the papers we survey were published after 2016, the year of publication for the Pendleton et al. study. Additionally, there are differences in scope which also reduce the overlap. Pendleton et al focus on metrics, while we focus on the assessment methods used to produce scores like metrics. Pendleton et al examine system-level metrics while our primary focus is on vulnerability-level assessment. Finally, we focus on the topic of exploitability, a subset of security. Focusing on exploitability allows us to examine assessments in greater detail than would be feasible if we broaden our scope to incorporate all possible security-related assessments.

As shown in Table 4.1, the remaining surveys overlap at-most with one category. In most cases such as Shoshitaishavili et al [258], we only partially overlap within the same category.

4.2 CVSS Components and Sub-Scores

The Common Vulnerability Scoring System (CVSS) is used or referenced by over half of the papers in this survey. The prevalence of CVSS in exploitability literature influenced our survey process. We provide a high-level description of CVSS in Chapter 2. To facilitate further discussion of exploitability assessment, we provide additional details of CVSS, focusing on the exploitability-related components and subcomponents. The full specification for CVSS and other CVSS documentation may be found on the CVSS website <https://www.first.org/cvss>. The current version of CVSS as of this research was v3.1.

Table 4.1: Related Surveys

Ref.	Year	Partial Overlap				Summary	How we expand upon their work (Differences)
		CVSS	Det	Prob-LM	Prob-O		
[221]	2016	X	X	X	X	Survey & taxonomy of system-level security metrics, including CVSS-based and network-topology-based metrics	We focus on vulnerability-level metrics, rather than system-level metrics.
[258]	2016		X			Describe and implement binary analysis techniques from prior security research, including that involve binary analysis	We examine a wider range of exploitability assessment techniques, including probabilistic techniques. We focus on defensive, rather than offensive, techniques.
[167]	2022		X			Survey on binary exploitation in Industrial Control Systems including work on the causes and consequences of exploitation, categories of exploit, known cyber incidents, and mitigations.	We focus on exploitability assessment for individual vulnerabilities rather than systems, exploring a wider range of contexts.
[266]	2021			X		Overview of academic research on learning models for vulnerability assessment including detection, exploitability, and propagation	We look at vulnerability exploitability assessment methods beyond learning models.
[153]	2022			X		Review academic literature on machine learning frameworks for security and bug-finding tasks, using program-analysis-based features.	We examine learning models for vulnerability exploitability assessment in relation to other forms of exploitability assessment, rather than other (non-exploitability) learning models.
[162]	2022			X		Categorize and describe the key tasks performed, data features used, and evaluation methods for vulnerability assessment models using machine-learning, deep-learning, and natural-language-processing.	

4.2.1 Exploitability-related CVSS components

CVSS v3.1 has three metric groups: Base, Temporal, and Environmental [59]. The Base Metric group includes characteristics of the vulnerability that are constant, regardless of the context. The Temporal metric group captures the characteristics of a vulnerability that can change over time, such as whether an exploit has been made publicly available. The Environmental metric group includes characteristics of a vulnerability that may vary based on the environment, such as whether a vulnerable library is being used by an application that manages sensitive data or an application that stores data that is not sensitive, such as cat pictures.

In CVSS v3.1, the Base score is derived from two subscores, Exploitability and Impact, then adjusted based on a third Scope score. For this survey, within the Base score we only consider the Exploitability score and its sub-components to be “Exploitability-related”. The Exploitability subscore [59] includes four components: Attack Vector (AV) indicates how close an attacker must be to be able to exploit the vulnerability; Attack Complexity (AC), indicating whether exploiting the vulnerability requires conditions that the attacker does not directly control; Privileges Required (PR), which describes the level of logical access needed to exploit the vulnerability; and User Interaction (UI) which indicates whether a user, other than the attacker, is required to provide inputs to the application for the attacker to exploit the vulnerability. In CVSS v3.1, the Base Exploitability subscore is calculated as $8.22 \times AV \times AC \times PR \times UI$.

The Temporal group includes the Exploit Code Maturity (E) metric, which “measures the likelihood of the vulnerability being attacked, and is typically based on the current state of exploit techniques, exploit code availability, or active ‘in-the-wild’ exploitation” [59]. The other Temporal metrics, Remediation Level (RL) and Report Confidence (RC), are not directly related to exploitability.

The Environmental metric group [59] includes modified scores of all Base metrics, including the Exploitability metrics: Modified AV (MAV), Modified Attack Complexity (MAC), Modified Privileges Required (MPR), and Modified User Interaction (MUI). The remaining Environmental metrics are based on Impact. The modified scores update the Base Metric subscores based on environmental factors such as use of a non-default configuration setting [59].

4.2.2 Differences in Exploitability-related subscores between CVSS v3 and CVSS v2

Most of the publications surveyed that use or refer to CVSS, leverage CVSS v2 since CVSS v3 was not released until 2015 [57, 243] and was not officially supported in the NVD until 2019 [197]. In the Base Exploitability group of CVSS v2 [57], the Attack Vector (AV) metric was referred to as the Access Vector (also abbreviated AV) metric, but was scored similarly. The AV metric in

CVSS v2 did not distinguish between logical access and physical access to the machine. The other AV levels (Adjacent and Network) remained the same. In CVSS v2, the Attack Complexity (AC) and User Interaction (UI) metrics were combined in the Access Complexity measure (also abbreviated AC). Finally, instead of a Privileges Required (PR) metric, CVSS v2 had an Authentication (AU) metric which recorded the minimum number of times an attacker had to provide credentials to an application when exploiting the vulnerability. The Base score CVSS v2 also lacked the Scope (S) subcomponent [57]. The CVSS v2 Base Exploitability metric is also calculated $20 \times AV \times AC \times AU$, i.e. using a slightly lower weight than CVSS v3 (2 vs 8.22)

In the Temporal metric group, the name of the metric referred to as “Exploitability” in CVSS v2 was updated to “Exploit Code Maturity” (E) to more accurately reflect what the metric evaluates [57]. Additionally, in the Environmental Metric, CVSS v2 had two distinct submetrics: Collateral Damage Potential (CDP) and Target Distribution (TD) which assessed impacts of exploiting a vulnerability. The CVSS v2 Environmental Metric did not include the Modified Base scores. To compute the CVSS v2 Temporal score, the CDP and TD metrics are combined with a weighted impact score in which the CR, IR, and AR Environmental metrics are used to weight the C, I, and A metrics from the Base score group instead of MC, MI, and MA.

4.3 Methodology

We followed a two-phase process for collecting studies, as shown in Figure 4.1 involving both a *Keyword Search using Active (Machine) Learning* and *Snowballing*, based on the SYMBALS methodology proposed by van Haastrecht et al [279]. The inclusion/exclusion criteria used in both phases of the review are discussed in Section 4.3.1. In the first phase, we use an active (machine) learning tool, FAST2 [305], to analyze the results returned from a keyword search, followed by further review and analysis by three researchers. In the original SYMBALS research by van Haastrecht et al [279], the authors found FAST2 consistently performed better than the alternative learning system, ASReview, in terms of recall (60-90% compared to 40-70%). We discuss the first phase further in Section 4.3.2. In the second phase discussed in Section 4.3.3, we performed Snowballing [279] to collect the papers cited and the papers cited by the studies returned in the first phase. Proponents of using active learning for literature surveys have found that active learning alone can produce precision and recall over 70% [305, 279], which is similar to the performance of human researchers. Combining active learning with Snowballing can produce precision and recall above 90% [279]. Once the papers were collected, we organized and categorized the papers as described in Section 4.3.4.

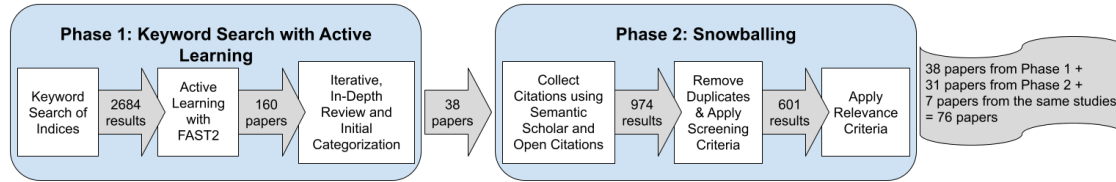


Figure 4.1: Paper Collection Process

4.3.1 Inclusion/Exclusion Criteria

Paper selection in both phases of the search leveraged the inclusion / exclusion criteria shown in Table 4.2. The **Screening** criteria are based on commonly used criteria for other surveys and literature reviews [150]. The **Relevance** criteria are based on the goal of our survey. The first column of Table 4.2 indicates which criteria are *Screening criteria* and which criteria are *Relevance criteria*.

In both Phase 1 and for our Backwards snowballing, we only use the criterion from Table 4.2. As can be seen in Table 4.2, we did not limit our search based on venue or year in our primary set of inclusion / exclusion criteria.

As we note in Section 4.3.3 we performed additional Forwards Snowballing to identify publications from major venues that were published in 2021 and 2022, since Backwards Snowballing provided no new papers from 2021 or 2022. Hence two additional criteria were applied for Forward Snowballing: year, (2021 - 2022), and venue quality. For conferences, we determined venue quality based on the GII-GRIN-SCIE (GGS) [107] and Computing Research and Education Association of Australasia (CORE) [290] Conference Ranking Systems. If a conference was considered tier 1 or tier 2 in GGS *and* had A- or higher in CORE, we consider it a “Major” Venue. Neither the GGS and CORE systems no longer maintain rankings for journals [107]. Hence for journals we rely on Impact Factor reported by Journal Citation Reports (JCR) [44] and H-Index reported by SCImago [253] for journals. We consider journals with Impact-Factor *and* H-Index scores in the top quartile for Computer Science to be “Major” venues. We also filter venues by discipline, as reported by CORE and GGS for conferences, and as reported in JCR and SCImago for journals.

4.3.2 Phase 1: Keyword Search with Active Learning

We used a keyword search to gather results from four database indices shown in Table 4.3: ACM Digital Library, IEEE Xplore, Compendex, and Web of Science. We selected these indices

Table 4.2: Inclusion/Exclusion Criteria

	Inclusion Criteria Include papers if:	Exclusion Criteria Exclude papers if
Screening	The paper is available	We are unable to obtain a copy of the paper
	The entire paper is available in English	The main body of the paper is in a language other than English
	The paper is published via peer-review	The paper is not selected via peer-review (e.g. Technical Report)
		The paper has been retracted
The paper is full-length	The paper was not full-length (e.g. short paper, poster)	
Relevance	The paper proposes or evaluates exploitability assessments for individual software vulnerabilities	All exploitability assessments proposed or evaluated in the paper assess exploitability of systems as a whole rather than individual vulnerabilities
		The paper focused network vulnerabilities or system vulnerabilities, such as configuration vulnerabilities, which are not software-specific
		The assessment is <i>not</i> security-related
	The paper discusses exploit generation as a software vulnerability exploitability assessment technique	If the paper proposes or evaluates a method for computing CVSS-based scores, (exclude the paper if) the paper only computes or evaluates an overall severity score; i.e. exploitability-related components, such as the Base Exploitability score, are not computed or evaluated.

because they are widely used and include major relevant publication venues. We do not include the widely-used Google Scholar because the results can only be downloaded via web crawler, which was disallowed by the google scholar robots.txt file² at the time of the search³. The query was constructed based on the research goal. For each index, the query required the terms “exploitability”, “vulnerability”, and “software”. Since the term “assessing” has many synonyms and forms, we designed the query to include at least one of the following terms: “metric”, “measure”, “measurement”, “assess”, or “assessment”. Table 4.3 provides the database-specific syntax of this query.

Our original search returned 2684 results. Three researchers used the FAST2 system [305] to perform an initial, independent screening of the titles and abstracts of the 2684 results, using the inclusion/exclusion criterion described in Table 4.2. FAST2 leverages machine learning to reduce the overall workload in the initial screening process. We applied FAST2 following the guidelines described by Yu et al [305]. The FAST2 tool includes a graphical user interface that

²A robots.txt file indicates which pages the website maintainers intend to allow a crawler to visit [152]

³<https://scholar.google.com/robots.txt>

Table 4.3: Databases Examined

Name <small>(URL)</small>	Description	Query Syntax
ACM Digital Library <small>(dl.acm.org)</small>	database & index from the Association for Computing Machinery (ACM)	<i>[All: exploitability] AND [All: vulnerability] AND [All: software] AND [[All: metric] OR [All: measure] OR [All: measurement] OR [All: assess] OR [All: assessment]]</i>
IEEE Xplore <small>(ieeexplore.ieee.org)</small>	database & index from the Institute of Electrical and Electronics Engineers (IEEE)	<i>("All Metadata":exploitability) AND ("All Metadata":vulnerability) AND ("All Metadata":software) AND ("All Metadata":metric" OR "All Metadata":measure" OR "All Metadata":measurement" OR "All Metadata":assess" OR "All Metadata":assessment")</i>
Compendex <small>(www.engineeringvillage.com)</small>	engineering database & index from Elsevier	<i>exploitability AND vulnerability AND software AND (metric OR measure OR measurement OR assess OR assessment)</i>
Web of Science <small>(www.webofscience.com)</small>	multidisciplinary reference database & index	<i>(ALL=(exploitability)) AND (ALL=(vulnerability)) AND (ALL=(software)) AND ((ALL=(metric) OR ALL=(measure) OR ALL=(measurement) OR ALL=(assess) OR ALL=(assessment))</i>

presents a list of 10 papers to classify as relevant or irrelevant. After the user has classified 10 papers, another 10 papers are selected by the tool. Initially, these papers are chosen randomly. Over time, the papers are presented based on their classification by an underlying Support Vector Machine (SVM) machine learning algorithm [306]. The FAST2 tool includes an estimated recall for the number of papers included at each step of the review. All three of the reviewers reached the 90% estimated recall target recommended by the authors of FAST2. Any paper selected for inclusion in the output from at least one researcher’s application of FAST2 using the inclusion/exclusion criteria from Table 4.2 was included at the end of the initial screening. The initial screening reduced the list of papers from 2684 to 160.

The 160 papers were then read in detail by researchers, and 122 papers were removed due to not meeting our inclusion/exclusion criterion in Table 4.2, such as due to being a short paper; or where the work was not articulated clearly enough to be accurately categorized. Ultimately, 38 papers were included at the end of Phase 1.

4.3.3 Phase 2: Snowballing

Once we had extracted our initial set of 38 papers, we performed snowballing by identifying papers that were cited by the initial set of papers (backward snowballing), as specified in the SYMBALS methodology [279]. Backward snowballing found no new papers from 2021 and 2022, the last two years of our search. We therefore also looked for papers that cited papers from our initial set (forward snowballing), and which were published at major computer science venues in 2021 and 2022, as described in Section 4.3.1. For both backward and forward snowballing we used the Semantic Scholar [255] and OpenCitations [222] APIs to collect papers. We collected 974 entries from the APIs. One researcher then analyzed the list of results to remove duplicate entries. The researcher also applied the *Screening* criteria from Table 4.2 and discussed in Section 4.3.1. After removing duplicates and applying the Screening criteria, 601 papers remained.

Of the 601 papers, 3 were extensions of work included in the previous phase of the survey, and therefore were assumed to meet the criteria from Table 4.2 and automatically included. Two researchers reviewed the other 598 papers using the *Relevance* criteria from Table 4.2. The first researcher performed two iterations of review. The second researcher then performed an initial analysis of 100, and the two researchers compared their results. Based on this discussion, the second researcher reviewed the remaining 498 papers in two iterations, with a brief discussion between iterations to clarify high-level trends (minimizing discussion of individual papers). For the final set of categorizations, the observed agreement was 95.2% with Cohen's Kappa 0.54 (95% Confidence Interval for the Kappa: ± 0.18), indicating moderate agreement [156, 84] even when considering agreement due to chance. The two researchers then discussed and resolved the remaining disagreements, resulting in the decision to include 28 papers. Ultimately, 31 papers were added in the second phase, including the 3 papers which were extensions of work included in Phase 1.

4.3.4 Organization and Categorization

Once we had identified the set of papers, we grouped together papers that were part of the same study and should be considered together in a practical application of the assessment. We consider papers to be part of the same study if the papers (1) were from the same authors; and include some of the same analysis; or (2) if later papers were explicitly building on the prior model for exploitability, such as the work by Alhuzali et al. ([7, 8]). To ensure we could correctly categorize papers, we reviewed the studies to determine if there was prior work that was necessary to understand the assessment technique proposed and should be included in the study, but where the original paper would not have been classified as “exploitability”

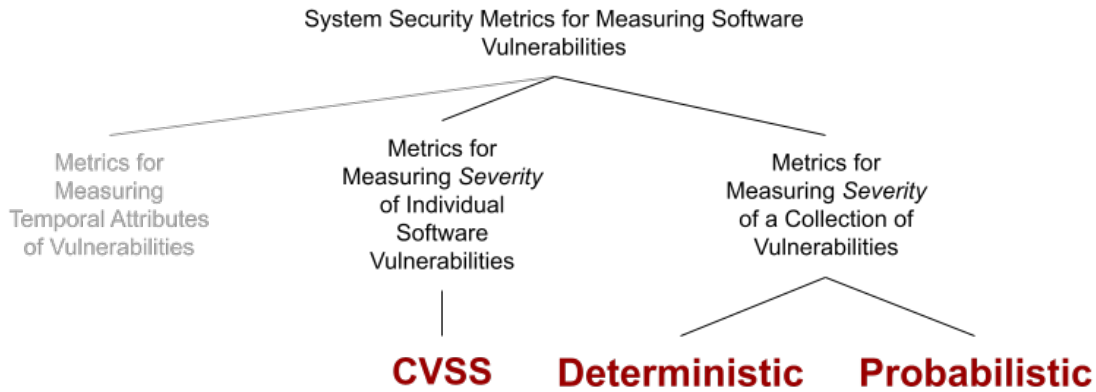


Figure 4.2: Graph Based on Metric Taxonomy in Pendleton et al. [221], with relevant attributes highlighted

related. For example, earlier work in S21 did not refer to the tool in terms of “exploitability”. The search for additional papers in each study added 7 papers to the survey. Our survey ultimately includes 76 papers, 72 of which are grouped into 59 studies, while four papers were proposing or presenting industry standards CVSS and EPSS, written by the authors of the standards.

After the initial Keyword Search, an initial classification of papers was performed by two researchers and iterated on further by the first author. We used an iterative, keywording approach [225] to determine the categories and characteristics useful to compare and contrast the different vulnerability assessment methods. These categories were further iterated on and expanded during and after Phase 2.

Our keywords aligned closely with three categories for “Metrics for Measuring Severity” of software vulnerabilities from Pendleton et al [221]: CVSS-based, Deterministic, and Probabilistic Assessment systems. The Vulnerability-Related subset of the taxonomy from Pendleton et al. [221] is shown in Figure 4.2, with the three categories we use in our survey highlighted in Red.

As can be seen in Figure 4.2, there are aspects of their taxonomy which differ from our work since the original classification applied to *system security metrics* and our survey focuses on *exploitability assessments of software vulnerabilities*. First, we focus on *exploitability*, a subset of severity. Hence we focus on the two “severity” subcategories and do not use a separate “temporal” category or its subcategories, as indicated by the grey color in Figure 4.2. Second, there deterministic methods for assessing the exploitability of vulnerabilities which are not based on a *collection of vulnerabilities*, unlike in the system metrics examined by Pendleton et al. [221]. Hence we do not use the second tier of the taxonomy in Figure 4.2. Finally, we define our categories based on the assessment method rather than the metric or score produced.

For example, Pendleton et al. define deterministic metrics as metrics which “assume that vulnerabilities can certainly be exploited” [221] while we define deterministic assessment methods based on process - assessments which will always produce the same output, given a particular input [195]. We provide definitions of each category in Section 4.4.

4.4 Categories of Assessment Methods in Each Study

As discussed in Section 4.3.4, the high-level categories of exploitability assessment methods used in this survey were based on categories from Pendleton et al. [221]. We classified the methods examined into three categories: CVSS-Based, Deterministic, and Probabilistic methods. We then further subdivide Probabilistic methods into methods based on Learning Models (LM), such as Support Vector Machines and Neural Networks; and Other (O) Probabilistic Models.

In the context of our survey, assessment methods classified in the “CVSS-Based” high-level category are based on the original manual/expertise-based assessment described in the CVSS Specification [60] for determining the low-level CVSS submetrics (e.g. AV, AC, RP, and UI). The only assessments we identified in the literature that use a similar, manual/expertise-based approach rely on the CVSS assessment. Assessments based on the original CVSS specification are not purely “Deterministic” since different experts may evaluate the score slightly differently, nor is it “Probabilistic”. Where Deterministic or Probabilistic methods are used to compute the CVSS submetrics, we classify the method as Deterministic or Probabilistic, accordingly. We discuss CVSS-based assessments in Section 4.5.

Deterministic Assessments are assessments that will always provide the same output for a particular input [195]. We discuss deterministic assessments in Section 4.6.

Probabilistic Assessments are assessment methods that rely on a statistical, probabilistic analysis of a set of vulnerabilities. In our study, we further subdivide Probabilistic methods into methods based on Learning Models (LM), such as Support Vector Machines and Neural Networks, and Other (O). Probabilistic Models. LM are discussed in Section 4.7, while Other Probabilistic models are covered in Section 4.8.

Our final set of 59 studies and 2 industry standards encompassing 76 papers, categorized based on assessment methods, are shown in Table 4.4. Table 4.4 gives an ID for each study in the first column. Academic studies begin with an “S”, while the two industry standards are referred to by their acronyms: Common Vulnerability Scoring System (CVSS) and Exploit Prediction Scoring System (EPSS). The second column indicates the bibliography entries for the paper(s) that were part of the study. The third column indicates the year(s) the papers were published. The remaining columns indicate the type(s) of exploitability assessments discussed in each study. A study, or even an individual publication, may cover multiple assessment methods. For

example, CVSS scores from the U.S. National Vulnerability Database (NVD) are frequently used as a baseline in evaluations of techniques in other categories, such as the learning model in S03 ([33]). To distinguish between when an assessment is the main focus of a study or when it is being used for comparison, we use “M” to indicate that the assessment method is the main focus of the study and “C” to indicate if a method is primarily used for comparison.

4.5 CVSS and CVSS-based metrics

The CVSS SIG provides a User Guide including decision trees for each of the CVSS Base Score components (AV, AC, PI, UI, C, I, A, & S) [60]. CVSS does not provide a way to automate assessment of CVSS scoring. CVSS, as specified, is, therefore, an expertise-based assessment of vulnerabilities [33, 12]. As we will see in Sections 4.6 and 4.7, rule-based and machine-learning models have been proposed to automate the CVSS scoring process [315, 113, 161]. However, these automated, i.e. not expertise-based, assessments are not in the original CVSS specification.

The NVD [197] provides a CVSS Base Score for all vulnerabilities in the CVE list [55], and there are few vulnerability datasets available to researchers that are not connected to the CVE list [34]. The NVD-provided Base Score includes both the overall score, and the calculation for each element of the Base Score such as AV, AC, PR, UI, and the overall Exploitability subscore [197]. Similarly, maintainers of proprietary datasets such as IBM X-Force may provide their own calculation of CVSS scores for vulnerabilities [270]. These assessments are at least partly expertise-based, following the original CVSS specification.

We identified three groups of studies examining CVSS Scores. The first group, which we discuss in Section 4.5.1, focuses on understanding *How CVSS scores are assessed* via user studies of the expertise-based CVSS assessment method. In Section 4.5.2, we discuss evaluations and criticisms of CVSS scores output by the CVSS assessment process, which primarily focus on scores from the NVD but, in some cases examine other sources such as IBM X-Force [144, 270]. Finally, in Section 4.5.3 we discuss changes that have been proposed based on some of the criticisms discussed in Section 4.5.2

4.5.1 How CVSS is calculated (manual/expertise-based)

In S34, Allodi et al. [11] evaluate what information is useful for determining the CVSS Base score via a user study of CVSS v3. Using the CVSS scores provided in the CVSS v3 example document [58] as ground truth, the researchers provided study participants with a tutorial

Table 4.4: Exploitability Assessment Methods Proposed and/or Evaluated in Each Study

M - assessments from this category are the main focus of the study; C - assessments from this category are compared against the main category

ID	Bib.	Year(s)	CVSS	Det.	Prob. (LM)	Prob. (O)
S01	[284]	2008		M		
S02	[101]	2009				M
S03	[33]	2010	C		M	
S04	[166]	2011	M			
S05	[103]	2011	M			
S06	[24], [252], [40], [25]	2011, 2014		M		
S07	[123], [124], [125]	2012, 2018, 2019		M		
S08	[9], [10]	2012, 2014	M			
S09	[130, 131]	2012, 2014, 2017, 2020		M		
S10	[268]	2013	M			
S11	[129]	2013	M			
S12	[170]	2014	M			
S13	[304], [301]	2014, 2016		M		
S14	[296]	2015			M	
S15	[126]	2015	M			
S16	[102]	2015		M		
S17	[303]	2015	M			
S18	[300]	2016			M	
S19	[246]	2015			M	
S20	[80]	2015			M	
S21	[229],[230],[232], [132]	2015, 2018, 2020, 2021		M		
S22	[256]	2016				M
S23	[271]	2016	M			
S24	[239, 240]	2016, 2017				M
S25	[7], [8]	2016, 2018		M		
S26	[13],[14]	2017, 2018			M	
S27	[73]	2017		M		
S28	[297]	2017			M	
S29	[105]	2017		M		
S30	[273]	2017			M	
S31	[244]	2017		M		
S32	[34]	2017			M	
S33	[120]	2017		M		
S34	[11]	2018	M			
S35	[285], [286], [312]	2018, 2019, 2020		M		
S36	[292], [293], [41]	2018, 2019		M		
S37	[106]	2018		M		
S38	[267]	2018			M	
S39	[144]	2018	M			
S40	[16]	2019			M	
S41	[113]	2019			M	
S42	[161]	2019			M	
S43	[315]	2019	C	M		
S44	[12]	2020	M			
S45	[298]	2020			M	
S46	[69]	2020		M		
S47	[245]	2020	M			
S48	[81]	2020			M	
S49	[299]	2020		M		
S50	[311]	2020			M	
S51	[140]	2021			M	
S52	[179, 178]	2021, 2022			M	
S53	[163]	2021		M		
S54	[51]	2021		M		
S55	[270]	2022	C		M	
S56	[141]	2022	C	M		
S57	[309]	2022		M		
S58	[147]	2022		M		
S59	[291]	2022			M	
CVSS	[174, 59, 173, 250]	2006, 2009, 2019	M			
EPSS	[138, 139, 96]	2020, 2021, 2022			M	

on determining CVSS scores. The control group was then asked to determine CVSS scores based on the original vulnerability descriptions from NVD alone, while the treatment group was given vulnerability descriptions with additional information from the CVSS example document [58]. The authors examined four categories of information: information about the vulnerable *asset*, i.e. information on the type of system directly affected by the vulnerability; *attack* procedures that could be used against the vulnerability; *vulnerability type* information characterizing the technical root cause and result of exploiting the vulnerability; and *known threats*, i.e. whether there is evidence that malicious actors have exploited the vulnerability on production systems. The researchers assessed how the addition and removal of each category of information impacted the error rate of participants. They found that information on *assets* reduced the error rate for the A (Availability) metric of the Impact group, but *asset* information had little other effect. Information on the *attack* reduced error rates for the AV and AC metrics of the Exploitability group and the C (Confidentiality) metric of the Impact group. Information on *vulnerability type* was related to a reduced error rate for AC, UI and PR metrics in the Exploitability group. On the other hand, information about *known threats* was related to an increased error rate for AV and AC in the Exploitability group, as well as C and A in the Impact group. Individual differences in performance between participants were observed, e.g. information on *vulnerability type* reduced error rates more for individuals with more security expertise[11].

In S23 ([271]), the authors perform text mining on descriptions of CVEs in the NVD and analyze the correlation between terms and the values of the different CVSS v2 metrics (AV, AC, AU, C, I, and A) using Spearman's rho. Their results somewhat support the analysis in S34. In S23, the authors used stemming, or reducing terms to their root form such as using the root form "attack" for "attacking" and "attacks". In S23 the authors found that the term "attack" had a moderate positive correlation (0.30) with AV, a moderate negative correlation (-0.35) with AU, although the correlation with AC was not as strong (0.02). However, terms associated with a particular type of attack such as "cross-site-script" and "script" had a higher correlation with AC (0.42 and 0.36, respectively). Other terms that had a moderate or strong correlation with AV included "remote" which had a positive correlation (0.53), as well as "local" and "user" which had negative correlations (-0.70 and -0.49, respectively). Other terms which had a moderate correlation with AC included "html" and "web", which both had positive correlations (0.40 and 0.36, respectively). Additionally, the terms "authent" and "user" had moderate to high positive correlations with AU (0.65 and 0.43, respectively). Correlation is not causation, and so we cannot be sure based on S23 alone how much these terms contribute to an individual's ability to score the metric. However, combined with work such as S34 we can see a more full picture. Further research would be needed to understand whether the differences between S23 and

S34, such as the relationship of terms like “web” and “html” to the AC metric identified in S23, is due to differences between CVSS versions, due to the context in which those terms are being used in the vulnerability descriptions, or due to factors that contribute to CVSS assessment that have yet to be discovered.

In S44 Allodi et al. [12] further examine how the actual expertise of the individuals applying CVSS influences the results. The authors use CVSS v3 as the assessment system for their experiment. They divided the participants into three groups based on expertise. The first group was graduate students in computer science with no security expertise. The second group was graduate students in computer science who had taken security courses but did not have industry expertise, and the third group was security professionals. The assessment was based on 30 examples randomly selected from a set of 100 vulnerabilities used by the CVSS SIG when developing the standard [12]. The CVSS scores had, therefore, been determined previously by the CVSS SIG and could be used as “ground truth”. The authors found that computer science students who had taken security courses performed significantly better than students who had not taken security courses for the AC metric and the Impact metrics (C, I, and A), and had “borderline” better performance with the AV and PR metrics. Allodi et al. found no statistically significant difference between the two groups of students for the UI metric. The authors found security professionals were less likely to err on the UI metric compared with both groups of students, and on the AV metric, where there was a borderline difference between professionals and students who had taken security courses. However, the authors found no statistically significant difference in the performance of the students who had taken security courses and the security professionals across the AC, PR, C, I, and A metrics. Their results suggest that security knowledge is helpful in producing CVSS assessments. However, students with security experience were relatively competent, producing compatible scores with professionals for 4 of the 6 metrics.

In S11 ([129]), the authors propose combining multiple expert analyses of CVSS scores via the Delphi method. The Delphi method is a consensus-building technique frequently used in areas such as Systems engineering [247]. The authors of S11 illustrate the possibility of their approach through an illustrative example with 4 CVE. The authors also provide the questionnaire that would be used in their method.

4.5.2 Evaluations & Criticisms

We found three primary directions for analyses of CVSS scores provided by industry groups, particularly the NVD. The first set of analyses which we discuss in Section 4.5.2.1 looks at the *Reliability* of CVSS, a term used to refer to the now consistently a particular vulnerability is

scored using CVSS. The second group examines the relationship between CVSS and other exploit-related indicators, such as the presence of an exploit in Exploit DB. Finally, in Section 4.5.2.3, we discuss four studies that examine the distribution of CVSS scores in the NVD, e.g. the percentage of scores that are classified with “Low”, “Medium”, and “High” severity or exploitability.

4.5.2.1 Reliability of CVSS scores

At least two studies examining Exploitability-Specific elements of CVSS, S15 ([126]) and S39 ([144]), have looked at the reliability of the CVSS scores provided in the NVD. In S15, the focus of the reliability analysis is on the overall severity score, with additional analysis of information that should be added or removed as it relates to each of the subscores (AV, AC, and UI). We discuss the proposed changes to the underlying CVSS system in Section 4.5.3.2. However, the reliability analysis of S15 may be used to triangulate the analysis from S39, which examines exploitability-specific elements of CVSS in their reliability analysis. Both studies focus on CVSS v2. Overall, neither study was able to statistically disprove the reliability of the CVSS scores from the NVD.

In S15 ([126]), the authors perform a survey of 304 security experts from industry and academia to evaluate the overall reliability of the CVSS v2 severity scores in the NVD as well as to evaluate the subcomponents and structure of the CVSS framework. To understand the accuracy of the CVSS scores in the CVE list, each respondent was asked to provide a severity score for 10 vulnerabilities from the NVD. Of the 10 vulnerabilities presented to each respondent, 3 vulnerabilities were the same for all respondents to enable the authors to estimate consensus between experts, while 7 vulnerabilities were selected randomly from vulnerabilities in the CVE list. A total of 2131 unique vulnerabilities were assessed across the 304 experts. The experts were provided with the description of each vulnerability, as well as the Exploitability values for AV, AC, and AU (with a brief explanation of the attribute) and Impact values for C, I, and A. The survey did NOT include the equation for calculating the Exploitability score, Impact score, or Base Severity score, requesting instead that the experts provide their own value between 1 and 10. The authors note that 38% of the answers to the survey were different from the score provided by NVD and claim “This is certainly a higher figure than many users of the scoring system would be comfortable with” [126]. However, it is not clear if this discrepancy is specific to the CVSS scoring system, to the scores in the NVD, or to expertise-based systems generally. The authors found no statistically significant difference between the scores provided by experts and the overall severity score from the NVD.

In S39, the authors expand on the work in S15 by focusing specifically on CVSS scores from five databases: NVD, the proprietary IBM X-Force Exchange database, OSVDB, the Vulnerability

Notes database from the CERT group at Carnegie Mellon University (CERT-VN), and the alert database provided by Cisco for its products (Cisco). For scores from the OSVDB, which includes CVSS scores credited to a variety of sources including the NVD, the authors exclude scores credited to the NVD to reduce potential bias. The authors acknowledge that OSVDB, CERT-VN, and Cisco all acknowledge their scores may be influenced by information from other sources, such as the NVD. They point to the differences between the scores provided in each database as evidence of independent scoring. The authors use Bayesian analysis to develop a Ground Truth for each CVSS subcomponent (AV, AC, AU, C, I, and A) based on the CVSS scores from the 5 databases. Based on this Ground Truth, the authors found the NVD to be the most accurate across all metrics (93% on average) and for the Exploitability metrics specifically (AV: 99%, AC: 88%, AU: 99%). The authors noted that the greatest disagreement occurred in the Access Complexity (AC) exploitability score, particularly for lower AC scores, and that Exploitability sub-components generally had higher disagreement than Impact subcomponents.

4.5.2.2 Exploitability scores from the NVD compared to publicly available Exploit-based datasets

Statistical analyses of the CVSS scores from the NVD in relation to other exploit information in S08, S03, S17, and S55 have generally suggested that the CVSS exploitability score is not strongly connected with the existence of exploits in exploit databases or the likelihood of exploitation. However, the statistical analysis in S47 ([245]) suggests that the relationship between exploits in exploit databases and the individual CVSS components (AV, AC, AU, C, I, A) may be stronger, particularly when factors such as company maintaining the software (e.g. Microsoft) are controlled for.

In S08, Allodi and Massacci [9, 10] examine the relationship between CVSS v2 scores, whether a vulnerability is associated with an exploit in ExploitDB, whether a vulnerability is associated with an exploit from a commercial exploit kit, and whether a vulnerability is associated with an exploit signature in Symantec Intrusion-Detection and Anti-Malware products. The exploit signatures from Symantec (SYM) were used as the “ground truth”. The commercial exploit kit information in S08 includes automated (i.e. code script) exploits extracted from malicious websites. In S08, Allodi et al. found that the CVSS v2 scores in NVD showed little variability and that only a few of the possible values for the different subscores were actually used. The authors note that “The CVSS base score alone is a poor risk factor from a statistical perspective.” However, the authors also found that considering both CVSS scores and other factors, such as an exploit in ExploitDB or EKITS, the relationship with the SYM data, i.e. risk of exploitation, improved.

Similarly, in S03 ([33]), Bozorgi et al. use the CVSS “exploitability” scores from the NVD as

the control against which they compare their own AUTO-LM system, which we will discuss in Section 4.7. The labels for the AUTO-LM system were the exploit availability labels from OSVDB [33]. The authors compare the distribution of the CVSS “Exploitability” provided by the NVD against the signed distance to the maximum margin hyperplane separating positive and negative examples in their SVM model, i.e. their LM. The authors illustrate with histograms how their score produces a clearer distinction between vulnerabilities that the OSVDB indicates have an exploit, compared to vulnerabilities that do not have an exploit.

Converting CVSS scores from the NVD into a binary exploitability score has also yielded low precision and recall in relation to the existence of exploits and exploit signatures from public databases. In S17 ([303]), Younis and Malaiya compare CVSS v2 against the Microsoft Rating System (MSRS) using exploits from ExploitDB as ground truth. The Microsoft Rating System was a predecessor to the current Microsoft Exploitability Index (MSEI) [176] and Microsoft Severity score [177], which are provided by Microsoft when disclosing vulnerabilities in their products to help users prioritize security patches. In S17, the authors use the median CVSS score for vulnerabilities as the cutoff for the confusion matrix of exploitable and not-exploitable vulnerabilities. Similarly, for the MSRS, the authors used the median value of “1” as the threshold for whether a vulnerability was exploitable or not. In other words, vulnerabilities with a MSRS of 1 were considered exploitable while vulnerabilities with a rating of 2 or 3 were considered not to be exploitable for the purposes of the evaluation. Using this approach, the authors determined that CVSS had a precision of 7% and recall of 97% for Internet Explorer (IE), and a precision of 20 % and recall of 65 % for Windows 7. The authors found that MSRS performed similarly. For IE, this threshold of the MSRS resulted in a precision of 7% and a recall of 85%. For Windows 7, the threshold for the MSRS resulted in a precision of 15% and a recall of 83%. The authors argue that the low precision and recall indicate that CVSS and MSRS are not good indicators of exploitability, and new metrics are needed.

Similarly, a preliminary comparison of CVSS v3 with Proprietary measures in S55 ([270]) examined the precision and recall of the Base Exploitability score, setting the threshold at each possible value from 0 to 3. That is, they examined the precision and recall if a vulnerability with a Base Exploitability score of “0” or higher was considered “exploitable”, then they examined the precision and recall if a vulnerability had a Base Exploitability score “1” or higher, etc. For the ground truth, the authors use a combined dataset of exploit signatures from Symantec products; information extracted from Bugtraq, Tenable, Skybox, and AlienVault OTX vulnerability databases; and exploits extracted from the Contagio dataset, a publicly-available list of exploit kits and malicious websites used in academic studies [9, 10, 154, 313, 159, 224]. Using a Base Exploitability threshold of 0 or 1 had approximately 85% recall⁴, while using a threshold

⁴Values are estimated based on a chart from the paper. Numeric results were not provided outside the bar

of 3 resulted in less than 20% recall. Precision values were below 20% for all thresholds. The authors of S17 and S55 [270] both use their evaluation of CVSS to argue that better exploitability measures are needed.

In contrast with the analysis examining the CVSS Base Exploitability score as a whole, statistical analysis by Roumani and Nwankpa in S47 ([245]) found that the CVSS v2 exploitability (AV, AC, AU) and Impact (C, I, A) subcomponents from the base score all had statistically significant relationships with the “hazard” that an exploit would be made available in exploit DB. The authors controlled for the affected product type, number of affected software versions, number of past exploits, year of disclosure, size of the software vendor, and R&D budget of the software vendor. This suggests that the relationship between CVSS Exploitability-related components and exploit availability may be more nuanced, and requires further investigation.

4.5.2.3 Distribution of CVSS scores in the NVD

Another common critique of CVSS relates to the overall distribution of exploitability and severity values of the CVSS scores provided by the NVD. S04 ([166]), S10 ([268]), and S12 all critique CVSS scores in the NVD for being disproportionately “High”, which they attribute to problems with the CVSS calculation method. We discuss their proposed changes to the CVSS calculation method further in Section 4.5.3.1. However, in this section, we examine their criticisms and compare their analysis with the work by Gallon in S05, which is less critical of the system and found a different distributional imbalance.

In S04, The authors argue that “In our opinion, the number of vulnerabilities with ‘Medium’ severity ranking should be the largest and the number of vulnerabilities with ‘High’ or ‘Low’ severity ranking [should be] much smaller.” [166]. In S10, the authors argue that severity scores should have a more diverse range of values, and should be more evenly distributed [268]. The authors of S12 also call for increased diversity, stating “The CVSS empirical values given by CVSS-SIG cannot distinguish software vulnerabilities that have identical scores but different severities.” [170]. In S12, Luo et al. also point to correlations between the sub-metrics of the Base CVSS v2 Score (AV, AC, AU, C, I, and A) from the NVD, as determined by a chi-squared test, as an indicator that vulnerabilities are not being scored independently.

The vulnerabilities examined in S04, S10, and S12 contain considerable overlap and have similar distributions. In S04, the authors evaluate 34,093 CVE vulnerabilities published from 1999 to 2008, excluding vulnerabilities with “incomplete” information of which 6.8% had a “Low” CVSS severity, 47.8% had a “Medium” CVSS severity, and 45.5% had a “High” CVSS severity. In S10, the authors evaluate CVSS scores from 9455 vulnerabilities in the NVD published

chart. The preliminary evaluation served as motivation for the main model proposed in the paper, which we discuss in Section 4.7.

between November 1 2010 and October 31 2012, where the vulnerability report had sufficient information. In S10 7.9% of the vulnerabilities had “Low” severity, 53.2% of vulnerabilities had a “Medium” severity, and 38.0% of the vulnerabilities had “High” severity. Additionally, in S10 the authors analyze the distribution of all components of the CVSS score including AV, AC, and AU, showing that over 80% of vulnerabilities in their sample of 9455 vulnerabilities from the NVD had an AV of “Network” (the highest value) and required no authentication (AU). However, AC was more evenly distributed. In S12, the authors examine 54,432 vulnerabilities which “encompasses all valid CVE entries published between 2002 and 2012” [170] for their evaluation of the independence of the sub-metrics (AV, AC, AU, C, I, and A).

The approach in S04, S10, and S12 contrasts with the approach taken by Gallon in S05. The authors examine 40,026 vulnerabilities in the NVD published between 1999 and 2009. The authors found a distribution with 45% of vulnerabilities having “Low” severity, 46% of vulnerabilities having “Medium” severity, and only 9% of vulnerabilities having “High” severity. However, the authors similarly found that the diversity in the combinations of subcomponent values, particularly the Impact subcomponents (C, I, A) was relatively low. Unlike in the other studies, the authors of S05 do not inherently consider this a defect of the scores in the NVD or of the CVSS scoring system itself. The authors then examine how the Environmental Impact metrics included in the CVSS framework may alter the scores, such as to improve diversity, finding that the Environmental Impact scores are more likely to decrease the overall severity score.

4.5.3 Proposed Changes/Improvements

Based on analyses and criticisms discussed in Section 4.5.2, at least four authors suggest potential changes that could be made to the NVD. We divide the proposed changes into two groups. First, in Section 4.5.3.1, we discuss two studies, S04 ([166]) and S10 ([268]), which propose altering the equations used to calculate CVSS, including the equation used to calculate the Base Exploitability score, such as by altering how each component (e.g. AV, AC) is weighted. They do not propose altering the underlying components. In Section 4.5.3.2 we discuss two studies, 10 ([170]) and S15 ([126]), which propose adding or altering the components themselves. In S10, S04, and S12, which are based on criticisms of the distribution of CVSS scores in the NVD as discussed in Section 4.5.2.3, the authors evaluate their changes by illustrating how they produce a distribution of severity scores different from the distribution of scores in the NVD. In S15, ([126]), the proposed changes are based on survey responses. They are used to suggest that the existing categories of CVSS are not appropriate from a theoretical perspective, although the authors do not fully specify an alternative in their study.

4.5.3.1 Equation Changes

As discussed in Section 4.5.2.3, S04 ([166]), S10 ([268]), and S12 all critique CVSS scores in the NVD for being disproportionately “High”. S04 and S10 suggest alternative ways of weighting and calculating CVSS Base scores and subscores, including the Exploitability Score. However, they do not indicate that the underlying determination of AV, AC, and AU should be performed any differently from the original CVSS score. All of these studies look at CVSS v2, where the Base Exploitability Score was calculated as $20 \times AV \times AC \times AU$ as discussed in Section 4.2.2.

In S04 ([166]), the authors make more significant changes to the impact score, but only change the exploitability score by a factor of 10 (to $2 \times AV \times AC \times AU$), so that the Base Exploitability score has a range of 0-1 instead of 0-10. The authors evaluate their overall proposed score against the overall CVSS severity score using AV, AC, and AU scores provided by the NVD, finding that their changes to produce a greater percentage of “medium” severity vulnerabilities, which the authors argue is preferable in a severity score, when compared to the original CVSS calculation.

In S10 ([268]), the authors change the exploitability score to $6 \times AV \times AC \times AU$. The combined changes to the exploitability, impact, and overall severity scores proposed in S10 result in a more even distribution of the severity score across the “low”, “medium”, and “high” levels, compared to CVSS.

4.5.3.2 Component Changes

Based on their analysis of the distribution of CVSS scores discussed in Section 4.5.2.3 S12 ([170]), Luo et al. propose a metric calculated based on the constituent CVSS subscores (AV, AC, AU, C, I, A) from the NVD, combined with temporal factors, such as the time since the vulnerability was disclosed. Unlike CVSS scores and the metrics in S04 and S10, which can be independently calculated for each CVSS score, the metric in S12 is calculated relative to other vulnerabilities within the same set, such as relative to all other vulnerabilities in the NVD. The authors demonstrate that their metric produces a different distribution of values than CVSS when applied to 54,432 vulnerabilities from the NVD. Luo et al. were not alone in their concerns, and many of the changes now incorporated into CVSS v3 were due to critiques similar to Luo et al.’s. For example, the Scope variable was added to capture whether exploiting the vulnerability could have impacts outside the vulnerable component [57].

As discussed in Section 4.5.2.1 In S15 ([126]), the authors perform a survey of 304 security experts from industry and academia to (1) evaluate the overall accuracy of the CVSS severity scores in the NVD and (2) Whether the underlying components of the Base Score in CVSS v2 (AV, AC, AU, C, I, and A) “are appropriate from a theoretical perspective.” [126]. To understand

the “appropriateness” of the components of the Base Score in CVSS v2 (AV, AC, AU, C, I, and A), the authors asked open-ended questions to the survey respondents about whether any additional components were needed and whether the existing components required revision. The authors identified 8 categories of suggested additions and revisions: *Prevalence of the vulnerable application*, *Cost of Impact*, *Availability of an exploit*, *Possibility of automization*, *the Availability of a patch*, *Availability of an exploit*, *Availability of detection system signatures*, *Effectiveness of detection system signatures*, and the use of *Vulnerabilities in combination* with each other. Interestingly, of the 38 respondents who discussed “Cost of Impact”, 8 respondents indicated that “Cost of Impact” should be considered as part of or replacing the “AU” metric while one respondent indicated “Cost of Impact” should be considered as part of or replacing the “AC” metric, even though both AV and AC are Exploitability metrics. The authors note that most of these metrics are Environmental metrics some of which, such as Availability of an Exploit or Cost of Impact, are explicitly covered in the existing Temporal and Environmental metrics of CVSS.

4.6 Deterministic

We first the automated exploitability assessment tools based on whether the determination of vulnerability exploitability is primarily a rule-based process, or if the determination is performed by a Learning Model (LM). 22 out of 23 of the studies on Deterministic, Automated exploitability assessment tools in this survey base the assessment on information gathered from program analysis, which we discuss in Section 4.6.1. A similar approach using network analysis is discussed in Section 4.6.2. However, the key underlying component of the network-based metric, a graph of the system state machine, is equally applicable to the analysis of vulnerabilities in other contexts.

4.6.1 Program-State-Based

In this section, we examine Deterministic tools that examine how a sequence of program states may lead to an “exploitability property” [25] being met. The exploitability property is determined by the intended output, such as the type of exploit to be produced or the sub-components of the CVSS score. The exploitability property is then defined in terms of the types of program analysis performed by the tool, such as static binary analysis or memory monitoring. For example, the exploitability properties in S06 ([24], [252], [40], [25]) are defined in terms of elements of the internal program state space such as functions and register calls, e.g. “*the IP register holds a value that corresponds to some function f of user input i (such as, f may*

be a call to tolower on the input i) and the resulting IP points to shellcode”[25]. The authors use static analysis and binary instrumentation to extract assembly language functions from the program executable (also referred to as a “binary”) [24, 40] to gather information about the program state space that may be used by a Satisfiability Modulo Theory (SMT) Solver [25] to determine if there is an execution trace in the source or assembly code that results in the exploitability property being met. The tool uses the SMT result to produce an exploit. In S43, the properties are determined based on CVSS, and defined in terms of the outputs of instrumented binary analysis. For example, the AV (access or attack vector) property was determined based on the existence and observed behavior of function calls such as `socket` and `connect` when a *dynamic trigger* for the vulnerability is run against the target application.

Table 4.5 shows several characteristics of the tools implemented in studies of automated, rule-based exploitability assessments. The first two columns of Table 4.5 indicate the study ID and associated bibliography entries. The third column indicates the years in which the studies were published. The fourth and fifth columns pertain to the tool outputs, which we will discuss in Section 4.6.1.1, including the *type of exploit* that the tool can generate, if applicable. Columns 6-9 show the primary inputs to the exploit generation tools, which we discuss in Section 4.6.1.2. The tenth column is the *type of vulnerability* the tool is designed to analyze. The eleventh column is the language(s) of the vulnerable applications that the tools can be applied to. Column 11 indicates the types of programs (applications) that the tool is intended to be used for. Column 12 indicates the size of the dataset used in the evaluation of the tool in terms of the number of vulnerabilities. Column 13 indicates the amount of time to run the tool per vuln (unless otherwise noted). The rows of Table 4.5 are ordered based on their outputs and then on the year(s) in which the work was published since, as we will discuss in Section 4.6.1.1, the tool output is a key distinguishing factor between studies.

4.6.1.1 Outputs

Within the studies examining Deterministic tools based on program-state-space, 3 studies (S43, S33, and S13) produce an output indicating whether a particular exploitability property is met, such as whether the PoC input provided to the tool in S43 triggers specific functions associated with lower or higher CVSS v3 AV, AC, PR, and UI scores. E.g. statements using the phrase `chmod`, the name for the utility for changing privileges in Linux-based OS[122], is associated with higher PR (Privileges Required) scores. The other 17 (S06, S07, S09, S31, S35, S36, S36, S46, S49, S56, S57, S53, S15, S21, S58, S25, S29, and S54) studies examine tools which produce, as output of the exploitability analysis, an input to the system under test (SUT) that can be used to trigger the exploitability property. These are sometimes referred to as “Exploit Generation” (EG) tools [25]. The types of exploits developed by the tools in our survey are listed under “Exploit Type” (the

Table 4.5: Studies on Automated, Rule-Based Program Analysis for Assessing Exploitability “NA” = “Not Applicable”; “NP” = “Not Provided”

ID	Bib.	Year	Output	(Output) Exploit Type	Inputs				Vuln. Type(s)	Lang.	Eval.	
					source code	exec. binary	vuln. dynamic	Inst.			Trigger	Target Software
S43	[315]	2019	CVSS Scores	NA	X	X	X	Any/ Unspecified	C/C++	OS (Linux kernel), Services (Apache, FTP)	98	NP
S13	[304], [301]	2014, 2016	Exploit Info.	NA	X	X	X	Any/ Unspecified	C/C++	OS (Linux Kernel), Services (Apache)	111	NP
S33	[120]	2017	Exploit Info.	NA	X	X	X	Memory (heap-based)	C/C++	“real-world” programs	9	<2m avg. 5m max.
S06	[24], [252], [40], [25]	2011, 2012, 2014	Exploit	Control Flow Hijack ^a	X	X	X	Memory	C/C++	command-line programs	29	<1m to 3h 41m
S07	[123], [124], [125]	2012, 2018, 2019	Exploit	Control Flow Hijack	X	X	X	Memory	C/C++	OS (Linux Kernel), language interpreters	10	27m to 53m
S09	[130, 131]	2012, 2014	Exploit	Control Flow Hijack	X	X	X	Memory	C/C++	command-line & user-level programs	33	<1m to 4h ^b
S31	[244]	2017	Exploit	Control Flow Hijack	X	X	X	Memory (heap-based)	C/C++	Windows services & libraries	8	10m to 20h 27m
S35	[285], [312]	2018, 2020	Exploit	Control Flow Hijack	X	X	X	Memory (heap-based)	C/C++	user-level programs	24	<1m to 22m
S36	[292], [293], [41]	2018, 2019	Exploit	Control Flow Hijack	X	X	X	Memory (heap-based)	C/C++	OS (Linux Kernel)	27	1m to 2h ^c
S37	[106]	2018	Exploit	Control Flow Hijack	X	X	X	Memory	binary	Web Browsers	5	<1m to 2m
S46	[69]	2020	Exploit	Control Flow Hijack ^a	X	X	X	Memory (heap-based)	binary	user-level programs	20	15m to 41m
S49	[299]	2020	Exploit	Control Flow Hijack ^a	X	X	X	Memory	C/C++	Language Virtual Machines	11	11h 57m to 13h 26m
S56	[141]	2022	Exploit	Control Flow Hijack	X	X	X	memory	C/C++	command-line programs	38	8hr ^d
S57	[309]	2022	Exploit	Control Flow Hijack	X	X	X	Memory (heap-based)	C/C++	OS (Linux Kernel)	17	5m ^d
S53	[163]	2021	Exploit	Triggering Race Condition (PoC)	X	X	X	concurrency bug	C/C++	OS (Linux kernel, Microsoft Windows)	10	<1m to 2m
S16	[102]	2015	Exploit	Input that is not sanitized (PoC)	X	X	X	sensitive stmt. (user-defined) ^a	Java	Android Apps	26	2m to 33m ^b
S21	[229], [230], [232], [132]	2015, 2018, 2020, 2021	Exploit	Unexpected Behavior (PoC)	X	X	X	Vuln. Third-Party Components	Java	Programs using Third-party Libraries	627	<1m to >1h
S58	[147]	2022	Exploit	Unexpected Behavior (PoC)	X	X	X	Vuln. Third-Party Components	Java	Programs using Third-party Libraries	42	<1m to 10m ^c
S25	[7], [8]	2016, 2018	Exploit	Injection, Exec. After Redirect	X	X	X	Input Validation, Business Logic	php	Web application	26	<1m to 2h 19m ^b
S29	[105]	2017	Exploit	DoS; Injection ^a	X	X	X	Null Ptr. Validation; Data Check ^a	Java	Android Apps	2092	<1m to 3h 27m ^b
S54	[51]	2021	Exploit	Injection, Path Manipulation	X	X	X	Input Validation, Hardcoded Key, Dangerous Func., Open Redirect, Info. Disclosure	php	Web Applications	403	NP

^aStudies S06, S29, S46, S49, and S16 include a template-like mechanism to analyze additional types of vulnerabilities and/or exploits

^bIn S09, S16, S25, S29 the Time to Run is reported “per application” rather than “per vulnerability”

^cS36 and S58 stopped their tool at the maximum timestamp, even if the tool had not achieved its goal.

^dIn S56 and S57, the tool runtime was capped at 8hr and 5m, respectively, since the tools could, theoretically, run indefinitely

^eIn S06, the maximum range value at over 3hr was an outlier. The vulnerability with the second-highest time-to-run only required 16m.

fourth column) of Table 4.5. The horizontal line in Table 4.5 differentiates between studies where the main focus is on tools whose output determines *whether a particular exploitability property is met* and EG tools.

The work in S56 ([141]) highlights the relationship between tools that determine whether the exploitability property is met and EG tools. As a separate research question, the authors determine a vulnerability's severity score based on the capabilities of the exploits generated by the EG tool. The capabilities assessed to determine the severity score in S56 overlap with the properties examined to determine severity in S33 ([120]). For example, both studies use the number of bytes over-written by an exploit of a buffer-overflow read/write vulnerability as part of their assessment. Similarly, in S13 ([304], [301]), the authors determine vulnerability exploitability based on reachability analysis very similar to the exploit generation tools examined in S21 ([229],[230],[232], [132]) and S16 ([102]) which only provide a "PoC" exploit to determine the reachability of vulnerabilities in third-party libraries. However, the tool in S13 does not provide the exploit.

4.6.1.2 Inputs

All of the automated, rule-based exploitability assessment tools require at least two inputs - an *instance of the application containing the vulnerability (Inst.)* and *information about the vulnerability itself (Vuln.)*. The *instance of the application* may be in the form of **source code**, or in the form of an **executable** (often referred to as a "**binary**") that has been compiled from the source code. While starting with source code may seem to provide more options since the source code may be turned into the executable form more easily than the executable form can be reverse-engineered, Hu et al. argue that starting with the executable is more advantageous since "real-world programs are ... often available in binary-only form". The early work of S06, as well as five other studies (S13, S58, S25, S29, and S54) start with the original source code for the *instance of the vulnerable application*, while later work in S06 and 8 studies (S07, S31, S35, S36, S37, S46, and S49) use a compiled (binary) executable form of the *instance of the vulnerable application*. S31 also requires a set of test cases for the entire target application such as a regression test suite [125] as part of their inputs.

Information about the vulnerability, on the other hand, is either the location of the vulnerable statement within the codebase (**vuln. stmt. loc.**) which is typically extracted through *static* analysis; or an application input which triggers unintended behavior that indicates a vulnerability, which may be referred to as a **dynamic trigger**. A dynamic trigger is typically produced via through *dynamic* analysis that does not have access to source code. As can be seen in Table 4.5, **vuln. stmt. loc** is typically used by tools that focus on the analysis of the original source code, while a **dynamic trigger** is more frequently used with binary analysis.

4.6.1.3 Vuln. Type(s) & Language

As can be seen in Table 4.5 program analysis for assessing memory vulnerabilities in C/C++ programs is an area of considerable prior research. Memory vulnerabilities, particularly heap vulnerabilities [124, 125, 309], require an analysis of a program in the context of the system within which it will be run. For example, the exploitability of heap vulnerabilities depends on the memory allocator in the system within which the vulnerable application is running. Consequently, most of the tools targeting the heap vulnerability involve dynamic analysis in which the program is running in a particular context. In contrast, tools examining other types of vulnerabilities in other languages tend to rely on Static analysis, including for taint tracing and other techniques, as shown in Table 4.5

4.6.1.4 Evaluation

Table 4.5 highlights three key aspects of each tool’s implementation and evaluation: the types of software targeted by each tool, the number of vulnerabilities (# vuln.) examined in the evaluation which we use as a common measure for the size of the dataset examined in each study; and the range of values for the Time to Run each tool, which is assessed per vulnerability unless otherwise noted. For studies that included more than one paper, the target software was the same or similar in all papers. In Table 4.5 we include the # vuln and Time to Run from the largest, most recent evaluation. We discuss Target Software in Section 4.6.1.4.1, # vuln and technique effectiveness in Section 4.6.1.4.2, and results for the efficiency or “Time to Run” in Section 4.6.1.4.3.

4.6.1.4.1. Target Software

The target software for the tool(s) used in the study is shown in the eleventh column of Table 4.5. Where the authors do not specify a class of software program the tool is intended to be used for, we generalized the “Target App” based on the systems analyzed as part of the empirical evaluation. For example, S35 and S46 were both evaluated against programs from Capture-the-Flag competitions. S35 refers to their targets simply as “programs”, while S46 refers to the CTF programs as “user-level applications”; we therefore refer to the target application as “user-level programs” in the “Target App.” (seventh) column of Table 4.5. As can be seen in Table 4.5, 9 of the studies focused on services, command-line, and/or user-level vulnerabilities. 5 focus on Operating Systems. Two studies examine language interpreters and language virtual machines. Two studies focus on Android Applications (apps). Two studies focus on client programs of third-party libraries. One study examined Web Browsers.

4.6.1.4.2. Number of Vulnerabilities (# vuln.) & Effectiveness

Column 12 of Table 4.5 shows the number of vulnerabilities used in the largest evaluation for each system. Many of these datasets are relatively small. The largest number of vulnerabilities analyzed, study S29 ([105]), were extracted from 835 Android apps tool using static analysis.

Of the 17 Deterministic, Program-State-Based studies where the exploitability of the vulnerabilities is known, at least 10 studies report no false negatives (S13, S33, S06, S07, S09, S31, S37, S49, S56, and S53), suggesting a recall of 100%. Even fewer studies in this category discuss much less report, False Positives for their exploitability assessment method. The smaller datasets and reduced focus on Confusion Matrix (True Positives, False Positives, True Negatives, and False Negatives) and metrics composed from the Confusion Matrix, contrast sharply with the other large group of exploitability assessment studies, Probabilistic LM studies, where a Confusion-Matrix-Based metrics such as precision and recall are the primary drivers of evaluations. Additionally, as noted in the “Lessons Learned” from Ponta et al. based on their work to transition the tools developed in S21 into practical use at the company SAP [232], tool Precision is a key factor. While exploitability analysis may be able to improve the precision of vulnerability detection tools [232], researchers should ensure the evaluation of exploitability assessments is as rigorous as possible.

4.6.1.4.3. Time to Run

The use of small datasets may be partly explained by the time it takes to run many of these tools, shown in the last column Table 4.5. In the “Time to Run” Column we include summary information based on the results provided in the largest evaluation for each. S57 and S58 set a maximum time limit for how long to run a tool. For the other studies, we provide either the range of the time to run or the average time to run if the range is not provided in the study. S43 and S13, do not include an evaluation of Time to Run. A lower “maximum” does not inherently mean that a tool is always faster. As indicated by the “Target App” and “# Vuln” columns, different tools were evaluated against different vulnerabilities in different systems. The simpler, CTF-based “user-level programs” used in S35 and S46 are likely to have a smaller state space compared with the Language Virtual Machines examined in S49. The discrepancy in performance between systems is examined in S09 ([130, 131]), where the authors found that their tool could produce exploits in less than 5 seconds on simple example code, but required 4 hours to run when applied to a larger piece of software, Foxit PDF Reader, containing over a million lines of code [131]. Similarly, in S06 the authors found that their tool required nearly 4 hours to run on one executable which was encoded or “packed” [40], likely causing an increase in analysis complexity. In comparison, the second-highest run time was only 16

minutes. However, as noted by Ponta et al. [232] in 21, based on their work at SAP, the amount of time required to run a tool influences when and how it can be used. While the vulnerability detection component of their tool only required 76s to run and could be included in frequent scans, the exploitability analysis often required hours to run and was more likely to be included in deeper scans run shortly before release, or in manual analysis performed after a release.

4.6.2 Network-System-State-Based

We identified one study, S01 ([284]) which predates the program analysis work, but leverages a similar, deterministic vulnerability exploitability metric as part of their network security analysis. They model the network as a graph of system states and measure the exploitability of a vulnerability as the in-degree of the vulnerability within the system state graph. Although this graph is based on a network and not a program, their graph of system states is very similar to the graphs used to analyze the system state of a program in other deterministic systems such as S16 ([102]) and S13 ([304, 301]). The assessment in S01 focuses on system-level metrics which their vulnerability-level metric contributes to, rather than an evaluation of the vulnerability exploitability metric.

4.6.3 Attacker-Based

We identified one unique study, S27 ([73]), in which “exploitability” was defined based on attacker characteristics and activity. In S27 ([73]), the authors propose a metric **Threat Agent Count (TAC)** which indicates the *quantity of threat actors capable of exploiting* a particular vulnerability. The authors evaluate their metric by comparing six vulnerability prioritization policies: (1) a first-in-first-out (FIFO) policy, (2) prioritize vulnerabilities with the highest CVSS score first, (3) prioritize vulnerabilities with the highest score from the adapted CVSS framework proposed in S04 ([166]) first, (4) prioritize fixing vulnerabilities with the highest TAC score first, (5) prioritize fixing vulnerabilities based on highest CVSS score, then highest TAC score in case of a tie, and (6) prioritizing fixing vulnerabilities based on highest TAC value, then highest CVSS value in the case of a tie. They evaluate and compare the policies based on how much the policy exposes a software system to vulnerabilities with a known exploit. Exposure is measured as a function of time t where E_t is the set of vulnerabilities with known exploits that have yet to be fixed, such that

$$Exposure(t) = \sum_{i=0}^t |E_i|$$

. The authors used 1,000 randomly selected vulnerabilities to represent the vulnerable Information System (IS), which were evaluated via simulations of 1,000 time-steps t . At each time-step,

1 vulnerability was fixed according to the policy being evaluated. The authors report their results at the end of each quarter of the simulation, i.e. every 250 t . The policy of prioritizing vulnerabilities based on TAC (policy 6), then by CVSS in the case of a tie had the lowest exposure at the end of each quarter, followed by prioritization entirely based on TAC, where in the case of ties the first-found vulnerability is removed first (policy 4). This leads the authors to claim that TAC “is a significantly better predictor of exploitable vulnerabilities than CVSS Score” [73].

4.7 Probabilistic Assessments: Learning Models (LM)

Another body of research builds models that take input from existing datasets and convert the data into a set of features, which are then analyzed using machine learning or neural networks to produce a value indicative of a high-level concept. We refer to these machine learning and neural network-based models collectively as Learning Models (LM). Table 4.6 summarizes the LM examined in the studies surveyed. The first two columns indicate the study ID and corresponding bibliography entry, as previously shown in Table 4.4. The next column indicates the Year(s) in which the studies were published. Columns 4-9 indicate aspects of the data used as the “ground truth” (GT), such as for training/testing of supervised learning models, which we discuss in Section 4.7.1. Columns 10-14 indicate types of data used as Features (FT) for the models, which we discuss in Section 4.7.2. The data sources for GT and FT will be covered in their respective sections, and are shown in Column 15. The number of vulnerabilities in the dataset used to train and test the model is shown in Column 16. Where more than one model is examined, we focus on the newest model that is proposed in each study. If it is unclear which model is the newest based on publication date, we use the largest. Finally, In the last column of Table 4.6 we list the Modeling Technique(s), such as Support Vector Machine (SVM) which are examined in each paper and may be useful for some readers. However, we leave a detailed analysis of the technical details of modeling techniques to surveys focused on the Learning Model techniques used in Security such as the works by Sotos et al [266], Kotenko et al [153], and Le et al [162] discussed in Section 4.1.

In Table 4.6, we separate the learning models into four groups based on GT and FT, as indicated by the solid lines, to highlight high-level trends in LM exploitability assessment research. These divisions are based on the following distinctions:

GT: Base CVSS “exploitability” vs Other Exploit Indicators There appears a clear distinction between models that target the subcomponents of the CVSS score, including exploitability-related subcomponents, and models that target other exploit-related information. Models in the ‘Base CVSS’ GT group exclusively use the subcomponents of the CVSS score as their

GT data, while models that focus on other exploitability-related indicators sometimes use a range of different sources.

FT: Program Analysis vs Metadata & NLP As can be seen in Table 4.6, studies tend to focus either on features gathered through Program Analysis, or on features gathered from other sources such as Vulnerability Reports and Social Media posts. Non-Program-Analysis features tend to be in the form of metadata such as the number of references or the Common Weakness Enumeration Scores (CWE) associated with a vulnerability, or extracted via Natural Language Processing (NLP) techniques.

As can be seen in the “Year” (3rd) column Table 4.6, the use of Program Analysis-based FT to predict CVSS scores has only become a popular research topic in recent years (since 2021). In comparison, the use of Program-Analysis-based FT to predict Exploits goes back to 2016, while the use of Vulnerability Reports and other FT for both CVSS and other exploit-related predictions goes back to 2015 and 2010, respectively. We discuss types of GT and FT in more detail in Sections 4.7.1 and 4.7.2.

4.7.1 What is used as the ground truth labels for Training / Testing (GT)?

All of the models in this section are supervised machine learning or neural network models, requiring training data that includes a set of features on which the model is based and labels indicating the ground truth for each instance, e.g. the exploitability for each vulnerability. Empirical studies of machine learning models typically perform their evaluation by withholding a subset of the labeled data to use in testing the model once it has been trained. The models can be split into two distinct groups based on their GT. The first group is models that use the base CVSS score directly or indirectly from NVD as the target of their model, which we discuss in Section 4.7.1.1. The second group uses one or more of a combination of indicators, including exploits from a database like ExploitDB, Exploit Signatures, specially dedicated exploit flags in vulnerability databases, and other exploit-related indicators. As shown in Table 4.6, authors can combine multiple sources as part of the same GT, hampering further orthogonal classification. We discuss this second group of Other Exploit-Related GT in Section 4.7.1.2

As we discuss in Section 4.5, and as is documented in work on LMs such as EPSS where the maintainers report the features that contribute most to the LM - exploit and exploitability indicators are only loosely related to each other, and are also related to factors such as the organization who maintains the software (e.g. Microsoft). Hence there is no clear “winner”. Furthermore, as can be seen in the Data Source(s) column (Column 15) of Table 4.6, the existing work primarily uses publicly available datasets which are known to provide higher coverage of large organizations such as Microsoft. More work is needed to understand how

these models generalize to less well-known organizations and smaller projects, or with data collected internally which may be too sensitive to release publicly.

4.7.1.1 Base CVSS

As discussed in Section 4.2, CVSS is one of the most common standards for software assessment. Since we are primarily concerned with the exploitability part of the assessment, we focus on papers in which learning algorithms are used to target the sub-components of CVSS, including those related to exploitability. As seen in Table 4.6, the subcomponents of the Base CVSS score provided by the NVD are used as a GT for S14, S38, S41, S42, S48, S51, S52, and S59.

4.7.1.2 Other Exploit-Related Indicators

While the Exploitability subcomponents of the Base CVSS score are an industry-recognized standard for “exploitability”, *there is less consensus on which other GT are indicators of “exploitability”*. In S03, S18, S26, S28, S30, S40, and S45 the authors indicate that their models predict concepts such as “exploitability” or “whether a vulnerability is exploitable”. However, S19, S20, S32, and EPSS do not indicate that they are explicitly “exploitability” models, instead indicating that they are models of “likelihood of exploitation”, i.e. the likelihood that an attacker will exploit a vulnerability. In S03, the authors assert that these concepts of “exploitability” and “likelihood of exploitation” should be highly related [33], while other researchers have found that indicators of exploitability may only explain part of the likelihood of exploitation [245, 9]. However, these models focus on the same practical observations, regardless of what it represents. As can be seen in Table 4.6, the “ground truth” used for training/testing (GT) “exploitability” models such as in S18 or S26 may be the same GT used in “likelihood of exploitation” models such as in S20 or S32. The lack of usability or methodically collected practitioner feedback on exploitability LM in the academic literature further complicates the conceptual classification of these models. Consequently, we focus on the actual observations used in each of the LM both for the ground truth (GT) and for the features (FT) used to predict GT.

4.7.1.2.1. *Exploitability Changes over Time: Temporal vs Nontemporal Models*

As previously noted by Le et al. [162] in their analysis of LM, temporal aspects of exploitability are a theme in LM. At least three studies (S03, S20, and S55) targeted a temporal value. We note which studies include a temporal model in the fifth column of Table 4.6.

In both S03 ([33]) and S20 ([80, 79]), the authors examine and compare a model for non-temporal, binary classification (i.e. whether an exploit exists); and a prediction model for *Time to Exploit*, i.e. the difference between when an exploit is publicly available and when

the vulnerability was disclosed. Both authors note that the time to exploit may be more useful for practitioners in determining how to budget their resources. However, both authors found the nontemporal model to have higher accuracy than the temporal model. In S03, the accuracy decreased from 89% to 79% [33] between the nontemporal and temporal models. In S20 ([80, 79]), Edkrantz found that optimal models for the nontemporal classification had 81-82% accuracy, while their attempts to construct the temporal model resulted in precision, recall, and F1 scores so low (all below 0.6) that the authors did not investigate the temporal model further.

In S55 ([270]), the authors had better results than in S03 or S20, but predict a slightly different concept. In S55, Suciú et al. develop a model to predict “*over time* the likelihood that a *functional exploit* will be developed” [270]. As described by the authors, “functional exploits go beyond proof-of-concepts (POCs) to achieve the full security impact prescribed by the vulnerability” [270]. The models of S55 had precision over 0.8 and recall over 0.6 with an overall AUC of 0.73 [270].

4.7.1.2.2. *Types*

The non-CVSS types of indicator used as GT labels can roughly be divided into four categories: Exploit Datasets, Exploit Signature Datasets, Exploit Flags in Vulnerability Datasets, and four data categories that were unique to a particular model which we classify as “Other” and discuss individually. Even within one of these categories - different data sources within a particular category can influence the performance of the resulting model. For example, as shown by Allodi and Massacci as early as 2012 in S08 ([9, 10]) and examined further by other research (e.g. S19 [246]), different exploit datasets may contain exploits for different vulnerabilities. Furthermore, as shown in Table 4.6 many papers use multiple data sources. Where multiple sources are used, a vulnerability only has to be determined as “exploitable” by one of the sources to be labeled as a True Positive [246, 16, 270].

Exploit Datasets The most common observation used as a GT label is whether an *exploit is available* in a dataset, being used in 8 of the studies. As seen in the last column of Table 4.6, the most common source of exploits is ExploitDB (used for GT in by S19, S20, S32, S40, S45, and S18). ExploitDB describes itself as an “archive of public exploits and corresponding vulnerable software, developed for use by penetration testers and vulnerability researchers” [87]. In S55 ([270]), the authors consider exploits from ExploitDB to be PoC exploits, rather than “functional” exploits. Instead, the authors of S55 used exploits from commercial databases: Metasploit⁵, a penetration testing tool with a corresponding

⁵<https://www.rapid7.com/products/metasploit/>

database, as well as other commercial exploit tools including Canvas⁶ and the D2 Elliot Web Exploitation Framework⁷. In S55, the authors also used exploits and other malware samples from the Contagio dataset which has been used in other research [313, 159, 224]. Similarly, in S30 the authors use exploits from a variety of datasets created in prior work including exploits previously used in the Mayhem publication from S06 (an EG study).

Exploit Signatures Another set of observations used as training/test data are sets of exploit signatures. As we discuss in Section 4.1.1, an exploit signature uniquely identifies a particular exploit as part of security tools such as Intrusion Detection systems. Exploit signatures are associated with “likelihood of exploitation” as much as exploitability [192, 139]. Exploit signature datasets take two forms:

- First is the existence of the signatures themselves, most frequently gathered from Symantec (a commercial vendor) as was done in S26 ([13, 14]) and S55 ([270]). Signatures are also available through the open-source databases available through the SNORT⁸ and Suricata⁹ intrusion detection platforms, which were used in earlier versions of EPSS [139] as part of their dataset from Proofpoint.
- The second form of signature data, the frequency with which the signatures are observed, is only used in two studies whose models are *not* explicitly exploitability models. Signature detection frequency is considered to be a more robust estimate of the likelihood of exploitation than the signatures alone [192]. However, telemetry data on the frequency with which vulnerabilities are exploited and the context in which exploitation occurs is inherently sensitive [76], and requires special permissions to access. The authors of S19 ([246]) used the Worldwide Intelligence Network Environment (WINE) dataset [76], a set of attack signature observations collected by Symantec between 2008-2014 [294] as part of their GT. More recently, developers and maintainers of the EPSS model ([139], [96]) work with AlienVault and Proofpoint, two commercial intrusion detection system vendors, to obtain more recent data on which exploits had been actively observed.

Exploit Flag in Vulnerability Database Some vulnerability databases include a specific flag about whether an exploit is known to exist for a particular vulnerability or has been seen in the wild, even if the database does not include the exploit itself or how the information was collected. For example, the OSVDB included information about vulnerabilities including

⁶<https://www.immunityinc.com/products/canvas/>

⁷<https://www.d2sec.com/elliott.html>

⁸<https://snort.org>

⁹<https://suricata.io>

whether a vulnerability had “an available, rumored, or private exploit” as well as the date an exploit was first recorded [33]. Since the OSVDB is no longer available, this observation is used less frequently. However, in S50 ([311]), published in 2020, the authors use the Vulners vulnerability database, which also has information on whether an exploit is available [311].

Other Four studies (S19, S55, S03, and S40) use GT data unique to those studies and not used in other models.

- In S19, Sabottke et al. [246] consider vulnerabilities with a Microsoft Exploitability Index (MSEI) [176] of 0 or 1 to be “exploited” as part of their GT
- In S55 ([270]), in addition to other GT described previously, the authors used the Exploit Code Maturity levels of the Temporal CVSS score from commercial sources¹⁰ including IBM X-Force Exchange and Tenable Nessus. The authors of S55 also use NLP rules to extract evidence of exploits and of exploitation in the wild from databases including BugTraq, Tenable, Skybox, and AlienVault OTX.
- In their model for *time to exploit* in S03, Bozorgi et al. [33] use additional data from prior work by Stefan Frei [100] to obtain more accurate information on when vulnerability exploits were released.
- In the second of the two models examined in S40 ([16]), the authors identify 12 vulnerabilities used by a threat actor known as APT 28. The authors note that vulnerabilities targeted by APT 28 all appear in one of the following services: Adobe Flash, Java, Windows, Microsoft Office, and Microsoft Word. The authors then label all vulnerabilities in these five services as likely to be exploited, assuming APT 28 will continue to invest resources in exploiting the same services.

All forms of exploit-related GTL have limitations. For example, Exploit Signature databases have limitations, in addition to accessibility. One limitation is the generalizability of *where the signatures are collected from*. For example, some types of vulnerability may be over- or under-represented in exploit signature data due to how current intrusion-detection and malware-detection systems are implemented [139, 96]. Input validation vulnerabilities that result in Cross-Site Scripting (XSS), the signature may target the attacker’s behavior of attempting to input a script rather than targeting a specific vulnerability. These generic signatures may not be mapped to a particular vulnerability, which can result in the under-representation of XSS in some datasets. However, exploit signatures are considered by some authors as a better indicator that a useful exploit can be created [270].

¹⁰the Temporal score is not available from the NVD or many other public databases [270]

4.7.2 What categories of information are commonly used as features (FT)

We categorize the features used in the different learning models into several categories, each of which is described in more detail in the following sub-sections. The features used in each paper are shown in Table 4.6.

4.7.2.1 Program Analysis

The first category of features used in models are those derived through program analysis. The studies identified in our survey that use Program Analysis as a feature include S52 ([179, 178]), S59 ([291]), S18 ([300]) and S30 ([273]) as shown in Table 4.6. The use of program analysis techniques in these learning models indicates some similarity between these models and the Deterministic tools discussed in Section 4.6.1, which also rely on program analysis.

In S52 ([179, 178]) and S59 ([291]) both rely on neural network models to extract features from source code. As indicated in the “# vuln” column of Table 4.6, for both S52 and S59 the LM performs its classification at the level of the actual statements in a commit (S52) or functions (S65), rather than the entire vulnerability which may contain multiple broken statements or functions. As seen in Table 4.6, the statements or functions map to a smaller number of vulnerabilities from the CVE list in the NVD, which is used in other models.

In the first study in S52 ([179]) features are extracted and classified at the commit-level, while the second study takes a more detailed approach - classifying the individual vulnerable statements within each commit. The authors of S52 then examine how different models for extracting the context of the vulnerable statement - such as examining the full function rather than just the vulnerability statement. The authors found that adding function context improved classification for all scores, but improved classification for AV and AC less than other categories. For example, AV and AC showed 6.4 and 6.5% improvements when function context was used instead of only the vulnerable statement, while AU showed a 9% improvement [178]).

In S59, which focuses on vulnerabilities in the Linux kernel, instead of basing the model on the functional source code - the model extracts function descriptions from the source code which have been formatted according to the kernel-doc format [291]. Precision and recall for the models in S59 for different exploitability values (AV, AC, AU) were relatively high, ranging between 86% and 95%.

In S18 ([300]), Younis et al. examine the discriminative power of eight metrics extracted from the function call graph of two applications: Apache HTTP server and the Linux Kernel. The authors also analyze the predictive power of these metrics when different feature selection and machine learning algorithms are applied. The authors found that Count Path, which measures the number of paths in the call graph (excluding abnormal exits and GoTos) which go through

the vulnerable function [254, 300], had the highest discriminative power according to Welch's t-Test [300]. Source Lines of Code (SLOC), a measure of code size; and Called-by Functions (also known as Out-Degree), which measures the number of functions called by the vulnerable function [300]; also had statistically significant discriminative power. The discriminative power was not statistically significant for any other features, including Cyclomatic Complexity and Calling Functions (also known as In-Degree). The authors then applied Correlation-based, Wrapper subset evaluation, and Principle Component Analysis (PCA) feature selection techniques with Logistic Regression, Naive Bayes, Random Forest, and Support Vector Machine machine learning algorithms. Each model performed best when paired with a different feature selection technique. The models' precision on the Apache HTTP Server ranged from 44% to 84%, while the models' recall ranged from 60% to 83%. However, for the Linux Kernel, none of the models had a recall score above 70%, the threshold set by the authors for acceptable model performance [300]. The authors suggest that one reason these program-analysis features are less predictive for the Linux Kernel may be that controlling the OS provides higher value to an attacker.

In S30 ([273]), the authors build a tool, which they refer to as Exniffer, to assess the exploitability of memory corruption vulnerabilities detected due to system crashes. As described by the authors, "A crash as a result of a safety-critical bug may not necessarily be exploitable as it may not depend on the malicious inputs, whereas a crash due to a security bug necessarily depends on the malicious inputs." [273]. Similar to EG models for memory corruption vulnerabilities, the authors start with an executable that is run in an instrumented environment. When a crash occurs, the authors extract "static features" such as the x86 instruction being executed at the time of the crash (e.g. `mov eax, [ecx]`), and the type of exception thrown (e.g. memory access violation, floating point exception, etc.). The authors also extract "dynamic features" using the PIN analysis program to simulate Last-Branch-Record (LBR), a program tracing functionality available in many processors. These dynamic features include whether the crash occurred during the execution of a loop, and the type of branch instruction (e.g. jump, function call, function return) most recently executed. Exniffer labeled the data using a variety of sources, including prior work and exploits found online and downloaded by the authors. They use an SVM algorithm to classify vulnerabilities as exploitable or non-exploitable, and applied Recursive Feature Elimination (RFE) to rank features. The top three features included corruption of a backtrace, Null Memory operand, and Executable Extended Instruction Pointer memory segment. Overall, their tool had a precision of 0.96 and a recall of 0.81.

4.7.2.2 Vulnerability Reports

The most common source of features in vulnerability prediction models is vulnerability information available in reports such as those provided by the U.S. National Vulnerability Database. Some of this information is available in pre-processed categories, such as CVSS Base Score vector elements, vulnerability type, and the product/vendor responsible for maintaining the software. However, these models also use features extracted from un-categorized text, such as the description field in the NVD reports, using natural language processing techniques. Precision and Recall for models that use Vulnerability Reports. Models using data from Vulnerability Reports generally have strong performance, although it varies between studies - suggesting that the success of these models is dependent on other factors being evaluated in the same study such as the GT. For example, Suciú et al. report AUC above 0.9 [270] for their temporal metric for exploit development which also include Exploit Data based features, while (non-temporal) experiments which led to EPSS [139] using slightly different GT sources and fewer Exploit Data based features had AUC between 0.78 and 0.85.

Some of the information available in reports, such as the vendor responsible for maintaining the code, may be difficult to extract from other sources. Another concern with vulnerability-report-based models is ensuring that the feature(s) of the model do not contain information about the GT that would be unavailable for vulnerabilities for which the GT was unknown. For example, in S45 ([298], the authors note that when training data is labeled with information, such as exploits in ExploitDB, and references that may include the link to the label data, e.g. an exploit from ExploitDB, are used as features for the LM; the features used will inherently contain the predicted variable. Using a wide range of features, they found that the use of reference-based features only improved a model if those references included the exploits used for GT.

4.7.2.3 Social Network

The use of social-network features was first proposed by Sabottke et al. in S19 ([246]) who build features based on Twitter posts that mention vulnerabilities, specifically CVEs. In addition to applying NLP to the content of the posts themselves, the authors of S19 used the following features: Number of tweets about the vulnerability; # users tweeting about the vulnerability with minimum T followers; # users tweeting about the vulnerability with minimum T friends; # retweets/replies; # replies; # tweets favorited; Avg # hashtags mentions per tweet; Avg # URLs mentions per tweet; Avg # user mentions per tweet; # verified accounts tweeting about the vulnerability; Avg age of accounts tweeting about the vulnerability; Avg # of tweets per account. In S19 the authors found that incorporating text and social network features from Twitter may

improve the precision of learning models for likelihood of exploitation.

In S26, the authors examine how “Darkweb” data from TOR network sites can be used to predict vulnerabilities using a dataset known as D2Web. As part of this work, they also examine the social network between users within the D2Web data, examining how measures, such as the in-degree and out-degree of the D2Web social network data of users who mention a vulnerability.

In S32 [34], the authors compare whether a model built from text features extracted from Twitter as done in S19 performs better than a model using only text features from the NVD and CVSS score-based features. In contrast with the prior work in S19, the authors of S32 find that the Twitter information does not consistently improve the model in terms of precision and recall; and its value is particularly questionable in scenarios where additional infrastructure will need to be established to extract information from Twitter. S55 similarly did not find Twitter features to be good predictors, and excluded them from their primary model. The different findings in the precision and recall between S32 and S19, highlight a problem found in several studies including S19’s attempts to compare their work with S03 - the replicability of results from learning models of exploitability is low. In particular, social-network features should be carefully examined to determine if they are worth the effort.

4.7.2.4 Exploit Data

In comparison with the other studies that use exploit information as one of many FT, in S55 ([270]) Suciú et al. propose a novel model primarily based on exploit information. In S55, the model is designed to predict when a *functional* exploit will be developed, i.e. an exploit that can achieve an attacker’s goals. Their model extracts FT from exploits that have been already released, but which may only cause unexpected behavior, such as a crash. The exploits used as FT are referred to as Proof of Concept (PoC) exploits, which are extracted from ExploitDB, BugTraq, and Vulners databases. The FT in S55 include the programming language of the PoC; # of language-reserved keywords used in the PoC; measures of code size and complexity of the PoC; n-grams (e.g. words) extracted from the PoC; and n-grams extracted from text and comments in the PoC. The authors found that models based on combined feature sets with PoC exploit features, language n-grams from vulnerability write-ups, CVSS scores, product/vendor information, and CWE types performed better in terms of AUC than using the PoC exploit information FT alone, or using the other FT without exploit information.

4.7.2.5 Timeline

In three (3) studies, the features used in the LM include features based on the time of the publication for vulnerability information, and other available information on vulnerability and exploit timelines. Most of these features are based on the timeline features used in S03 ([33]). In S03, Bozorgi et al. use features that include key dates extracted from the OSVDB and CVE, the difference between the CVE last modified date of the CVE documentation and the date the CVE was created, and the difference between the last modified date and the date the vulnerability was published on OSVDB [33]. These last two features - the differences between the last modified date and the creation and disclosure dates - were among the top 10 weighted features which were therefore also used by Sabottke et al. in S19 to evaluate how much different types of features may contribute to a learning model for exploitability. Similarly, in S50 ([311]), Zhang et al. extracted features from the Vulners database including the difference between the modified date and the original published date for the vulnerability, and the difference between the last seen date and the original publish date. EPSS ([139, 96]) also uses a timeline measure - the number of days since the CVE was published, which is in the top 30 contributing features for EPSS [96].

4.8 Other Probabilistic Assessments (Non-LM)

We identified four studies, S02 ([101]), S24 ([239, 240]), S22 ([256]) which propose or evaluate exploitability assessments that rely on probabilistic models but are not based on Machine Learning.

In S02 ([101]) and S24([239, 240]), the authors use an equation based on a Pareto Distribution observed by Frei et al [99] as part of a larger severity or risk metric. In S02, the authors use the Frei equation to estimate part of the Temporal metrics from CVSS, while S24 uses the Frei et al equation to build Nonlinear Statistical Models to estimate the probability of being exploited as a function of time.

4.8.1 Exploit Availability Distribution Function from Frei et al.

In 2006, Frei et al [99] examined vulnerability and exploit information based on vulnerabilities published in the OSVDB and NVD from 1996-2006. Their analysis focused on four points in the vulnerability lifecycle - the time of vulnerability *Discovery*, the time of vulnerability *Disclosure*, the (earliest) time of *Exploit* availability for the vulnerability, and the time of *Patch* availability. The timeline for each vulnerability was extracted through a number of sources including three public organizations: - the CERT organization at Carnegie Mellon and the French Security

Incident Response Team, and the milw0rm hacktivist group, as well as four companies that provide security-related software: Internet Security Systems (ISS) X-Force (now IBM X-Force), Secunia, Symantec SecurityFocus, Packetstorm, and Metasploit [99]. As part of their analysis, they identified a function matching the distribution of *exploit availability* in terms of time t since *Disclosure*:

$$F(t) = 1 - \left(\frac{0.0016}{t} \right)^{0.260}$$

The *Exploit Availability* function determined by Frei et al. was used by two studies in our survey, S02 ([101]) and S24 ([239, 240]) to estimate Exploitability.

As discussed in Section 4.2, the NVD only provides the Base CVSS score and subcomponents, and does not provide estimates of the Temporal or Environmental scores. However, several authors have proposed using Probabilistic assessments to estimate the Temporal scores, including the Exploit Code Maturity metric. In S02 ([101]), Frühwirth and Mannisto analyze the difference in CVSS v2 metrics when they use probability distributions to approximate the Temporal and Environmental metrics. For the “Exploit Code Maturity Metric”, the primary Exploitability-specific Temporal or Environmental metric in CVSS v2, the authors use the equation from Frei et al. [99]. In S02 ([101]), the authors found that on a set of 720 vulnerabilities recorded between January 5 and March 20 2009, the CVSS score computed using their probability-based system for the Environmental and Temporal metrics resulted in differences ranging from a 2.5 point decrease to a 2.5 increase from the score computed using default values, with 60% of scores having a 0.5-1% decrease from the default values. Overall, this increased the number of “Low” severity vulnerabilities, which the authors argue would result in cost-savings, assuming that lower-severity vulnerabilities are less expensive to fix [101].

Similarly, in S24 ([239, 240]) the authors use the same model fit by Frei et al to estimate an “exploitability” score they describe as the likelihood that a vulnerability will be exploited before being patched or disclosed. The authors combine this exploitability score with the overall CVSS Severity score and an estimate of the likelihood that a vulnerability will be patched, also based on Frei’s work, to develop nonlinear statistical models to estimate the probability of the exploitation of vulnerability as a function of time. This model is intended to improve on their previous model ([239]), which used the CVSS Base Exploitability metric as part of their process. The authors of S24 also refer to their overall models of the probability of exploitation as “Exploitability” models [240]. In S24 the authors evaluate their Risk models using R^2 and Residual analysis. R^2 values can range between 0 and 1, where 1 indicates that the targeted phenomenon (e.g. Risk) can be perfectly explained by the predictors [72]. The adjusted R^2 value is used to account for larger numbers of predictor variables in a model. The adjusted R^2 for the initial model in S24 was 0.85 [239], while the most advanced model had an adjusted R^2

of 0.96 [239].

4.8.2 Developing Prioritization Rules based on Exploit Likelihood Analysis

In S22 ([256]), the authors use a Cox Proportional Hazard survival model to examine how vulnerability characteristics available through the OSVDB such as the Severity, whether the software is open-source, and whether the vulnerability has been disclosed relate to the amount of time between when a vulnerability is discovered and when an exploit is published. The authors found that Disclosure Status (whether the vulnerability had been disclosed at the time of discovery) was the strongest predictor of the length of time between when a vulnerability is discovered and the exploit is published, followed by severity. The authors propose a set of rules based on the results of their analysis, such as “Immediately disclosed, highly severe, remote vulnerability targeted at open-source, infrastructure software” should be a top priority for patching [256]. The authors suggest that a similar modeling and rule-development process could be used by managers to prioritize patching efforts.

4.9 Limitations

Given the volume of exploitability research, some published studies may not appear in this survey. However, although this is not a formal Systematic Literature Review (SLR), we used the SYMBALS methodology associated with literature reviews to reduce the likelihood that we would miss key papers.

We may have introduced bias in our selection of papers. However, we had at least two researchers involved in our paper selection process for both Phase 1 (Keyword Search with Active Learning) and Phase 2 (Snowballing), as noted in Section 4.3 to reduce the risk of introducing bias from a single individual.

Additionally, a potential limitation of the study is that the characteristics and categorization in our study are unreliable or incomplete. We do not claim that our categorization or the characteristics examined in this study are the only possible categories and characteristics of note in exploitability research. However, we base our categorization on prior work to improve its reliability.

In Table 4.5, we use the range of values reported for Time to Run as our reported statistic. We recognize that Range is not the most descriptive statistic [72]. However, Range was the statistic we could extract most consistently across studies for time to run and provides some estimation of variability [72] as well as the known “worst case scenario” for run times. Where possible, we use footnotes for Table 4.5 and our discussion in Section 4.6.1.4 to provide additional

information on studies where the minimum and maximum are particularly unrepresentative, such as S06.

4.10 Discussion

We begin our discussion in Section 4.10.1 examining high-level trends in exploitability assessment research. In Section 4.10.2 we examine how the categories of exploitability assessment techniques might be useful to practitioners, depending on the values and context in which the techniques are being deployed. Finally, in Section 4.10.3, we focus on gaps and future directions that may be useful to researchers.

4.10.1 Temporal Trends

In this section we examine temporal trends and cross-cutting factors in exploitability assessment systems. Since Table 4.4 is ordered by publication year, it is a useful reference for observing and discussing temporal trends

CVSS - less novel, more standard:

As can be seen Table 4.4, earlier research (e.g. research prior to 2015) tended to focus on evaluating and proposing CVSS based models. Later work such as S43, S55, and S56 seems to have accepted CVSS as a baseline against which to compare other exploitability assessments.

Rise of the (Learning) Machines:

Another trend seen in Table 4.4, is that while the volume of CVSS-based evaluations have declined, LM have increased. This trend is additionally reflected in the number of recent surveys have focused exclusively on security LM as shown in Section 4.1.2.

Increased focus on Program Analysis:

As noted in Section 4.7, within the Probabilistic LM models, we see an increasing trend towards using program-analysis-based features. This may be due to improvements in program analysis techniques, which have also spurred innovation on Deterministic assessments leading to deterministic tools such as S21 being available and in-use in industry settings.

4.10.2 Aligning with Industry values for vulnerability prioritization

As highlighted in work by de Smale et al. [259], practitioners make tradeoffs in determining what vulnerabilities to prioritize or even what information to collect for vulnerability management. What methods and tools are useful for a particular organization will depend on the tradeoffs they make. We use the three sets of value tradeoffs identified by de Smale et al. obtained via interviews with practitioners to frame our discussion of the different vulnerability detection techniques.

In this discussion, we are not proposing that one technique should be used *instead of* another. We use de Smale's sets of opposing values to highlight how some categories of technique may align more strongly with certain values.

Independent Analysis vs. Trust

The first set of values discussed by de Smale et al. [259] is how much organizations *trust* information provided by others, compared with the time and effort required to perform an in-depth, *independent* analysis of the vulnerability in the organization's own context. CVSS scores provided by a third party, such as the NVD requires a high amount of trust. Similarly, many LMs leverage vulnerability reports from the NVD and other sources, requiring a high amount of trust in those sources. As noted by Ponta et al. [232] in their "Lessons Learned" as part of transitioning S21 into industry practice [232], developers may be particularly skeptical of metadata-based assessments, and as shown in Table 4.6, many of the proposed LM approaches for exploitability assessment are based on metadata from vulnerability reports. However, if an organization has sufficient data from its own sources, it may be able to build and use a more *independent* LM. Using Deterministic tools may provide more *independent* information for that organization's context, but do require time to run. Similarly, evaluating CVSS manually requires expertise and time, but can be done *independently*, with the Environmental metrics of CVSS specifically designed to be tailored to a particular context.

Proactive vs. Reactive

Smaller organizations tend to take a primarily *reactive* approach [259] and rely on information pushed to them by government organizations - and may therefore be more likely to rely heavily on CVSS scores from the NVD. Gathering sufficient information to leverage an LM model requires a highly *proactive* approach, while Deterministic methods lie somewhere in-between. Having LM scores publicly available, such as with EPSS, may facilitate the use of LM results in a reactive context.

Formalized vs. Ad-Hoc Processes

Probabilistic approaches, which require a collection of vulnerabilities to perform statistical analysis on may be difficult to incorporate into ad-hoc processes. As with *Proactive vs. Reactive* decisions, the availability of a pre-calculated EPSS score may mitigate the need for a *formalized* process but also reduces the ability to tailor the score to the organization's context. CVSS-based scores, however, can be more readily calculated on an individual basis. Similarly, the Program-Analysis-based Deterministic assessments may be more readily incorporated into ad-hoc processes, although some work may be required to set up, configure, and run the program analysis tools. As discussed in Section 4.6.1.4, given the time required to run many of the Deterministic Program-State-Based assessments on a single vulnerability - these approaches may be less useful if an organization is attempting to analyze all vulnerabilities at once on a regular basis as part of a formalized processes. The authors of S21 found that the vulnerability detection component in their tool, which only required 76 seconds to run, could be used as part of frequent, systematic scans at the company SAP [232]. However, the full exploitability assessment required hours to run and was more likely to be used closer to a release date or to supplement manual analysis.

4.10.3 Research Gaps and Future Directions

We found studies that used other assessment methods as a control group in their experiments, i.e., to demonstrate that their proposed technique is an improvement. We found relatively little research on how assessment methods might work well together. One of the closest studies in this regard may be S22 ([256]) discussed in Section 4.8.2 in which the authors propose building more deterministic rules for vulnerability prioritization based on the results of a statistical model. However, even in S22, the study focuses on the relationship between factors within the statistical model rather than evaluating the combination of techniques. The length of time and resources to run many of the Deterministic, Program-State-Space models discussed in Section 4.6.1 may make it difficult for such tools to be used at a scale that could contribute to developing an LM. However, future research may focus on the assessment techniques that could and should be used to better determine when to deploy the Program-State-Space models.

The usability of Probabilistic exploitability assessment techniques is understudied. Studies S34 ([11]) and S44 ([12]) examine user concerns when applying CVSS. For deterministic studies, S57 determined that there were misconceptions among experts about the relative merits and drawbacks of different features in exploit generation tools for heap-based vulnerabilities, while S21 ([229], [230], [232], [132]) includes a summary of lessons learned from transitioning their deterministic tool into practical use at SAP. However, we could not find studies looking into any

usability-related aspects of the Probabilistic models and no formal usability studies of many of the deterministic models.

On the other hand, while work such as the reliability analysis of exploit stabilization mechanisms in exploit generation tools for heap overflow vulnerabilities in S57 ([309]) and larger-scale analyses in studies such as S21 ([229],[230],[232], [132]) begin to look at effectiveness using more systematic, scalable evaluations, most studies examining Program-State-Based programs focus on smaller vulnerability datasets. While smaller datasets may be understandable due to the complications that arise from the state-space explosion, developers like to know how precise a tool is expected to be [232]. However, to provide such information accurately, it may be beneficial to perform additional research into what practitioners mean by “exploitability” and what they expect from exploitability assessment.

4.11 Conclusion

We surveyed 59 studies and 2 standards covering 76 papers proposing or evaluating exploitability assessment methods for software vulnerabilities. These exploitability assessment methods can be divided into three groups: CVSS-Based, Deterministic, and Probabilistic assessments using a similar structure to the metric taxonomy proposed by Pendleton et al [221]. Deterministic, State-Based assessments and Probabilistic Learning Model assessments are the most prominent subcategories, with over 20 studies of assessments in each of the State-Based and LM categories.

Over the years, exploitability assessment has evolved from a theoretical analysis to multiple, practical implementations from CVSS to the Deterministic EclipseSteady tool examined in S21, to the Probabilistic EPSS. We have, as a community, answered the question *How can we assess the exploitability of software vulnerabilities?* What remains only partially explored, however, is *How **should** we assess the exploitability of software vulnerabilities?* Are some assessment methods easier to integrate with existing practices? What are the long term benefits of using different methods? Many questions remain unknown or underexplored.

Table 4.6: Learning Model(s) Examined in Each Study

ID	Bib.	Year	Ground Truth (GT)						Features (FT)				Data Source(s) for Training / Testing Ground Truth (GT), and Features (FT)	# vuln.	Modeling Technique(s) Examined	
			Base CVSS	Temporal (T), or Nontemporal (N), or Both (B)	Exploit	Sig	DB Flag	Other	Program Analysis	Vulnerability Report	Social Network	Exploit Info				Timeline
S14	[296]	2015	X	N							X	X		NVD (GT, FT)	61300	SLDA
S38	[267]	2018	X	N							X	X		NVD (GT, FT)	99091	RF, Decision Tree, GBM
S41	[113]	2019	X	N							X	X		CVE Details (GT, FT)	95147	Multi-task learning with softmax task-specific classifiers; CNN, BiLSTM, & attn. BiLSTM
S42	[161]	2019	X	N							X	X		NVD (GT, FT)	105124	NB, LR, SVM, RE XGBoost, LGBM
S48	[81]	2020	X	N							X	X		NVD (GT, FT)	103979	LR
S51	[140]	2021	X	N							X	X		CVD & NVD (GT, FT), SecurityFocus (GT, FT), Vendor Websites & Technical Reports (GT, FT)	93311	KNN, LR, NBSVM, LSTM-ANN, MLP, Majority-Vote Ensemble
S52	[179, 2021]	2021	X	N							X			NVD (GT), Github (GT, FT), VulasDB (GT, FT)	429	CNN, Attention GRU, CodeBERT; LR, SVM, KNN, XGBoost, LGBM, k-means cluster
S59	[291]	2022	X	N							X			NVD (GT) Linux & Android source code (FT)	65	attn. BiLSTM and softmax activation func.
S03	[33]	2010		B			X				X			OSVDB (GT, FT), MITRE (FT)	14764	SVM
S19 ^a	[246]	2015		N		X	X	X			X	X		ExploitDB (GT), Microsoft Advisories (GT), Twitter (FT), OSVDB (FT), NVD (FT)	5865	SVM
S20 ^a	[80]	2015		B		X					X	X		ExploitDB (GT), NVD (FT), Recorded Future (FT)	55914	SVM, KNN
S26	[13], [14]	2017		N			X				X	X		Symantec (GT), NVD (FT), ExploitDB (FT), D2Web (FT), Zero-Day-Initiative (FT)	12598	RF, LR, NB, RF
S32 ^a	[34]	2017		N		X					X	X		ExploitDB (GT), NVD (FT), Twitter (FT)	38129	SVM
S40	[16]	2019		N		X		X			X	X		ExploitDB (GT), NVD (GT, FT)	112503	LR, RE, Neural Network
S45	[298]	2020		N		X					X	X		ExploitDB (GT), NVD (GT, FT)	123454	LR, SVM, RE, DT
S50	[311]	2020		N			X				X	X		Vulners (GT, FT)	5325	Neural Network
S55	[270]	2021		T		X	X	X			X	X		Tenable (GT), Canvas (GT), D2 (GT), Symantec (GT), IBM X-Force (GT, FT), Metasploit (GT, FT), NVD (FT), ExploitDB (FT), BugTraq (FT), Vulners (FT), VirusTotal (FT)	3119	Feedforward Neural Network
EPSS ^a	[139, 2020]	2020		N							X	X		Proofpoint (GT), Fortinet (GT), AlienVault (GT), MITRE (FT), NVD (FT), ExploitDB (FT), Metasploit, Elliot, & Canvas Frameworks (FT), Github (FT)	210223	XGBoost, LR, RF, SVM, Dense Neural Network
S18	[300]	2016		N		X					X			ExploitDB (GT), Program Analysis (FT)	183	LR, NB, RE, SVM
S28	[297]	2017		N			X				X			Fuzzing results (GT), Program Analysis (FT)	2200	Bayesian reasoning-based
S30	[273]	2017		N		X					X			Exploits Downloaded by Authors and from Prior Work (GT), Program Analysis (FT)	585	SVM

^aS19, S20, S32, and EPSS use exploitability as a feature but do not refer to their model as an "exploitability" model

CHAPTER

5

OPENSSEF SCORECARD AGGREGATION

Interviews and surveys of practitioners [164, 287, 66, 188, 157, 189, 295, 1, 2] have found that the primary factor practitioners use to decide whether to include an open source component is the functionality and features provided by the component. However, other factors such as the presence of vulnerabilities and signs of community support and adoption can also play a role in the decision [164, 287, 66, 188, 157, 189, 295, 1, 2].

The Open Source Security Foundation (OpenSSF) Scorecard is a tool designed to help practitioners decide whether to use a particular third-party component by automatically collecting a set of measures associated with security risk [209]. OpenSSF Scorecard measures cover many of the common attributes previously highlighted in interviews and surveys of practitioners, such as the number of *Vulnerabilities* and measures of whether a project is actively *Maintained* [164, 287]. As an emerging standard, OpenSSF Scorecard is developed by practitioners from organizations including Google, Snyk, and the Linux Foundation.

Measuring the risk for the package and its direct dependencies tree may only be the tip of the iceberg. Zerouali et al. [310] found that more than 15% of the software packages in the NodeJS Package Manager (npm) are exposed to security vulnerabilities in their direct dependencies. However, 36.5% of npm packages are exposed to vulnerabilities indirectly via transitive dependencies¹.

¹See Ch. 2 Section 2.2.2 for an explanation of “direct” and “transitive dependencies”

The goal of this research is to aid developers and software architects in making good component choices through an analysis of the relevance and utility of security measures (OpenSSF metrics) of the entire dependency tree associated with a package.

Our overall research question for this study is: *How should security scores from throughout a package's dependency tree be aggregated?* First, we perform a set of preliminary interviews to help us develop and refine our aggregation approaches. These interviews also helped us identify and minimize potential pitfalls for our later analysis. For the main part of our research, we take two approaches. For one, we perform a large-scale data analysis of packages in the Go language ecosystem to answer the following research questions:

- RQ3.1a: *Depth*: What is the relationship between scorecard scores and the average depth of a package in a dependency tree?
- RQ3.1b: *Difference*: What is the difference between the original score and the aggregate score?
- RQ3.2c: *Vulnerabilities*: How does including aggregate information from a package's dependency tree change the relationship between security metrics and vulnerabilities?

The other approach is a survey of practitioners, to address the following research question:

- RQ3.2: What approaches do practitioners recommend for aggregating measures across a component's dependency tree?

The contributions of this work are as follows:

- an analysis and dataset of security risks in the Go language ecosystem;
- an analysis of the relationship between security metrics and depth;
- a comparison of security scores for the base component against security scores that aggregate information from the component's entire dependency tree; and
- a survey of practitioners to better understand how they would prefer security scores to be aggregated.

The remainder of this chapter is organized as follows. Section 5.1 provides additional background and technical details of the OpenSSF Scorecard, which are necessary to understand our approach. Section 5.2 discusses the ecosystem we will use as the target of analysis in our study. Section 5.3 discusses the closely related literature and its implications for our work and explains how we expand beyond the prior work. In Section 5.4 we discuss a series of preliminary

interviews we performed, which informed our later methodology. In Section 5.5 we discuss the dataset we used in our research. In Section 5.6, we describe the details of our methodology. In Section 5.7 we describe the aggregation methods we examined with our methodology. In Section 5.8 we describe our results. In Section 5.11 we discuss the implications of our results. In Section 5.9 we discuss the limitations of the paper. In Section 5.10 we highlight the ethics practices we followed in our research. In Section 5.11 we discuss the implications of our work. Finally, Section 5.12 is our conclusion.

5.1 OpenSSF Scorecard

The Open Source Security Foundation (OpenSSF) [207] is a cross-industry organization to promote technical initiatives and to develop best practices for securing open-source ecosystems, such as Github and the npm registry. Members of OpenSSF include Github, IBM, Google, Cisco, Apple, Microsoft, Sonatype, f5, and other industry leaders. The OpenSSF Scorecard is one of the technical initiatives promoted by OpenSSF.

5.1.1 The Checks

The measures assessed by OpenSSF Scorecard are separated into 18 checks. The names and descriptions of each check from OpenSSF are shown in the first two columns of Table 5.1. We discuss names, descriptions, and how they are grouped together in Section 5.1.1.1. Each Scorecard check has a *risk* level, similar to the “risk” and “severity” levels of the checks in SAST and DAST tools described in Chapter 3. The risk level for each check is listed in the third column of Table 5.1, and is discussed in Section 5.1.1.2.

The fourth and fifth columns of Table 5.1 provide information on the *method* through which each check measures the entity and attributes. In the fourth column of Table 5.1, we include the Constituent Elements of each check to help provide context on what the measurement is assessing, which are briefly discussed in Section 5.1.1.3. To enable readability, we provide an overview of these elements in Table 5.1 with details on how each check is computed in Table D.1 in Appendix D. In the fifth column of Table 5.1, we indicate the Standardization approach used for each check to convert the constituent source measures into a score between 1 and 10 inclusive. We discuss Standardization approaches in Section 5.1.1.4.

Table 5.1: OpenSSF Checks

Name	Description ²	Risk	Constituent Elements	Standardization
GROUP: Holistic Security Practices - Code Vulnerabilities				

Table 5.1 (Continued)

Name	Description	Risk	Constituent Elements	Standardization
Vulnerabilities	“Does the project have unfixed vulnerabilities?”	High	Vulnerabilities identified with OSV Scanner	Deduction
GROUP: Holistic Security Practices - Maintenance				
Dependency Update Tool	“Does the project use tools to help update its dependencies?”	High	Github Apps and/or Github workflows (checks for: Dependadabot, Renovate Bot, Sonatype Lift, PyUp)	Binary
Maintained	“Is the project maintained?”	High	Archived Status, Activity (Commits & Comments on Issues)	Case-Based: Proportional and Binary
Security Policy	“Does the project contain a security policy?”	Med.	Static Analysis to identify SECURITY.md file and examine its contents	Points-Based
License	“Does the project declare a license?”	Low	Static Analysis to identify license files and examine their contents	Points-Based
Best Practices	“Does the project have an OpenSSF Best Practices Badge?”	Low	Best Practices Badge in (Github) metadata	Case-Based: Points-based
GROUP: Holistic Security Practices - Continuous Testing (Test)				
CI Tests	“Does the project run tests in Continuous Integration (CI)?”	Low	Git metadata about Commits & Pull Requests	Case-Based: Proportional and Binary
Fuzzing	“Does the project use fuzzing tools?”	Medium	OSS-Fuzz project list. Static Analysis to find files and functions associated with a particular fuzzer.	Binary
SAST	“Does the project use static code analysis tools?”	Medium	Github Apps and/or Github workflows with Sonar or CodeQL	Case-Based: Proportional and Binary
GROUP: Source Risk Assessment				
Binary Artifacts	“Is the project free of checked-in binaries?”	High	Binaries identified with source code (static) analysis	Deduction

Table 5.1 (Continued)

Name	Description	Risk	Constituent Elements	Standardization
Branch Protection	“Does the project use Github Branch Protection?”	High	Github Settings	Points-Based
Dangerous Workflow	“Does the project avoid dangerous coding patterns in GitHub Actions?”	Critical	Static Analysis of Github Workflows	Binary
Code Review	“Does the project require code review before code is merged?”	High	Code Change (i.e. commit) & related metadata	Proportional
Contributors	“Does the project have contributors from multiple organizations?”	Low	authors of code changes, and their organization	Proportional
GROUP: Build Risk Assessment				
Pinned Dependencies	“Does the project declare and pin dependencies?”	Medium	Dockerfiles, shell scripts, and/or Github workflows	Proportional
Token Permissions	“Does the project declare GitHub workflow tokens as read only?”	High	Github workflow yaml files	Deduction
Packaging	“Does the project build and publish official packages from CI/CD, e.g. GitHub Publishing?”	Medium	Github Packaging Workflows and/or Github Actions	Binary
Signed Releases	“Does the project cryptographically sign releases?”	High	filenames of “release assets” on github	Proportional

5.1.1.1 Group, Name, & Description

As shown in Table 5.1, the 18 OpenSSF Scorecard Checks are divided into five groups, which fall under three themes. The first theme, “Holistic Security practices”, contains three groups: *Code Vulnerabilities*, *Maintenance*, and *Continuous testing*. The other themes, *Source Risk Assessment* and *Build Risk Assessment*, each have one group, with the same name as the theme [209].

The name of each check indicates the concept that the check attempts to capture. The description in column 2 of Table 5.1 is taken directly from the OpenSSF Scorecard documenta-

tion [209]. These descriptions provide additional clarification on the intent behind the name.

5.1.1.2 Risk

Each check has an associated level of risk: Critical, High, Medium, or Low. As we will discuss in Section 5.1.2, the risk level is used to compile the overall security score, with critical risks weighted at 10, high risks weighted at 7.5, medium risks weighted at 5, and low risks weighted at 2.5 [209]. The risk level is representative of the security risk and not necessarily the extent to which the check is used in overall decision-making. For example, the “License” check is given a “Low” security risk, but is commonly seen in interviews in surveys of practitioners as a key part of determining whether to incorporate a component [188, 287, 164, 3].

5.1.1.3 Constituent Elements

In Table 5.1, we provide high-level overviews of the constituent elements collected for each check. For example, the *Vulnerabilities* check collects the # Vulnerabilities using the OSV scanning tool. In contrast, points for the *Security Policy* check are based on static analysis of the SECURITY.md file. Further details on how each check is computed are available in Table D.1 in Appendix D.

5.1.1.4 Standardization Approaches

OpenSSF Scorecard results are standardized to a value between 1 and 10. As is illustrated in the existing aggregation of the *Vulnerabilities* check, which we will discuss further in Section 5.1.3, the standardization approach may influence how the measure should be aggregated.

With the exception of the *Best Practices* and *Token Permissions* Checks, the standardization approaches used by OpenSSF checks can be grouped into four types: Deduction, Binary, Points-Based, and Proportional. Some checks may apply different approaches on a case-by-case basis, which we indicate with the “Case-Based” label in the “Standardization” column of Table 5.1. For example, the SAST check uses binary standardization to capture the use of the Sonar SAST tool, but proportional standardization is used to capture other SAST tools.

These four main standardization approach types can be described as follows:

Deduction Deduction standardization approaches deduct constituent elements, such as the # Vulnerabilities in the *Vulnerabilities* check, from the maximum score (10). If the subtracted value is less than 0, the tool outputs 0. Otherwise, the tool outputs the result of the subtraction. The *Vulnerabilities*, *Binary Artifacts*, and *Token Permissions* checks use Deduction standardization.

Binary Binary standardization approaches are those that produce a binary output (either 0 or 10) based on the presence or absence of attributes. The *Dependency Update Tool*, *Fuzzing*, *Dangerous Workflows*, and *Packaging* checks are all Binary checks. Additionally, the *Maintained*, *CI Tests*, and *SAST* checks are binary for some cases.

Points-Based In Points-Based checks, the package is awarded points based on measurements taken of multiple attributes. For example, in the *License* check, 6 points are awarded if a license is detected, and an additional 3 points are awarded if the license file is the top-level directory, with the final (1) point awarded based on the license type. The *Security Policy*, *License*, *Best Practices*, and *Branch Protection* checks all use Points-Based standardization.

Proportional Proportional standardization approaches determine the expected output of the constituent elements, divide it by the expected (minimum) output, and multiply by 10. In cases where the result would be greater than 10, the tool outputs 10, otherwise the tool outputs the result. The *Code Review*, *Contributors*, *Pinned Dependencies*, and *Signed Releases* checks all use *Proportional* standardization. The *Maintained*, *CI Tests*, and *SAST* checks use proportional standardization for some cases.

5.1.2 Calculating the Overall Scorecard Score

OpenSSF Scorecard includes an overall Security Score. This score is the weighted average across all checks. Each check is weighted based on the risk value. Critical checks receive a weight of 10, high-risk checks receive a weight of 7.5, medium-risk checks receive a weight of 5, and low-risk checks receive a weight of 2.5. In other words, the overall score is calculated as:

$$\frac{\sum s_i * r_i}{\sum 10 * r_i}$$

where s_i is the score for the i th check (e.g. the score for the *Vulnerabilities* check for the system), and r_i is the weight of the risk associated with that check (e.g. 7.5 since the *Vulnerabilities* check has “High” risk)

5.1.3 Existing Aggregation Across the Dependency Tree for *Vulnerabilities*

One check, the *Vulnerabilities* check, is already aggregated across the dependency tree [208]. As shown in Table 5.1, the *Vulnerabilities* check is a *deduction* check. Specifically, in the case of the *Vulnerabilities* check, the score is normalized by subtracting the number of vulnerabilities from 10 and taking the maximum between the subtracted value and 0 (to ensure the value is never less than 0). To aggregate the *Vulnerabilities* check across the dependency tree, OpenSSF

collects the # Vulnerabilities from the entire dependency tree instead of only from the base component. Hence, this aggregation method heavily depends on the *Standardization* method.

5.1.4 The OpenSSF Scorecard Tool

OpenSSF Scorecard has an automated tool for calculating each individual check. The tool is run bi-weekly to populate a database available through Google BigQuery [209]. Scorecard data is also available directly through the OpenSource Insights deps.dev website. Currently, OpenSSF Scorecard is only able to analyze data for packages whose source repository is hosted on Github or Gitlab [209]

5.2 Go (The Programming Language)

Go, also referred to as Golang, is a statically typed, compiled programming language designed by a team at Google that included Robert Griesemer, Rob Pike, and Ken Thompson [53, 228]. Go is designed to produce faster, more efficient programs, compared to languages like Python, while still being more usable for programmers than C and C++.

5.2.1 Package Management in Go

Go uses a partially decentralized package management system, in which package IDs and the location of their source code are published in an index. The current primary index, managed by Google, is the Go Index: <https://index.golang.org>.

5.2.2 Why Go?

We focus on Go for several reasons. First and foremost is the feasibility of building an accurate model of individual packages with individual Scorecard check values, given the current strengths and limitations of both the OpenSSF Scorecard tool and the package index. Technically, the OpenSSF Scorecard tool equates packages to individual source repositories for the purpose of collecting and analyzing data about those packages. Since the Go package index also directly links package IDs to source repositories, which are often stored on Github, how the concept of a “package” is represented in the OpenSSF Scorecard tool is more easily mapped to how a “package” is represented in the Go package index. This contrasts, for example, with the NPM ecosystem where packages and source code are stored separately.

Go is a language currently used by developers, particularly at large corporations and in cloud environments. Companies that currently use Go in part of their software include Google,

Microsoft, Capital One, American Express, Uber, and Netflix³. Go is recommended by the NSF as a memory-safe language [191]. As a static, compiled language that is type-safe and memory-safe, Go is similar to languages such as Java and C++, which are commonly used in web development [53].

5.3 Related Work

In this section, we highlight related work which provides background to the research. In Section 5.3.1 we discuss studies which have examined why and how practitioners select third-party libraries. In Section 5.3.2 we discuss work that has looked at how vulnerability-related metrics have been aggregated throughout the dependency tree.

5.3.1 Understanding Third-Party Component Selection

Academic research [164, 287, 66, 188, 157, 189, 295, 1, 2] has used surveys and interviews with practitioners to identify the attributes and measures used to select open-source, third-party components to be used in their software, i.e. to determine whether to add a particular package to their supply chain. Their findings are similar to the industry-developed measures of OpenSSF Scorecard. We expand on prior work by further examining how measures from throughout the dependency tree of the component being selected should be aggregated. We summarize these survey and interview studies in this section.

Existing interview and survey studies provide valuable insights into practitioner preferences, but do not yet look into how values from throughout the dependency tree should be aggregated

Li et al. [164] interviewed 23 practitioners to determine what measures they use to determine whether to incorporate a particular open-source component into their software, i.e. when to add a particular package to their supply chain. The study participants included developers, project managers, and software architects. From these interviews, Li et al. identified a range of attributes which they organize into 8 main concepts and 46 sub-concepts for selecting open-source components, as well as over 100 different measures that mapped to the concepts and sub-concepts. The 8 main concepts were: *Community support and adoption* (mentioned by 10 practitioners), *Documentation* (mentioned by 14 practitioners), *License* (mentioned by 21 practitioners), *Operational Software characteristics* (mentioned by 6 practitioners), *Maturity*

³<https://go.dev/solutions/case-studies>

(mentioned by 6 practitioners, *Quality* (mentioned by 6 practitioner) which includes a “Security” subconcept (mentioned by 15 practitioners), *Risk* (mentioned by 7 practitioners), and *Trustworthiness* (mentioned by 6 practitioners).

Wermke et al. [287] interviewed 25 practitioners on challenges in the open-source supply chain. Their analysis also identified selection measures for open-source components. They found that popularity measures (mentioned by 16 participants), a large and active community (mentioned by 11 participants), specific features (mentioned by 10 participants), and activity measures, such as commit frequency and recent releases (mentioned by 10 participants) all reflected positively on a component. Measures that indicated an inactive project (mentioned by 5 participants), fewer contributors (mentioned by 4 participants), and violations of organizational policies (mentioned by 3 participants) could all reflect negatively on a component. Wermke et al. also specifically asked about security-related selection measures and found that measures indicating “a positive security history” (mentioned by 8 participants), and the presence of vulnerabilities or malicious code (mentioned by 3 participants) could contribute to the decision to include or exclude a particular component.

De la Mora and Nadi [66] performed a survey of 61 practitioners examining whether measure-based comparisons were useful for library selection and, if so, which measures were useful. They found measure-based comparisons were considered useful, with Performance, Popularity and Security as the most useful attributes measured. The authors also found that the importance of measures varied by domain. For example, Security measures were particularly important when examining libraries providing cryptographic algorithms.

Mujahid et al. [188] performed a mixed-methods analysis of the characteristics of “highly-selected” packages in the npm ecosystem. Packages were “highly selected” if they had a large number of dependent packages in npm. The authors examine 17 attributes derived from a wide range of academic literature on software engineering [61, 29, 117, 257, 119, 175, 32, 274, 274, 235, 1, 2, 3]. According to their survey respondents, the most important concepts were the *Documentation*, *Downloads*, *Github Stars*, and known *Vulnerabilities*. Conversely, the *Badges* that a repository may have and the number of repositories *Forks* on Github⁴ were not considered important.

Vargas et al. [157] use semi-structured interviews to develop a set of 26 attributes used when determining whether to use a particular third-party library. They then validated those concepts using a survey. The concepts were divided into three categories: Technical concepts, Human concepts, and Economic concepts. Technical concepts fell within three aspects: Functionality, Quality, and Release. The authors identified a fourth technical characteristic - whether the

⁴Github defines a Fork as “a new repository that shares code and visibility settings with the original ‘upstream’ repository.” [109]

system was a new development (“green field”) or an existing project (“brown field”). Human concepts could be separated into four aspects: concepts relating to project *stakeholders*, concepts surrounding the *organization* using the library, *individual* aspects such as developers’ self-perception of a project, and concepts of the *community* supporting the library. Economic concepts were divided into two aspects: total cost of ownership and risk. Vargas et al. noted that for several concepts, such as the *active maintenance* of a library (one of the attributes under the “Release” aspect of Technical concepts), developers may suggest multiple ways of measuring the concept. Vargas et al. [157] also note that selecting third-party libraries can be a two-stage process, in which (1) the *up-front* selection of one or more candidate libraries is followed by (2) a prototyping phase in which candidate libraries are evaluated using proof-of-concept code.

Nadi and Sakr [189] build on the Vargas et al. work, leveraging the same 26 concepts in a survey of data scientists to understand how the data science community selects third-party libraries and to determine how selection concepts may differ between communities. Nadi and Sakr also found concepts that were of particular importance within the data science community for the types of libraries they use, such as “statistical soundness”, which were not captured by Vargas et al. Neither Vargas et al. or Nadi and Sakr examine how concepts or the measurement methods used to derive the concepts should be aggregated across a library’s own dependency tree.

Xu et al [295] examine why practitioners may choose to use a library instead of implementing the functionality themselves or, inversely, why practitioners may choose to implement the functionality themselves instead of using a library. The authors performed targeted surveys of the Android and Python communities, which received 19 and 30 responses, respectively. They also performed an “open” survey of practitioners from a range of communities, which received 56 responses. Xu et al. found that *Efficiency*, *Reliability*, *Testability*, and *Maintainability* all seem to have been factors in choosing to use a library instead of implementing code. The authors did not explicitly ask about security but noted it was listed in some practitioners’ free-form responses explaining why they chose to use a third-party library. The authors also found that the most common reason for not using a library was that they were not aware of a third-party library that had the requisite functionality. A desire to *Reduce Dependency* was given as the primary reason to replace a library with code they wrote themselves, followed by *Flexibility*. Other factors that led practitioners to avoid third-party libraries included *Simplicity* and *Lack of Quality Libraries* for the desired functionality.

Abdalkareem et al. [1, 2] specifically examine the reasons why developers decide to use *trivial* packages which they define conceptually as “a package that contains code that a developer can easily code him/herself and hence, is not worth taking on an extra dependency for” [1]. The authors use Lines of Code (LoC) and complexity to determine when a package should

be considered “trivial”. In a survey of 88 package maintainers, they found that the majority (57.9%) do not consider using “trivial” or small packages to be a bad practice. Reasons for using trivial packages include the idea that they are *well implemented and tested*, and can *increase productivity* by reducing the time needed to implement the code. However, the authors found that contrary to popular belief, most trivial packages do NOT include automated tests [2]. The most common reason given by survey respondents to *avoid* using trivial packages was the additional maintenance and update overhead they introduced [2].

As noted by Nadi and Sakr [189], a practitioner is unlikely to use all 26 concepts that come from the Vargas et al. work alone [157], much less the combined set of all attributes that have been examined across the literature. However, as noted in nearly every work - there is no single attribute or measure that is “most important” for 100% of interview or survey participants. This diversity of responses reinforces the idea that tools may need to be flexible to accommodate different needs for different contexts [189, 66, 188].

5.3.2 Examining the relationship between package depth and measurement value

Paschenko et al. [219] perform an analysis of the Maven (Java) ecosystem, examining whether vulnerable dependencies are actually deployed in production. They found that only 37% of vulnerable dependencies are “direct dependencies”. However, if they grouped packages based on the organization responsible for them - as much as 82% of the vulnerable dependencies were potentially within the project’s control, i.e. direct dependencies of packages within the organization.

When analyzing dependencies, it may be useful to distinguish between dependencies that are used in production, and dependencies that are *not* used in production.

Latendresse et al. [160] perform an empirical analysis of the state-of-the-practice in the npm ecosystem. They examine attributes such as how many dependencies are *production* dependencies⁵, and which dependencies (production or non-production, direct or transitive) have more “npm-audit” security alerts. To enable the distinctions between production and non-production dependencies, the authors focus on 155 packages that use (1) the *webpack* or *rollup* module bundling tools; and (2) the “tree shaking” dependency pruning approach.

⁵for example, dependencies that are only used for testing as part of development and not included in a production build would NOT be considered *production* dependencies

Lantendresse et al. found that most dependencies (52,403 out of 53,405) were only for non-production builds. While the majority of these non-production dependencies were transitive dependencies (50,594 out of 52,403), only about half (178 out of 353) of the dependencies in production and run-time builds were transitive dependencies. The authors also found that most (308 out of 313) of the high- and critical-severity security alerts produced by the npm-audit tool occurred in development (not runtime) dependencies.

Work on other datasets has found higher numbers of transitive dependencies, such as Zimmerman et al. [314] whose analysis of 676,539 npm packages found an average of 80 dependencies, mostly due to transitive dependencies. It is unclear how much of the difference is due to how prior work controls for (or fails to control for) the difference between dependencies deployed to production and dependencies that are not deployed to production and whether other factors, such as whether the bundled modules from Latendresse et al. [160] may be smaller packages, plays a role. In either case, Latendresse et al.'s finding that non-production packages have higher numbers of transitive dependencies suggests that assessments of security risk in dependency trees should be careful to distinguish between production dependencies, which are more likely to be exposed to attackers and the majority of dependencies which are not exposed.

For vulnerability-based measures, the potential risks introduced by each vulnerability are perceived as constant, regardless of where the vulnerability occurs in the dependency tree. However, it is unclear if other measures of risk are also perceived as equally relevant for transitive as well as direct dependencies.

5.3.3 Prior work on OpenSSF Scorecard

The primary academic research on OpenSSF Scorecard to date are two studies by Zahan et al. [308, 307]. In one work [308], they examine whether OpenSSF Scorecard checks can predict vulnerabilities in the PyPi (Python) and NPM (javascript) ecosystems. They found that the Branch Protection, Code Review, Maintained, Pinned Dependencies, and Security Policy checks were good predictors for some machine learning algorithms. They also analyzed the relationship between different metrics, finding that the Token-Permissions and Pinned-Dependencies metrics had a particularly high correlation (0.84). We leverage some of their work in our own methodology, such as removing checks with a high percentage of missing or invalid scores from our data analysis.

In another work, Zahan et al. [307] examine the distribution of Scorecard scores, the efficacy of the Scorecard tool, and the adoption of the practices captured by the Scorecard metrics in

the NPM and PyPi ecosystems. Similar to our work in the Go ecosystem, they found that the Packaging and Signed Releases checks had a high number of “-1” scores, indicating that the Scorecard check did not successfully calculate the score. They also found that the Fuzzing check had a high percentage (82%) of scores that were 0. They found a slightly better distribution of scores for other checks. For example, the License, Dangerous Workflow, Pinned-Dependencies, and Token Permissions checks had fewer -1 scores (< 1%) of the packages) compared with our study. This may be due to differences in ecosystems or changes that have been made to the OpenSSF Scorecard tool since their study.

5.3.4 Prior work on the Go Ecosystem

While there has been substantial academic work around the development, verification, and structure of the Go programming language [53, 282, 233, 228, 251], as well as work around use of the Go programming language in computer science education [223], less work has been done on understanding the Go package ecosystem.

Some research has looked into finding and analyzing security-related defects in Go. Dunlap et al. [77] examined ways to use large-language models (LLMs) to identify fixes associated with Vulnerabilities through analysis of source code. The authors found that LLMs could be used to improve the precision over a naïve approach based on the commit links available with many security advisories. Wang et al. [283] developed a static analysis tool, go-sanitizer, for finding vulnerabilities in Go code. The authors summarize how CWEs may apply in Go, but consider the CWE list “not fully applicable to Golang”. They developed their own system of 9 bug types and developed a testing framework to identify these bugs. Our work does not focus on identifying and analyzing bugs and vulnerabilities. Instead, we look into other security metrics that can be used to understand the strengths and weaknesses of the software development process.

The main analysis of the Go ecosystem we are aware of is the analysis of vulnerability lifecycles by Hu et al [128]. Leveraging the same OpenInsights database we use in our work, the authors examine how long it takes for vulnerabilities to be patched in Go packages. The authors found that 67% of vulnerabilities had a patch available within a week but that most of the packages depend on vulnerable packages (71.87%). However, the authors consider all packages listed in the OpenInsights database without considering which packages may be intended for actual use and which are homework assignments, tutorials, or other packages not intended for actual use. We examine a narrower set of vulnerabilities in an effort to focus our work on packages that would be considered by a practitioner when determining what dependencies (third-party libraries) to use. Additionally, we are focused on a broader range of

security metrics.

A contribution of our work is expanding our empirical understanding of the state-of-the-practice in the Go package ecosystem.

5.4 Preliminary Interviews

We performed a set of semi-structured interviews to provide sufficient background for our research. In Section 5.4.1 we describe the interview process, while in Section 5.4.2 we describe the findings from the interviews that shaped our analysis. Our interview process follows our IRB protocol, number 26676, as we discuss in Section 5.10.

5.4.1 Interview Process

Each interview began with a series of background questions, including “What is your experience in working <in/with> security?” and “How are decisions made about whether to add a particular dependency to a project at your organization?” to help us interpret their responses. We then proceeded to the main part of the interview.

In each interview, we use an example package and its dependency tree, such as the one in Figure 5.1. Where possible, this tree was selected because it was a dependency of a piece of software the practitioner was familiar with. All technical questions were asked in the context of this dependency tree, and the scores for each of the packages of the dependency tree were shown, as can be seen in the example in Figure 5.1 which is for the Maintained check.

Maintained

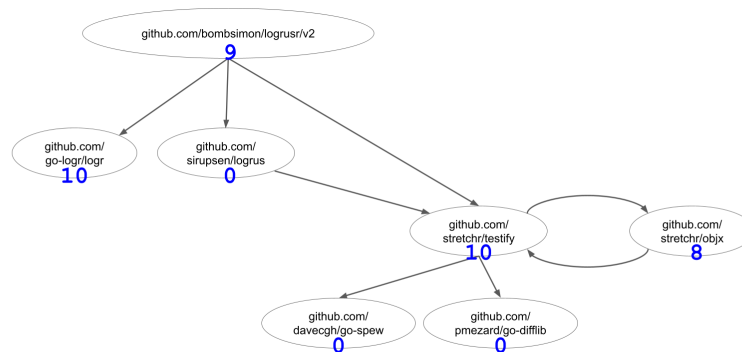


Figure 5.1: Example Dependency Tree from Interviews

To help orient the participants, we began by asking questions about the “Vulnerabilities” check. Vulnerabilities are commonly associated with security risk, and this check is already aggregated, allowing us to use this check to illustrate the type of aggregation we were referring to. We would provide hypothetical scores for the Vulnerabilities check for the package and its dependencies, and ask the following questions:

1. Are some of the scores more concerning than others? If so, why? What factors (e.g. size of the dependency, depth of the dependency) would be needed to make this determination?
2. How do you feel about this aggregation approach? If you are determining whether to include package A, should a different approach be used (e.g. should the total vulnerabilities be subtracted from 10)?

We would then go through as many of the other checks as we could within the one-hour timeframe. For each of the checks, we presented hypothetical scores for the check for all packages in the dependency tree and asked similar questions to those for the Vulnerabilities check:

1. Are some of the scores more concerning than others? If so, why? What factors (e.g., size of the dependency, depth of the dependency) would be needed to make this determination?
2. If you are determining whether to include package A, how should information from this entire dependency tree be aggregated? Should the scores be averaged? Should the minimum score be taken? Should the maximum score be taken? Should a different approach be used?

Participants: Overall, we interviewed seven participants, selected based on convenience. Of these participants, three had experience in Go and were therefore able to participate in our later survey, as we discuss in Section 5.6.2.3.

5.4.2 Findings

All of the following were discussed in at least three of the interviews and influenced our study:

1. *The aggregate score should not be higher than root*
2. It would be *good to know the “Worst Case” (i.e. minimum score) of the dependency tree*, but participants did not agree if it was appropriate on whether to use the Minimum score as a replacement for the existing value
3. Practitioners wanted to know *if there are any Binary-Artifacts in the package’s dependency tree*

4. *Code Review scores of nodes at lower levels of the dependency tree is less important* than scores of nodes closer to the root.
5. *Practitioners were more concerned about lower scores, and not concerned about higher scores*
6. It was *difficult to judge how the aggregate value would relate to risk for low-risk checks*, such as License
7. Particularly with the “maintained” check, we would expect to see less activity, and therefore lower scores, for a *smaller package deep within a dependency tree*.

The first four of these findings influenced our aggregation approaches, which we will discuss in Section 5.7. The last three observations influenced our overall methodology going forward with the data analysis and survey, as we will discuss in Section 5.6.

5.5 Data

The two primary components of our dataset are the Go packages and their dependency trees, which we describe in Section 5.5.1, and the Scorecard data, which we discuss in Section 5.5.2. For our Data Analysis, we also collected two ancillary datasets: the size of the repositories we examined, which we describe in Section 5.5.3, and vulnerabilities that have been found in Go packages, which we describe in Section 5.5.4.

5.5.1 Go Packages

To extract a list of Go packages and their associated dependency trees, we leverage the dataset provided by Open Source Insights⁶, which was previously used by Hu et al. [128] in their analysis of the Go ecosystem. At the time of our work, this dataset listed 768,115 Go packages, as shown in Table 5.2. We then map those packages to GitHub repositories using data stored in the Open Source Insights dataset, the package names themselves, which are often the URL of the source code, and the Go package index. Since we must map each package to a Scorecard score via a GitHub repo, we exclude all packages for which we could not locate a GitHub repo containing their source code. This resulted in the removal of 26,088 packages, with 742,027 packages remaining. Since we are primarily interested in packages that practitioners may decide to use, we narrowed our search to packages in the Open Source Insights dataset that had dependents (i.e., someone imported the package into their software), and the dependents

⁶<https://deps.dev/>

were NOT from the same organization. In other words, we selected packages that maintainers of another package from a different organization had decided to use. This further reduced the number of packages examined to 57,613.

One researcher randomly selected 100 packages that did not have any dependents and 50 packages that only had dependents from the same organization, while a second researcher randomly selected 200 packages of the packages that did not have an independent dependent. The first researcher found one packages that appeared to be a library in-use by practitioners. The remainder were not. For example, one of the more popular libraries the second researcher encountered was a “toy” library designed to mock kubernetes. We considered this margin of error sufficient, as the result was less than 1% of the packages examined and we did not have a less error-prone way to remove packages which were toy packages, tutorials, and other systems that would not be seriously considered when determining which third-party libraries to use.

We then remove packages for which we do not have dependency data⁷. The majority of the packages without dependency data (12,315 out of 13,385) are leaf nodes which have dependents, but do not have dependencies. The remaining 1,070 packages do not have full dependency tree information in the data collected from Open Source Insights. As we will discuss in the Limitations Section 5.9, resolving dependency trees is a challenging task. We selected the Open Source insights dependency tree data because it best reflected state-of-the-art best practices for dependency tree resolution, such as focusing on the production dependencies rather than development dependencies [219]. However, no dependency resolution tool or dependency tree dataset we are aware of is perfect. Consequently, we accept this limitation and remove the 1,070 packages as possible dependency root-nodes from our dataset, in addition to the 12,315 packages without dependencies. The resulting dataset contains 44,237 root nodes of dependency trees of third-party-libraries.

While we were able to map the package name of the root node for each of these 44,237 packages to a github repository, we also need to be able to map each of their dependencies to a github repository to have OpenSSF Scorecard data from the dependency tree to aggregate. We identified 34,617 packages for which we had a Github Repo name which met our previous criteria for the entire dependency tree. Next, we identified packages where the information between the packages was likely to be redundant, using the following

- First, it is a common practice in Go packages for major versions to get a new package name (for example ‘github.com/luzifer/rconfig’ and ‘github.com/luzifer/rconfig/v2’). These packages were consolidated to the latest version.

⁷Per Chapter 2, a package’s *dependent* is software that depends on the package. A package’s *dependency* is the software the package depends on. If package A depends on package B, then package A is package B’s dependent, and package B is package A’s dependency

- If the package name of one package contained the other, such as ‘github.com/olive-io/bpmn’ and ‘github.com/olive-io/bpmn/schema’, we selected the parent package., e.g. ‘github.com/olive-io/bpmn’.
- If the same package had multiple names, for example one under the github url and the other a different URL such as ‘github.com/gigawattio/testlib’ and ‘gigawatt.io/testlib’, we selected the non-github URL as the primary package name.

Removing redundant packages further reduced the count of packages to 30,730.

Finally, there were 5 packages for which we were unable get any OpenSSF Scorecard information since the original github repository was unavailable, i.e. we received a ‘404’ error when we attempted to access the site using a web browser. Removing these packages and the packages that depended on them further reduced the dataset to 30,717.

As we will discuss in Section 5.5.2.2, we used this initial dataset to identify which checks we had sufficient data for. In addition to the 5 checks which we were unable to get any Scorecard data for, some packages may have been missing data for a subset of the checks. An additional 2,766 packages were removed due to having invalid values for at least one of the six checks which met the criteria we will discuss in Section 5.5.2.2. The final dataset contained 27,951 packages as shown in Table 5.2.

Table 5.2: Go Dataset Summary

Total Number of Packages	768,115
Number of Packages where we can find a URL to a GitHub repo	742,027
Number of packages with at least one dependent from a <i>different</i> organization	57,613
Packages which have dependency tree data	44,237
Packages which have a Github repo for the entire dependency tree	34,617
Package Count after removing Redundant Packages	30,730
Packages with Valid Repository for Entire Dependency Tree	30,717
Repositories associated with the packages	30,545
Packages for which there were no missing or invalid scores for individual checks	27,951

5.5.2 OpenSSF Scorecard Data

Once we had our set of packages, we had to collect OpenSSF Scorecard Scores for each repository associated with the package. There were cases where multiple packages were associated with the same repository. Consequently, we analyzed scorecard data for **30,545 repositories**

associated with the 30,717 packages

Section 5.5.2.1 discusses how the data was collected, while Section 5.5.2.2 discusses which checks were included or excluded from our data analysis due to factors such as the availability of data which is based on the analysis by Zahan et al. [307, 308] with NPM and PyPI.

5.5.2.1 Source

To extract Scorecard scores, we first rely on the Google BigQuery dataset provided by OpenSSF Scorecard [208]. However, OpenSSF Scorecard only runs the checks on popular packages, and packages that have been specifically requested to be included. We extracted Scorecard data for 100,340 repositories (associated with 125,121 packages) from Google Bigquery. However, some of these packages did not have dependents or otherwise did not meet our inclusion criteria for the study, and some of the packages in our study were not sufficiently popular and/or had not been requested for inclusion in the original Scorecard dataset. Consequently, of the 30,717 root-level packages which met our non-Scorecard criteria, only 12,468 had Scorecard data in the original Biquery dataset. We ran Scorecard locally for any packages not included in the BigQuery dataset, using the same configuration as is used to populate the BigQuery dataset based on feedback from the OpenSSF Scorecard maintainers.

5.5.2.2 Checks Removed from the Analysis

Since the OpenSSF Scorecard’s vulnerability check is already aggregated, we only used it in our initial interviews and did not include it in any further analysis.

The CI-Tests, Contributors, and Dependency-Update-Tool metrics are not included in the dataset due to the volume of resources needed to run the checks, which would prevent the Google BigQuery dataset from being updated periodically. The metric configuration was confirmed by the OpenSSF team who indicated the location in the project’s source code where this is set ⁸. Consequently, these metrics were removed from our study. Our discussion with members of the OpenSSF Scorecard team also led us to remove the CII-Best-Practices-Badge check from our analysis. While adoption of the Best-Practices-Badge program has been growing, it is still only used by a very small number of repositories

When the Scorecard tool cannot produce a result for any reason other than being excluded from the analysis, the score is marked as “-1”. Table 5.3 shows the number and percentage of repositories (out of the 30,545 repositories associated with our main packages). Following the methodology of Zahan et al. [308], we focus our data analysis on checks for which less than 10% of the data is either missing or invalid. For the remaining checks, we remove packages

⁸<https://github.com/ossf/scorecard/blob/76a04bfe40c25746b2633127270a9219a98a37ad/cron/config/config.yaml#L48>

where the data is missing or invalid. As a result, 27,951 packages remained in the analysis.

Finally, as described in Section 5.4, when performing the interviews, we found that determining whether aggregating information from low-risk checks such as License would be a better indicator of risks was difficult and generated a different discussion from higher risk checks. Consequently, similar to our previous work with vulnerability detection tools in Chapter 3, we remove low-risk checks from our analysis. However, all low-risk checks either had insufficient data or were not included in the dataset at all. Hence, these checks would have been removed for other reasons.

Table 5.3: OpenSSF Checks Data Summary

Red text indicates why a check was excluded

Name	In OpenInsights	Missing or invalid	Risk
<i>GROUP: Holistic Security Practices - Maintenance</i>			
Dependency Update Tool	No	N/A	High
Maintained	Yes	1 (< 0.1%)	High
Security Policy	Yes	0 (0%)	Med.
License	Yes	3,851 (12.6%)	Low
Best Practices ^b	Yes	N/A	Low
<i>GROUP: Holistic Security Practices - Continuous Testing (Test)</i>			
CI Tests	No	N/A	Low
Fuzzing	Yes	9 (< 0.1%)	Medium
SAST	Yes	3,963 (13.0%)	Medium
<i>GROUP: Source Risk Assessment</i>			
Binary Artifacts	Yes	0 (0%)	High
Branch Protection	Yes	2,656 (8.7%)	High
Dangerous Workflow	Yes	22,257 (72.8%)	Critical
Code Review	Yes	71 (< 0.1%)	High
Contributors	No	N/A	Low
<i>GROUP: Build Risk Assessment</i>			
Pinned Dependencies	Yes	16,598 (54.3%)	Medium
Token Permissions	Yes	18,408 (60.3%)	High
Packaging	Yes	29,881 (97.8%)	Medium
Signed Releases	Yes	27,317 (89.4%)	High

^aThe Best Practices check was removed from our analysis due to the low number of participating organizations at the time of this research.

5.5.3 Size Data

As discussed in Section 5.4, our initial interviews suggested that for our analysis examining the relationship between scores and depth in RQ3.1.1, we should control for package size. As a basic size measure, we use the size of the package's repository, extracted via the GitHub API [110], which is the size of the github repository in Kilobytes.

5.5.4 Vulnerability Data

In RQ3.1.3, we examine whether the aggregation approaches alter the predictive relationship between the scores and vulnerability count. This analysis uses vulnerability data from the Open Source Vulnerabilities Database (OSV) [210]. We consider each unique ID in the OSV data to be a unique vulnerability. OSV is readily available and covers a greater number of vulnerabilities than GitHub Security Advisories.

5.6 Methodology

Our methodology is broken into two segments. In Section 5.6.1, we discuss the large-scale data analysis done in our study, covering RQ3.1a, RQ3.1b, and RQ3.1c. In Section 5.6.2, we discuss the methodology of our survey, covering RQ3.2.

5.6.1 Data Analysis

In this section, we describe the statistical analyses used to analyze the dataset collected as described in Section 5.5 to answer RQ3.1a Depth: *What is the relationship between scorecard scores and the average depth of a package in a dependency tree?*, RQ3.1b Difference: *What is the difference between the original score and the aggregate score?*, and RQ3.1c Vulnerabilities: *How does including aggregate information from a package's dependency tree change the relationship between security metrics and vulnerabilities?*

5.6.1.1 RQ3.1a: Depth

Given that practitioners were more concerned about lower-scoring packages as discussed in Section 5.4.2, we wanted to know if lower-scoring packages occurred more often deeper or higher in the dependency tree, or if lower scores were evenly distributed throughout a package's dependency tree. To understand the relationship between the depth at which a package occurs in a dependency tree and its Scorecard score, we first found the average depth for each package examined in this analysis. To understand this relationship while controlling

for size and avoiding “p-hacking”⁹, we used linear regression [78]. We used the default linear model in R, treating each of the checks and the control variable (size) as independent variables, with the package’s average depth as the dependent variable. Due to the size of our dataset, we selected $p < 0.01$ as our threshold for statistical significance.

5.6.1.2 RQ3.1b: Difference

To assess the impact of our changes, we assessed the frequency with which scores change. We assess whether the difference in the original and aggregated scores is statistically significant by applying the Mann-Whitney U test for each check, using the Bonferroni correction to mitigate the multiple comparisons problem. However, changing scores will, by definition, change the scores. More important is the question of whether those changes substantially impact the overall population. Consequently, we calculated Cohen’s d to understand the difference in the means between the original scores and the aggregate scores relative to the scores’ standard deviation [47]. We also provide the interpretation of Cohen’s d for effect size for each check.

5.6.1.3 RQ3.1c: Vulnerabilities

For RQ3.1c, we looked at the relationship between the Scorecard scores, our Aggregated Scorecard Scores, and the number of Vulnerabilities in a package. For the purpose of this analysis, we consider any vulnerability in the current version of the package itself or in any of its dependencies to be included in the package’s vulnerability count, which is how the OpenSSF Scorecard itself computes vulnerabilities. We use the number of vulnerabilities as the dependent variable in linear regression to determine the relationship between vulnerabilities, the checks, and the aggregation methods. In this analysis, we standardized the number of vulnerabilities and dealt with outliers using the same approach as the OpenSSF Scorecard vulnerabilities metric [208], capping the number of vulnerabilities at 10, i.e., for the two packages with more than ten vulnerabilities, the dependent variable was 10. 27,889 of the 27,951 packages did not have any vulnerabilities in the entire dependency tree. We modeled the independent variables as individual checks, using interaction [78] to analyze the difference between the original values and the aggregate scores. Due to the size of our dataset, we selected $p < 0.01$ as our threshold for statistical significance.

⁹“p-hacking” refers to practices whereby practitioners collect data or run tests until statistical significance is likely to be achieved by chance, even in non-significant results[121]

5.6.2 Survey

In this section, we describe the methodology for developing and deploying our survey to address RQ3.2 Vulnerabilities: *How does including aggregate information from a package's dependency tree change the relationship between security metrics and vulnerabilities?*. In Section 5.6.2.1, we discuss the design of the survey. In Section 5.6.2.2, we discuss survey deployment. In Section 5.6.2.3, we describe our process for recruiting survey participants.

5.6.2.1 Survey Design

When starting the survey, we identified a potential aggregation approach for each metric we are examining. We then constructed the survey such that for each metric, survey participants were asked whether they would prefer (a) the original score; (b) an aggregate score; or (c) another approach. If participants selected the third option, they were asked to fill in a free-response description of what approach they preferred. We also asked participants why they preferred the approach they selected (a, b, or c). We asked survey respondents to consider these questions in the context of determining whether to include or replace a package and provided a specific example package for reference, along with example scores for that package based on the example. The full survey can be found in Appendix E.

We identified five packages to use as example dependencies to base our survey questions on. Our selection of example packages was based on (1) the package's popularity (i.e. the number of dependents); (2) whether the package had a dependency tree to analyze; (3) was not associated with a well-known organization like Microsoft or RedHat to avoid introducing potential bias due familiarity and potential trust of the organization. We also ensured that at least one of the packages selected was a dependency of the projects developed by our interview participants, to facilitate their potential participation in the survey.

Similar to the interviews, the survey respondents were presented with the dependency tree of the example dependency to assist with understanding the aggregation approaches, such as the minimum of the root and the direct dependencies of the package. However, in the survey we did not provide the scores for each element of the dependency tree since the intent was to have a single score to present to practitioners.

At the end of the survey, we asked participants a series of questions to gain basic background information. First, we asked them "How many years have you worked in software development?" with a free-response answer. We then asked them "How would you rate your level of cybersecurity and/or software security expertise? (On a scale from 0 to 4)", with a multiple-choice answer. We then asked respondents "How often have you contributed to open-source software packages?" and "How often have you used open-source software packages?". For the last two

questions, respondents were presented with the multiple-choice options “often”, “sometimes”, and “never”.

5.6.2.2 Deployment

The survey was developed in two iterations. An initial pilot survey was distributed within the research lab. We then refined and clarified questions before deploying the survey. The survey was built and deployed using the Qualtrics tool [236], facilitating the complex, built-in survey logic for presenting different dependencies to different respondents. In the final deployment of the survey, the survey was open for 12 days from July 9 through July 21, 2024. Participants were initially told that the deadline was July 17, which was extended to the 19th and then to the 21st to accommodate some participants. Our surveys follow our IRB protocol, number 26676, as we discuss in Section 5.10.

5.6.2.3 Recruiting

To recruit participants, we used our dependency tree data to identify dependents of the example packages. Additionally, the website for one of our example packages included a list of other open-source projects that used the package. We then contacted the maintainers of dependent packages. We excluded dependents that were a fork of another package since they may not have made the decision to include the example dependency. Similarly, we excluded packages that included the dependency package’s name in the name of their own package to minimize cases where the use of the example dependency was integral to the system. We also did not attempt to recruit maintainers of packages that had already been archived. We searched for the organization or owner of the package’s repository and the name of the package itself, using Google search engine, to try to find contact information for maintainers of the repository. We then used that contact information to try to recruit survey participants. In compliance with our Institutional Review Board Protocol, we attempted to confirm that the package owners were NOT primarily based outside the US or Australia before contacting them. We also contacted interview participants who were familiar with Go to request their participation in the survey. To further incentivize participation and thank participants, participants are entered into a drawing for a \$20 Amazon gift card.

5.7 Aggregation Methods

Based on the existing aggregation approach for the Vulnerabilities check, our preliminary interviews, and the mechanics of each check (e.g., the fact that the Fuzzing check only has two

values, 0 and 10), we developed aggregation methods for each of the six checks for which met our criteria for the analysis.

Binary Artifacts Similar to the Vulnerabilities check, for the Binary Artifacts check we aggregated the score from throughout the package’s dependency tree by subtracting 1 from the score for each Binary Artifact in the entire dependency tree, rather than only deducting points for the root.

Branch Protection For Branch Protection, we aggregated scores by taking the minimum value (i.e. “worst case”) for the entire dependency tree

Code Review Following the feedback in the interviews that lower-level dependencies were less important when considering the Code Review check, we aggregated the Code Review check by determining the minimum value of the root node and its direct dependencies

Fuzzing Our approach to aggregating the Fuzzing metric is based on that for Binary Artifacts and Vulnerabilities. Since Fuzzing is a binary score (always 0 or 10), we subtract the number of dependencies that do *not* use fuzzing from the root score. If this aggregation results in a score below 0, then the score is 0.

Maintained We aggregate the Maintained check in the same way as the Branch Protection check. We used the Minimum score of the entire dependency tree of the package, including the root package itself.

Security Policy We aggregate the Security Policy check in the same way as the Maintained and Branch Protection checks. We used the Minimum score of the entire dependency tree of the package, including the root package itself.

For the purpose of our data analysis, if a dependency of one of the packages had a “-1” score, indicating that the OpenSSF Scorecard tool did not work for that check on that dependency, the -1 value was *not* included in the aggregation for the root package. However, we still include the root package in our analysis to try to avoid further decreasing our sample size.

5.8 Results

In Section 5.8.1, we discuss the results of our data analysis for RQ3.1a, RQ3.1b, and RQ3.1c. In Section 5.8.2, we discuss the results of our survey.

5.8.1 RQ3.1 - Data Analysis

In this section, we describe the results of our data analysis. In Section 5.8.1.1, we examine the relationship between the original OpenSSF Scorecard scores and the depth a package occurs in a dependency tree to answer RQ3.1a Depth: *What is the relationship between scorecard scores and the average depth of a package in a dependency tree?* In Section 5.8.1.2 we look at how much of a difference our aggregation approaches make, addressing RQ3.1b Difference: *What is the difference between the original score and the aggregate score?* Section 5.8.1.3 addresses RQ3.1c Vulnerabilities: *How does including aggregate information from a package's dependency tree change the relationship between security metrics and vulnerabilities?;* examining the relationship between the Scorecard checks and vulnerability count, as well as how much our aggregation approaches affect that relationship.

5.8.1.1 Depth

Table 5.4 shows the results of our analysis of the relationship between a package's original (non-aggregated) Scorecard check scores and the average depth the package occurs in the dependency trees of other packages. As we can see in the table, Branch-Protection has a statistically significant negative relationship with depth (e.g., lower scores for packages lower in the dependency tree). Hence for Branch-Protection there is more likely to be information deeper in the dependency tree that would be of concern to practitioners (lower scores). The Code Review and Fuzzing checks, on the other hand, have statistically significant, positive relationships with depth (i.e. higher scores tend to occur in packages that are lower in the dependency tree), suggesting we may see fewer cases where critical security issues are buried in the dependency tree. There is no statistically significant relationship between the Binary Artifacts, Maintained, or Security Policy metrics and the averaged depth a package occurs in a dependency tree.

5.8.1.2 Difference between Original and Aggregate Scores

See Table 5.5. One of the questions that we can statistically evaluate is how our aggregation approaches make a difference in the scores. Specifically, *RQ3.1b: What is the effect of aggregating scores from throughout the dependency tree?*

First, we look at the number of packages where the score differs, which is the second column of Table 5.5. Since the score can never be higher than the root score, any differences in score between the original and aggregate scores will be decreases in score.

Second, we look at whether the difference between the original and aggregate scores is statistically significant at $p < 0.01$ using the Mann-Whitney U-test with Bonferroni correction. We

Table 5.4: Relationship of Metrics to Average Depth

Metric	Coefficient	Std. Error	p-value
Binary-Artifacts	0.008	0.005	0.099
Branch-Protection	-0.034	0.002	< 0.001**
Code-Review	0.041	0.001	< 0.001**
Fuzzing	0.042	0.003	< 0.001**
Maintained	0.001	0.001	0.5624
Security-Policy	-0.001	0.002	0.374
size (control)	< 0.001	< 0.001	0.570

**Statistically significant at $p < 0.01$

then leverage Cohen's d , a statistic commonly used for effect size, to understand the difference in average scores between the original and aggregate scores. We include the Cohen's d value in the fourth column of Table 5.5 alongside the interpretation of Cohen's d statistic [47].

Table 5.5: Effect of Adjusted Scores

	# Packages whose Score Decreases	p-value	Cohen's d (95% confidence interval)	Interpretation
Binary Artifacts	999	< $2.2e - 16^{**}$	0.140 ± 0.017	Negligible
Branch Protection	2885	< $2.2e - 16^{**}$	0.406 ± 0.017	Small
Code Review	4775	< $2.2e - 16^{**}$	0.323 ± 0.017	Small
Fuzzing	201	0.06	0.042 ± 0.017	Negligible
Maintained	2688	< $2.2e - 16^{**}$	0.364 ± 0.017	Small
Security Policy	1106	< $2.2e - 16^{**}$	0.270 ± 0.017	Small

**Statistically significant at $p < 0.01$ with Bonferroni correction

The difference between the original and aggregate values was statistically significant for all checks except the Fuzzing check. This may be due to the relative scarcity of packages using Fuzzing tools that are picked up by this check. 39,008 out of the 39,314 packages have a 0 score for Fuzzing, meaning that only 306 of the Go repositories used the fuzzing tools identified by the OpenSSF Scorecard Fuzzing check.

The effect size of this difference was negligible for the Binary Artifacts and Security Policy checks but higher for the Branch Protection, Code Review, and Maintained checks. In the results for RQ3.1c (Section 5.8.1.3) and in our Survey Results for RQ3.2 (Section 5.8.2), we will

further explore the potential impacts of this difference.

5.8.1.3 Vulnerabilities

Table 5.6 shows the results of our regression analysis, analyzing the relationship between checks and vulnerabilities, and how our aggregation approach influences these checks. The first six rows show the relationship between each check and vulnerabilities. Row 7 shows the cumulative effect of aggregating the scores. In other words Row 7 addresses the question “Does aggregation alter the relationship between vulnerabilities and checks?” without discriminating between the different checks. Rows 8 through 13 show the extent to which aggregating the score alters the check’s relationship with vulnerability count for each check.

Table 5.6: Linear Regression of Checks and Adjusted Values against Active Vulnerability Count

	Coefficient	Std. Error	t Value	p-value
Binary Artifacts	< 0.001	0.001	0.439	0.661
Branch Protection	-0.003	0.044	-7.202	< 0.001**
Code Review	0.003	< 0.001	8.652	< 0.001**
Fuzzing	0.013	0.001	10.977	< 0.001**
Maintained	0.004	< 0.001	9.456	< 0.001**
Security Policy	0.003	< 0.001	6.950	< 0.001**
Aggregate checks	0.007	0.016	0.433	0.665
Binary Artifacts : aggregate	-0.0004	0.001	-0.247	0.805
Branch Protection : aggregate	-0.001	0.003	-0.155	0.877
Code Review : aggregate	0.003	0.001	3.669	< 0.001**
Fuzzing : aggregate	-0.001	0.002	-0.339	0.735
Maintained : aggregate	0.003	0.002	1.546	0.122
Security Policy : aggregate	-0.006	0.003	-2.235	0.026

***Statistically significant at $p < 0.01$*

As can be seen in the table, the Binary Artifacts check does not have a statistically significant relationship with vulnerability count, and aggregating the check does not alter the relationship in a statistically significant way.

The Fuzzing, Maintained, Security Policy, and Code Review checks all have statistically significant, *positive* relationships with vulnerability count (higher check scores are related to higher vulnerability counts). This may seem counter-intuitive if we consider vulnerability

count as an indicator of risk. However, this positive relationship may be explained by the nature of the checks themselves. If someone is using a fuzzer, has a security policy for how to report vulnerabilities, uses code review, and generally keeps their package maintained and up-to-date, we would expect them to occasionally find vulnerabilities that need to be mitigated. If a maintainer is not following these practices, they are less likely to find vulnerabilities that need to be mitigated.

Only Branch Protection has a statistically significant, *negative* relationship with vulnerability count (a.k.a., lower Branch Protection is related to more vulnerabilities). If our previous explanation is correct, this difference may be partly explained by the fact that Branch Protection is less directly linked to finding vulnerabilities. However, more research is needed to determine if this assumption is correct.

Our aggregation methods only resulted in a statistically significant change to the relationship with vulnerability count for the Code review check. In this case, our aggregation method is related to a further increase in the relationship between the Code Review score and vulnerability count.

5.8.2 RQ3.1 - Survey

Overall, we reviewed over 900 packages as part of our recruiting effort and contacted one or more maintainers at 22 organizations. Eight (8) individuals filled out the survey. Since each individual was randomly presented with a subset of the checks, the total responses for each check are lower than 8. Consequently, we include Total Responses for each check in the third column of Table 5.7.

Due to an error on one branch of the survey logic, one participants could not fill out the demographic information in the survey, and did not reply to our follow-up request. For the 7 participants who did fill out the demographic information, the average years of experience was *22.6 years*. The average security expertise on a scale from 0 to 4 was 3. All participants had experience using and maintaining open-source software.

The primary results of our survey are described in Section 5.8.2.1. In Section 5.8.2.2 we discuss the reasons practitioners gave for *why* they selected a particular choice. This section focuses on the six checks for which we had sufficient data. However, since the survey was deployed prior to the completion of data analysis, we examined three additional checks which we initially thought would have sufficient data based on the work by Zahan et al. [307, 308] and our discussions with OpenSSF Scorecard. The results for these three additional checks are in Appendix F.

5.8.2.1 Survey Results

Table 5.7 shows the survey results for the six checks examined in our study. For the checks with red text, practitioners preferred the original (unaggregated) score. For checks with dark grey text, practitioners preferred our aggregate score or an alternate aggregate score.

The first column of Table 5.7 lists the name of the metric, while the second column summarizes our aggregation approach, which the practitioners were asked about. The third column indicates the total number of respondents who were presented with the check. Each respondent was presented with a randomly selected subset of the checks, and so this total varies between checks. The fourth column indicates the number of respondents who indicated they preferred the original score. The fifth column indicates how many respondents prefer our aggregate score. The sixth column indicates how many participants indicated they would prefer some “other” score. Where practitioners preferred some “other” score, we summarize the scoring approach they indicated they *would* prefer.

5.8.2.2 Reasons Practitioners gave for their answers

The second question for each check asked practitioners *why* they had made the previous selection. We summarize their answers in the following subsections.

5.8.2.2.1. *General*

At least three participants indicated that the aggregate values did not place sufficient emphasis on the score for the root node. Similarly, at least three individuals indicated it would be nice to have both scores for one or more of the checks. The checks for which one or more participants indicated “both” scores would be useful were the Binary Artifacts, Code Review, Fuzzing, and Security Policy checks.

5.8.2.2.2. *Checks where Participants Preferred Aggregate Scores*

The reasons why practitioners made the selection they did for the Binary Artifacts check were mixed with no clear patterns or themes. There seemed to be confusion both over the question and over whether the presence of Binary Artifacts deep within the dependency trees of Go packages would have any effect at the root, with one respondent claiming that it would and another respondent claiming it would not.

For the Branch Protection check, practitioners’ responses on why they made the selection they did had no themes other than the broad indicator that, in the words of one respondent, they would “want to know the health/security of dependent projects”.

Table 5.7: Survey Results

Metric	Aggregation Approach	Total	Prefer Original	Prefer Aggregate	Other	(Explanation)
Binary Artifacts	Subtract 1 for each Binary Artifact in Dependency Tree (including in Root)	7	2	4	1	Neither (Check Irrelevant)
Branch Protection	Min. of Entire Tree	5	1	3	1	Check Irrelevant
Code Review	Min. of Direct Dependencies	7	4	1	2	1) Neither (Check Irrelevant) 2) Weighted Average (root contributes half the overall value of the score)
Fuzzing	Subtract 1 from Root Score for Each Dependency without Fuzzing	5	2	2	1	Min. of Entire Tree
Maintained	Min. of Entire Tree	7	3	2	2	1) Neither (Check Irrelevant) 2) Both Preferred
Security Policy	Min. of Entire Tree	6	3	1	2	1) Neither (Check Irrelevant) 2) strong preference to repositories that have a security policy

For the Fuzzing check, there were no consistent patterns across participant responses. However, one participant who agreed with the aggregated score indicated that they appreciated how the aggregation approach for Fuzzing provided “info about the dependencies’ behaviors” while “ focusing on the root”, echoing the sentiment of another participant who did not agree with the aggregate score because it did not sufficiently focus on the root score.

5.8.2.2.3. Checks where Participants Preferred the Original Score

Participants provided similar responses for their selections for the Code Review and Security Policy checks. For both Code Review and Security Policy, respondents who indicated they preferred the original score used statements such as “I usually just want to know the score of the top-level package”, suggesting that *no* aggregation approach would be appropriate for these checks. Additionally, for Security Policy, two participants indicated the low adoption rate of Security.MD files and security policies more broadly influenced their response.

For the Maintained Check, two individuals who did not select the Aggregate score raised concerns that smaller packages may not have recent activity that would be detectable with a static analysis tool such as Scorecard. The remainder of their answers differed, with no clear themes. Another individual indicated that as long as the root package was maintained, they should be able to handle any problems in their dependencies.

5.9 Limitations

We discuss the limitations of our approach in this Section. We group these limitations as threats to Conclusion Validity, External Validity, Internal Validity, and Construct Validity [49, 91, 289].

5.9.1 Conclusion Validity

Conclusion Validity is about whether conclusions are based on statistical evidence [49, 289]. For our Data Analysis, we have sufficient information to draw statistically significant conclusions for many outcomes. However, for our survey, we did not have sufficient participation to perform statistical analysis. This threat is slightly mitigated by the level of expertise of our participants, who, on average, have over 20 years of experience, as well as experience and expertise with open-source software and with security.

Another threat to conclusion validity is the low number of vulnerabilities in our dataset. Only 62 packages were associated with one or more vulnerabilities. Among other problems, this limited the statistical analysis we were able to do. For example, there was insufficient data to use a poisson model instead of a linear model, even though poisson models are generally preferred

for “count” data such as vulnerability counts [78]. Additionally, the coefficients of our model were all very small (< 0.05) However, this is a common problem in ecosystem studies [308, 82], and we were able to have some statistically significant results due to the large size of the overall dataset.

5.9.2 Construct Validity

Construct Validity concerns the extent to which the treatments and outcome measures used in the study reflect the higher level constructs we wish to examine [49, 289, 241]. Although we initially intended to use Vulnerability Count as a metric for risk, our results suggest the assumption that vulnerability count was a proxy for risk was unfounded. However, we can draw some conclusions without assuming that vulnerability count can be used as a proxy for risk. We can assume the number of vulnerabilities *found* in a package, which requires fewer assumptions. This is different from the *actual* number of vulnerabilities in a package, as we noted in Chapter 3 where we found that public vulnerability lists often contain only a fraction of the vulnerabilities in software.

Similarly, it is possible that the OpenSSF metrics do not adequately capture the concepts which they claim to represent. However, the tools are supported by several companies and industry organizations, and we are not aware of a more widely accepted alternative.

Additionally, our analysis of dependency tree information is reliant on the accuracy of our dependency trees. Resolving a package’s dependency tree is a non-trivial problem. We selected the Open Source Insights dataset [115] since the dependency trees in the Open Source Insights database only include production dependencies, as recommended by Paschenko et al. [219]. However, there are packages that the tools used to develop the Open Source Insights database are unable to resolve, which do not appear in their database and, therefore, in our data. The maintainers of the Open Source Insights database claim to have tested their algorithms against “native” dependency tree graph resolution implementation for each language and “given identical inputs the results agree closely: 99% or higher, often much higher” [116].

5.9.3 Internal Validity

Internal Validity concerns whether the observed outcomes are due to the treatment applied, or whether other factors may have influenced the outcome [49, 91]. The checks examined are not a comprehensive set of all possible characteristics of vulnerabilities. Hence, we limit the conclusions we draw based on linear regression to comparisons of the checks that we analyze rather than the output of the model.

It is possible that the set of packages we selected may have biased our outcome. To mitigate

this, we set our selection criteria for packages to focus our search on packages that might be selected as third-party libraries. Additionally, we only limit our selection to packages that have been imported by at least one other package rather than setting a much higher threshold, so that we are selecting more than the “most popular” or “most trusted” Go packages.

It is also possible that our criteria that a package must have a GitHub repo and data for all its dependencies may have affected our dataset such that larger packages with more dependencies are under-represented. However, the alternative of using packages for which we have incomplete data poses other potential reliability problems. Future research may examine if it is possible to get reliable aggregate scores using data from dependency trees for which we do not have Scorecard information from all dependencies.

5.9.4 External Validity

External Validity concerns the generalizability of our results [49, 91, 289]. Our results may not generalize to other ecosystems. However, the Go language is used in important applications such as payment applications for companies such as American Express and Capital One [111]. Go has many characteristics similar to those of more popular languages, such as type-safety and memory-safety.

5.10 Ethics

The preliminary interviews and survey were performed following North Carolina State University IRB protocol 26676. This protocol was exempt from full board review since we are not collecting or reporting sensitive information. We obtained informed consent from all participants, and structure our interview and survey to minimize the amount of personal information we collected, and anonymized all responses.

We are not identifying new security weaknesses or vulnerabilities in software. We therefore do not have a vulnerability disclosure process, another common concern in security research.

5.11 Discussion

We begin this section by returning to our original thesis - should Scorecard checks use data aggregated from throughout the dependency tree? As discussed in Section 5.11.1, the answer is “it depends”. In Section 5.11.2, we discuss potential improvements in our aggregation approaches and other factors to be examined in Future Work. Finally, in Section 5.11.3, we discuss additional implications of our findings beyond our initial research goals.

5.11.1 Should Metrics aggregate information from throughout the package's dependency tree?

Although our survey results were mixed, for all checks, at least one respondent preferred the aggregated metrics, and in three cases, Binary Artifacts, Branch Protection, and Fuzzing, more practitioners preferred an aggregated score than the original score.

Practitioner preferences vary by individual respondents and by the metric examined.

While some checks should be aggregated, aggregating the Code Review check may not be worth any additional analysis aggregation could incur. Based on the analysis of the relationship between checks and depth in RQ3.1a, the lower values of the Code Review check tend to occur nearer the top of the dependency tree. The aggregate value only increases the relationship with vulnerability count, which does not necessarily add value. Even the more limited aggregation approach we selected, taking the Minimum score of the root and its direct dependencies rather than the Minimum of the entire tree, was rejected by six of the seven survey respondents who were presented with this check.

For Branch Protection, in particular, there are multiple arguments for aggregating the score, including the fact that a greater number of lower scores may be hidden in the depths of the dependency tree.

On the other hand, survey participants preferred aggregate scores for branch protection and binary artifacts information. Branch Protection, in particular, had a negative relationship with package depth, suggesting more packages with low Branch Protection scores may be hiding deeper in a package's dependency tree.

Aggregating the Code Review metric is least helpful.

5.11.2 Improvements and Future Work

Our aggregation approaches can be improved. In particular, our aggregation approach for the Fuzzing check had no statistical difference from the original check. However, practitioners commented on how they preferred the additional weight placed on the root node in the Fuzzing check, suggesting that taking the absolute minimum of the package's dependency tree, as suggested by one survey participant, may not be universally desirable. In future work, we should consider additional approaches which place more emphasis on the root score, particularly for the Binary Artifacts, Branch Protection, and Fuzzing checks where the majority

of respondents thought the checks should be aggregated. We should compare these approaches against our approaches. For the Fuzzing check, we may also want to consider including the minimum of the dependency tree as a potential aggregation method.

Another approach to consider for future work may be to present the aggregate information alongside the current score, as suggested by multiple survey participants. However, usability studies should be done to understand if this results in too much information being presented to the end-user [30, 88] or if it is a better alternative.

Understanding the usability of presenting both the aggregate and original scores is a potential avenue of future work

5.11.3 Other Implications

Finally, similar to Zahan et al [308], we found that checks reflecting practices associated with vulnerability detection, such as Fuzzing, had a positive relationship with the number of vulnerabilities in software. If we assume that higher values mean lower risk and that vulnerability count is a measure of risk, then these findings are counterintuitive. However, organizations with robust practices around detecting and reporting vulnerabilities are more likely to find vulnerabilities in their system. As we mentioned in Section 5.3, the Go ecosystem has often been overlooked in security research, eliminating one source of external reports. Hence, one of the simpler possible explanations for the positive relationship between Fuzzing, Code Review, Security Policy, and Maintained metrics and vulnerability count could be that if you are looking for vulnerabilities, you are more likely to find them. More research is needed to fully confirm or refute this assumption, but it would align with the findings of our prior work, particularly our vulnerability detection work in Chapter 3.

Organizations using practices that are associated with vulnerability detection find more vulnerabilities

5.12 Conclusion

Using a combination of data analysis and feedback from practitioners via interviews and surveys, we have determined that some metrics, such as the Branch Protection metric, may be more helpful to practitioners if information from throughout the dependency tree is provided. Others, such as the Code Review metric, may be sufficient if only the root package is evaluated. Further research is needed to fully understand how the information should be presented, such

as whether presenting both initial and aggregate information provides too much information.

CHAPTER

6

SUMMARY

We begin our summary with lessons learned from our studies, before discussing potential areas of future work and providing an overall conclusion

6.1 Lessons Learned

One of the main lessons learned is that practitioner preferences often cannot be explained by focusing on the statistically best-performing tool. For example, although XSS vulnerabilities were of very high concern and SAST tools were not effective at finding XSS vulnerabilities, the first vulnerability detection technique OpenMRS implemented after our investigation was a SAST tool since it worked well with their development process. Similarly, in our final study - practitioner preferences did not align with the statistical performance of each aggregation approach.

One of the key anecdotal findings from both the Vulnerability Detection and Metric Aggregation studies is the extent to which practitioners consider the difficulty of mitigating potential risks. As we mentioned, OpenMRS focused on SAST tools due to their ease of implementation. In our interviews with practitioners about different OpenSSF Scorecard checks, one practitioner commented that part of the challenge with expecting improvements to the Code Review and Maintained checks, was that coming up with more maintainers or code reviewers to implement

these practices was non-trivial in smaller organizations.

6.2 Future Work

One of the potential areas of future work builds on the lessons learned by further examining the ease or difficulty of remediating different security risks. Remediation may involve applying best practices or fixing vulnerabilities, depending on the security assessments being analyzed. However, concerns about remediation remain a common thread and should be analyzed further.

Another area of future work is examining security assessments in other domains, such as Infrastructure as Code (IaC) scripts. Much of the research on security assessments in IaC has focused on state-of-the-art tools developed in academia, with less attention paid to the tools actually used by developers, such as the Ansible-Lint static analysis tool. Similarly, we can expand our Scorecard work by examining other ecosystems and, potentially, other tools.

Finally, more work can be spent examining how to present aggregate information from the dependency trees of third-party libraries. Multiple participants indicated that they thought they would benefit from having “both” the original score and the aggregate score, even if they preferred one over the other. However, given that there are already a large number of checks in the OpenSSF scorecard, if we were to incorporate aggregation for all checks and double the amount of information provided, we may run into concerns about providing more information than an individual can easily process. Consequently, more work can be done to understand better the nuances of how aggregate information can best be presented to practitioners.

6.3 Conclusion

Combining information from multiple software assessment methods and from throughout a project’s dependency tree may provide more information and be preferred in some cases. However, it is not necessary or perhaps even feasible in others. More research is needed to understand *why* it is less helpful.

REFERENCES

- [1] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 385–395, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351058. doi: 10.1145/3106237.3106267. URL <https://doi.org/10.1145/3106237.3106267>.
- [2] R. Abdalkareem, V. Oda, S. Mujahid, and E. Shihab. On the impact of using trivial packages: An empirical case study on npm and pypi. *Empirical Software Engineering*, 25:1168–1204, 2020.
- [3] A. Abdellatif, Y. Zeng, M. Elshafei, E. Shihab, and W. Shang. Simplifying the search of npm packages. *Information and Software Technology*, 126:106365, 2020. ISSN 0950-5849. doi: <https://doi.org/10.1016/j.infsof.2020.106365>. URL <https://www.sciencedirect.com/science/article/pii/S0950584920301336>.
- [4] A. Abran. *Software metrics and software metrology*. John Wiley & Sons, 2010.
- [5] E. Ackerman. Upgrade to Superhuman Reflexes Without Feeling Like a Robot. *IEEE Spectrum*, 2019. URL <https://spectrum.ieee.org/enabling-superhuman-reflexes-without-feeling-like-a-robot>. Publisher: IEEE.
- [6] O. Alhazmi, Y. Malaiya, and I. Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers & Security*, 26(3):219–228, 2007. ISSN 0167-4048. doi: <https://doi.org/10.1016/j.cose.2006.10.002>. URL <https://www.sciencedirect.com/science/article/pii/S0167404806001520>.
- [7] A. Alhuzali, B. Eshete, R. Gjomemo, and V. Venkatakrisnan. Chainsaw: Chained Automated Workflow-based Exploit Generation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 641–652, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4139-4. event-place: Vienna, Austria.
- [8] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrisnan. {NAVEX}: Precise and scalable exploit generation for dynamic web applications. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 377–392, 2018.
- [9] L. Allodi and F. Massacci. A Preliminary Analysis of Vulnerability Scores for Attacks in Wild: The Ekits and Sym Datasets. In *Proceedings of the 2012 ACM Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, BADGERS '12, pages 17–24, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1661-3.
- [10] L. Allodi and F. Massacci. Comparing vulnerability severity and exploits using case-control studies. *ACM Transactions on Information and System Security (TISSEC)*, 17(1): 1–20, 2014. Publisher: ACM New York, NY, USA.

- [11] L. Allodi, S. Banescu, H. Femmer, and K. Beckers. Identifying Relevant Information Cues for Vulnerability Assessment Using CVSS. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY '18*, pages 119–126, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5632-9.
- [12] L. Allodi, M. Cremonini, F. Massacci, and W. Shim. Measuring the accuracy of software vulnerability assessments: experiments with students and professionals. *Empirical Software Engineering*, 25(2):1063–1094, 2020. Publisher: Springer.
- [13] M. Almukaynizi, A. Grimm, E. Nunes, J. Shakarian, and P. Shakarian. Predicting Cyber Threats through Hacker Social Networks in Darkweb and Deepweb Forums. In *Proceedings of the 2017 International Conference of The Computational Social Science Society of the Americas, CSS 2017*, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5269-7.
- [14] M. Almukaynizi, E. Nunes, K. Dharaiya, M. Senguttuvan, J. Shakarian, and P. Shakarian. Proactive identification of exploits in the wild through vulnerability mentions online. In *2017 International Conference on Cyber Conflict (CyCon U.S.)*, pages 82–88, Washington, DC, USA, 2017. IEEE.
- [15] N. Alomar, P. Wijesekera, E. Qiu, and S. Egelman. “you’ve got your nice list of bugs, now what?” vulnerability discovery and management processes in the wild. In *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*, pages 319–339, Delaware, USA, 2020. USENIX Association. ISBN 978-1-939133-16-8.
- [16] K. Alperin, A. Wollaber, D. Ross, P. Trepagnier, and L. Leonard. Risk Prioritization by Leveraging Latent Vulnerability Features in a Contested Environment. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security, AISEC'19*, pages 49–57, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6833-9.
- [17] R. Amankwah, J. Chen, P. K. Kudjo, and D. Towey. An empirical comparison of commercial and open-source web vulnerability scanners. *Software: Practice and Experience*, 50(9): 1842–1857, 2020. Publisher: Wiley Online Library.
- [18] anaconda. Anaconda package readme, 2024. URL <https://github.com/ChimeraCoder/anaconda/blob/master/README.md>. Online; Accessed: 22-July-2024.
- [19] T. Anderson. Linux in 2020: 27.8 million lines of code in the kernel, 1.3 million in systemd. *The Register*, 2020. URL https://www.theregister.com/2020/01/06/linux_2020_kernel_systemd_code/.
- [20] N. Antunes and M. Vieira. Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services. In *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 301–306. IEEE, 2009.
- [21] N. Antunes and M. Vieira. Benchmarking vulnerability detection tools for web services. In *2010 IEEE International Conference on Web Services*, pages 203–210. IEEE, 2010.

- [22] A. Austin and L. Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 97–106. IEEE, 2011.
- [23] A. Austin, C. Holmgreen, and L. Williams. A comparison of the efficiency and effectiveness of vulnerability discovery techniques. *Information and Software Technology*, 55(7):1279–1288, 2013. Publisher: Elsevier.
- [24] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation. In *18th Annual Network & Distributed System Security Symposium Proceedings*, San Diego, California, USA, 2011. Network and Distributed Systems Security (NDSS) Symposium.
- [25] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic Exploit Generation. *Commun. ACM*, 57(2):74–84, Feb. 2014. ISSN 0001-0782. doi: 10.1145/2560217.2560219. URL <https://doi.org/10.1145/2560217.2560219>. Place: New York, NY, USA Publisher: ACM.
- [26] A. Bannister. Healthcare provider Texas ENT alerts 535,000 patients to data breach. *The Daily Swig*, 2021. [Online; Publication Date 20-Dec-2021; Accessed: 21-Dec-2021].
- [27] M. S. Bartlett. Properties of sufficiency and statistical tests. *Proceedings of the Royal Society of London. Series A-Mathematical and Physical Sciences*, 160(901):268–282, 1937. Publisher: The Royal Society London.
- [28] J. Bau, F. Wang, E. Bursztein, P. Mutchler, and J. C. Mitchell. Vulnerability Factors in New Web Applications: Audit Tools, Developer Selection & Languages. Technical report, Stanford, 2012. URL <https://seclab.stanford.edu/websec/scannerPaper.pdf>.
- [29] A. Begel, J. Bosch, and M.-A. Storey. Social networking meets software development: Perspectives from github, msdn, stack exchange, and topcoder. *IEEE Software*, 30(1): 52–66, 2013. doi: 10.1109/MS.2013.13.
- [30] M. A. Belabbes, I. Ruthven, Y. Moshfeghi, and D. Rasmussen Pennington. Information overload: a concept analysis. *Journal of Documentation*, 79(1):144–159, 2023.
- [31] Bitdefender. Bitdefender: What is an exploit? exploit prevention, 2023. URL <https://www.bitdefender.com/consumer/support/answer/10556/>. [Online; Accessed: 17-May-2023].
- [32] H. Borges and M. Tulio Valente. What’s in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129, 2018. ISSN 0164-1212. doi: <https://doi.org/10.1016/j.jss.2018.09.016>. URL <https://www.sciencedirect.com/science/article/pii/S0164121218301961>.
- [33] M. Bozorgi, L. K. Saul, S. Savage, and G. M. Voelker. Beyond Heuristics: Learning to Classify Vulnerabilities and Predict Exploits. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD ’10*, pages 105–114, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0055-1. event-place: Washington, DC, USA.

- [34] B. L. Bullough, A. K. Yanchenko, C. L. Smith, and J. R. Zipkin. Predicting exploitation of disclosed software vulnerabilities using open-source data. In *Proceedings of the 3rd ACM on International Workshop on Security And Privacy Analytics*, pages 45–53. ACM, 2017.
- [35] G. A. Campbell. What is 'taint analysis' and why do I care?, Feb. 2020. URL <https://blog.sonarsource.com/what-is-taint-analysis>.
- [36] M. Campbell. Keep your dependencies secure and up-to-date with github and dependabot. <https://github.blog/2019-01-31-keep-your-dependencies-secure-and-up-to-date-with-github-and-dependabot/>, Jan 2019. Online; Accessed: 08-Jun-2023.
- [37] S. Cass. Top Programming Languages 2021. *IEEE Spectrum*, Aug. 2021. URL <https://spectrum.ieee.org/top-programming-languages-2021>. Publisher: IEEE.
- [38] S. Cass, P. Kulkarni, and E. Guizzo. Interactive: Top Programming Languages 2021, 2021. URL <https://spectrum.ieee.org/top-programming-languages/>. [Online; Accessed: 20-Apr-2022].
- [39] cgroups. cgroups package readme, 2024. URL <https://github.com/containerd/cgroups/blob/master/README.md>. Online; Accessed: 22-July-2024.
- [40] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, 2012. doi: 10.1109/SP.2012.31.
- [41] Y. Chen and X. Xing. SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19*, pages 1707–1722, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6747-9. doi: 10.1145/3319535.3363212. URL <https://doi.org/10.1145/3319535.3363212>. event-place: London, United Kingdom.
- [42] D. V. Cicchetti and A. R. Feinstein. High agreement but low kappa: II. Resolving the paradoxes. *Journal of clinical epidemiology*, 43(6):551–558, 1990. Publisher: Elsevier.
- [43] Cisco. Cisco: What is an exploit?, 2023. URL <https://www.cisco.com/c/en/us/products/security/advanced-malware-protection/what-is-exploit.html>. [Online; Accessed: 17-May-2023].
- [44] Clarivate. Journal citation reports [portal], 2023. URL <https://jcr.clarivate.com/jcr/home>.
- [45] Cobra. Cobra package readme, 2024. URL <https://github.com/spf13/cobra/blob/master/README.md>. Online; Accessed: 22-July-2024.
- [46] J. Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960. Publisher: Sage Publications Sage CA: Thousand Oaks, CA.

- [47] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Routledge, 2nd edition, 1988.
- [48] C. Condon and H. Miller. Maryland health department says there's no evidence of data lost after cyberattack; website is back online. *Baltimore Sun*, 2021. URL <https://www.baltimoresun.com/health/bs-hs-mdh-website-down-20211206-o2ky2sn5znb3pdwtnu2a7m5g6q-story.html>. Online; Publication Date: 06-Dec-2021 [Online; Accessed: 21-Dec-2021].
- [49] T. D. Cook and D. T. Campbell. *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Rand McNally College Publishing, 1979.
- [50] J. Corbin and A. Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, Inc., 3rd edition, 2008.
- [51] R. Correa, J. R. Bermejo Higuera, J. B. Higuera, J. A. Sicilia Montalvo, M. S. Rubio, and Á. A. Magreñán. Hybrid Security Assessment Methodology for Web Applications. *Computer Modeling in Engineering & Sciences*, 126(1):89–124, 2021. Publisher: Tech Science Press.
- [52] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 119–129. IEEE, 2000.
- [53] R. Cox, R. Griesemer, R. Pike, I. L. Taylor, and K. Thompson. The go programming language and environment. *Commun. ACM*, 65(5):70–78, apr 2022. ISSN 0001-0782. doi: 10.1145/3488716. URL <https://doi.org/10.1145/3488716>.
- [54] D. S. Cruzes, M. Felderer, T. D. Oyetoyan, M. Gander, and I. Pekaric. How is security testing done in agile teams? a cross-case analysis of four software teams. In *International Conference on Agile Software Development*, pages 201–216. Springer, Cham, 2017.
- [55] CVE. Cve program (website), 2022. URL <https://www.cve.org/>. [Online; Accessed 23-Mar-2022].
- [56] CVE program. Cve process, 2023. URL <https://www.cve.org/About/Process>. Online; Accessed: 24-Oct-2023.
- [57] CVSS Special Interest Group (SIG). Common vulnerability scoring system v3.0: User guide. Technical report, Forum of Incident Response and Security Teams (FIRST), 2015. URL https://www.first.org/cvss/v3.0/cvss-v30-user_guide_v1.6.pdf. [Online; Accessed: 30-Mar-2022].
- [58] CVSS Special Interest Group (SIG). Common Vulnerability Scoring System v3.0: Examples. Technical report, Forum of Incident Response and Security Teams (FIRST), 2017. URL https://www.first.org/cvss/v3.0/cvss-v30-examples_v1.5.pdf. [Online; Accessed: 14-Nov-2022].

- [59] CVSS Special Interest Group (SIG). Common vulnerability scoring system v3.1 specification document. Technical report, Forum of Incident Response and Security Teams (FIRST), 2019. URL <https://www.first.org/cvss/v3.1/specification-document>. [Online; Accessed: 24-Mar-2022].
- [60] CVSS Special Interest Group (SIG). Common Vulnerability Scoring System v3.1: User Guide. Technical report, Forum of Incident Response and Security Teams (FIRST), 2019. URL https://www.first.org/cvss/v3-1/cvss-v31-user-guide_r1.pdf.
- [61] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in github: Transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work, CSCW '12*, page 1277–1286, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450310864. doi: 10.1145/2145204.2145396. URL <https://doi.org/10.1145/2145204.2145396>.
- [62] S. Dambra, L. Bilge, and D. Balzarotti. SoK: Cyber insurance—technical challenges and a system security roadmap. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1367–1383. IEEE, 2020.
- [63] A. Dann, H. Plate, B. Hermann, S. E. Ponta, and E. Bodden. Identifying challenges for oss vulnerability scanners - a study & test suite. *IEEE Transactions on Software Engineering*, 48(9):3613–3625, 2022. doi: 10.1109/TSE.2021.3101739.
- [64] F. D. Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS quarterly*, pages 319–340, 1989. Publisher: JSTOR.
- [65] B. Day. OSVDB: An Independent and Open Source Vulnerability Database. *LinuxSecurity (Web)*, Dec. 2003. URL <https://linuxsecurity.com/features/osvdb-an-independent-and-open-source-vulnerability-database>.
- [66] F. L. de la Mora and S. Nadi. An empirical study of metric-based comparisons of software libraries. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE'18*, page 22–31, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365932. doi: 10.1145/3273934.3273937. URL <https://doi.org/10.1145/3273934.3273937>.
- [67] A. Delaitre, P. E. Black, D. Cupif, G. Haben, A.-K. Loembe, V. Okun, and Y. Prono. Sate vi report: Bug injection and collection. NIST SP 500-341, National Institute of Standards and Technology (NIST), 2020. URL <https://doi.org/10.6028/NIST.SP.500-341>.
- [68] A. M. Delaitre, B. C. Stivalet, P. E. Black, V. Okun, T. S. Cohen, and A. Ribeiro. SATE V Report: Ten Years of Static Analysis Tool Expositions. NIST SP 500-326, National Institute of Standards and Technology (NIST), 2018. URL <https://doi.org/10.6028/NIST.SP.500-326>.
- [69] F. Deng, J. Wang, B. Zhang, C. Feng, Z. Jiang, and Y. Su. A Pattern-Based Software Testing Framework for Exploitability Evaluation of Metadata Corruption Vulnerabilities. *Scientific Programming*, 2020, 2020. Publisher: Hindawi.

- [70] Department for Digital, Culture, Media and Sport. Cyber security breaches survey 2024, 2024. URL <https://www.gov.uk/government/statistics/cyber-security-breaches-survey-2024/cyber-security-breaches-survey-2024>. Online; Accessed: 31-July-2024.
- [71] J. Desjardins. Here's how many millions of lines of code it takes to run different software. *Business Insider*, 2017. URL <https://www.businessinsider.com/how-many-lines-of-code-it-takes-to-run-different-software-2017-2>.
- [72] J. L. Devore. *Probability and Statistics for Engineering and the Sciences*. Cengage Learning, ninth edition, 2014.
- [73] A. Dobrovoljc, D. Trček, and B. Likar. Predicting Exploitations of Information Systems Vulnerabilities Through Attackers' Characteristics. *IEEE Access*, 5:26063–26075, 2017.
- [74] A. Doupé, M. Cova, and G. Vigna. Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 111–131. Springer, 2010.
- [75] T. Dullien. Weird Machines, Exploitability, and Provable Unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 8(2):391–403, 2017. doi: 10.1109/TETC.2017.2785299.
- [76] T. Dumitraş and D. Shou. Toward a standard benchmark for computer security research: The Worldwide Intelligence Network Environment (WINE). In *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, pages 89–96, New York, NY, USA, 2011. ACM.
- [77] T. Dunlap, J. S. Meyers, B. Reaves, and W. Enck. Pairing security advisories with vulnerable functions using open-source llms. In F. Maggi, M. Egele, M. Payer, and M. Carminati, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 350–369, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-64171-8.
- [78] P. K. Dunn and G. K. Smyth. *Chapter 1: Statistical Models*, pages 1–30. Springer New York, New York, NY, 2018. ISBN 978-1-4419-0118-7. doi: 10.1007/978-1-4419-0118-7_1. URL https://doi.org/10.1007/978-1-4419-0118-7_1.
- [79] M. Edkrantz. Predicting exploit likelihood for cyber vulnerabilities with machine learning. Master's thesis, Chalmers University of Technology, 2015.
- [80] M. Edkrantz and A. Said. Predicting Cyber Vulnerability Exploits with Machine Learning. In *SCAI*, 2015.
- [81] C. Elbaz, L. Rilling, and C. Morin. Fighting N-Day Vulnerabilities with Automated CVSS Vector Prediction at Disclosure. In *Proceedings of the 15th International Conference on Availability, Reliability and Security, ARES '20*, New York, NY, USA, 2020. ACM. ISBN 978-1-4503-8833-7.

- [82] S. Elder, N. Zahan, R. Shu, M. Metro, V. Kozarev, T. Menzies, and L. Williams. Do i really need all this work to find vulnerabilities? an empirical case study comparing vulnerability detection techniques on a java application. *Empirical Software Engineering*, 27(6):154, 2022.
- [83] S. E. Elder, N. Zahan, V. Kozarev, R. Shu, T. Menzies, and L. Williams. Structuring a Comprehensive Software Security Course Around the OWASP Application Security Verification Standard. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 95–104. IEEE, 2021.
- [84] K. E. Emam. Benchmarking Kappa: Interrater agreement in software process assessments. *Empirical Software Engineering*, 4:113–133, 1999. Publisher: Springer.
- [85] Epic Systems Corporation. From Healthcare to Mapping the Milky Way: 5 Things You Didn't Know About Epic's Tech, Feb. 2020. URL <https://www.epic.com/epic/post/healthcare-mapping-milky-way-5-things-didnt-know-epics-tech>.
- [86] Executive Order 14028. Executive order on improving the nation's cybersecurity. Exec. Order No. 14028, 86 FR 26633, May 2021. URL <https://www.federalregister.gov/d/2021-10460>.
- [87] ExploitDB. Exploit database history, 2022. URL <https://www.exploit-db.com/history>. [Online; Accessed: 22-Sept-2022].
- [88] F. Fagerholm, M. Felderer, D. Fucci, M. Unterkalmsteiner, B. Marculescu, M. Martini, L. G. W. Tengberg, R. Feldt, B. Lehtelä, B. Nagyvárad, and J. Khattak. Cognition in software engineering: A taxonomy and survey of a half-century of research. *ACM Comput. Surv.*, 54(11s), sep 2022. ISSN 0360-0300. doi: 10.1145/3508359. URL <https://doi.org/10.1145/3508359>.
- [89] fasthttp. fasthttp readme, 2024. URL <https://github.com/valyala/fasthttp/blob/master/README.md>. Online; Accessed: 22-July-2024.
- [90] A. R. Feinstein and D. V. Cicchetti. High agreement but low kappa: I. The problems of two paradoxes. *Journal of clinical epidemiology*, 43(6):543–549, 1990. Publisher: Elsevier.
- [91] R. Feldt and A. Magazinius. Validity threats in empirical software engineering research-an initial survey. In *Seke*, pages 374–379, 2010.
- [92] G. C. Feng. Factors affecting intercoder reliability: A Monte Carlo experiment. *Quality & Quantity*, 47(5):2959–2982, 2013. Publisher: Springer.
- [93] N. E. Fenton and S. L. Pfleeger. *Software metrics: a rigorous and practical approach*. PWS Publishing Company, Boston, 2nd edition, 1997.
- [94] R. T. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, jun 2014. URL <https://rfc-editor.org/rfc/rfc7231.txt>.

- [95] M. Finifter, D. Akhawe, and D. Wagner. An Empirical Study of Vulnerability Rewards Programs. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 273–288, Washington, D.C., 2013. USENIX. ISBN 978-1-931971-03-4.
- [96] FIRST. *The EPSS Model (Website)*. Forum of Incident Response and Security Teams (FIRST), 2022. URL <https://www.first.org/epss/model>. [Online; Accessed: 11-Nov-2022].
- [97] J. Fonseca, M. Vieira, and H. Madeira. Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks. In *13th Pacific Rim international symposium on dependable computing (PRDC 2007)*, pages 365–372. IEEE, 2007.
- [98] Fortinet. Fortinet: exploit definition, 2023. URL <https://www.fortinet.com/resources/cyberglossary/exploit>. [Online; Accessed: 17-May-2023].
- [99] S. Frei, M. May, U. Fiedler, and B. Plattner. Large-Scale Vulnerability Analysis. In *Proceedings of the 2006 SIGCOMM Workshop on Large-Scale Attack Defense, LSAD '06*, pages 131–138, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1-59593-571-1. doi: 10.1145/1162666.1162671. URL <https://doi.org/10.1145/1162666.1162671>. event-place: Pisa, Italy.
- [100] S. Frei, D. Schatzmann, B. Plattner, and B. Trammell. Modeling the security ecosystem—the dynamics of (in) security. In *Economics of Information Security and Privacy*, pages 79–106. Springer, 2010.
- [101] C. Frühwirth and T. Mannisto. Improving CVSS-based vulnerability prioritization and response with context information. In *3rd International Symposium on Empirical Software Engineering and Measurement*, pages 535–544, 2009.
- [102] D. Galligani, R. Gjomemo, V. Venkatakrishnan, and S. Zanero. Static Detection and Automatic Exploitation of Intent Message Vulnerabilities in Android Applications. In *Mobile Security Technologies (MoST) 2015*, 2015.
- [103] L. Gallon. Vulnerability Discrimination Using CVSS Framework. In *2011 4th IFIP International Conference on New Technologies, Mobility and Security*, pages 1–6, 2011. doi: 10.1109/NTMS.2011.5720656.
- [104] P. A. Games and J. F. Howell. Pairwise multiple comparison procedures with unequal n's and/or variances: a Monte Carlo study. *Journal of Educational Statistics*, 1(2):113–125, 1976. Publisher: Sage Publications Sage CA: Thousand Oaks, CA.
- [105] J. Garcia, M. Hammad, N. Ghorbani, and S. Malek. Automatic Generation of Inter-Component Communication Exploits for Android Applications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017*, pages 661–671, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5105-8.
- [106] B. Garmany, M. Stoffel, R. Gawlik, P. Koppe, T. Blazytko, and T. Holz. Towards Automated Generation of Exploitation Primitives for Web Browsers. In *Proceedings of the 34th Annual*

- Computer Security Applications Conference, ACSAC '18*, pages 300–312, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6569-7.
- [107] GII-GRIN-SCIE. The GII-GRIN-SCIE (GGS) Conference Rating, 2022. URL <https://scie.lcc.uma.es/gii-grin-scie-rating/conferenceRating.jsf>.
- [108] Github. The 2021 State of the Octoverse, 2021. URL <https://octoverse.github.com/>. [Online; Accessed: 20-Apr-2022].
- [109] Github. Fork a repo, 2023. URL <https://docs.github.com/en/get-started/quickstart/fork-a-repo>. Online; Accessed: 24-Oct-2023.
- [110] Github. Github rest api documentation, 2024. URL <https://docs.github.com/en/rest>. Online; Accessed: 22-July-2024.
- [111] Go. Why go -> case studies (website), 2024. URL <https://go.dev/solutions/case-studies>. Online; Accessed: 22-July-2024.
- [112] J. Gold. Open-source vulnerabilities database shuts down. *CSO Online (Web)*, Apr. 2016. URL <https://www.csoonline.com/article/3053549/open-source-vulnerabilities-database-shuts-down.html>.
- [113] X. Gong, Z. Xing, X. Li, Z. Feng, and Z. Han. Joint Prediction of Multiple Vulnerability Characteristics Through Multi-Task Learning. In *24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 31–40, Guangzhou, China, 2019. IEEE.
- [114] L. Gonçales, K. Farias, and B. C. da Silva. Measuring the cognitive load of software developers: An extended Systematic Mapping Study. *Information and Software Technology*, page 106563, 2021. Publisher: Elsevier.
- [115] Google. Open source insights (website), 2024. URL <https://deps.dev/>. Online; Accessed: 22-July-2024.
- [116] Google. Open source insights (website), 2024. URL <https://docs.deps.dev/faq/>. Online; Accessed: 22-July-2024.
- [117] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, page 345–355, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568260. URL <https://doi.org/10.1145/2568225.2568260>.
- [118] M. Hafiz and M. Fang. Game of detections: how are security vulnerabilities discovered in the wild? *Empirical Software Engineering*, 21(5):1920–1959, 2016. Publisher: Springer.
- [119] H. Hata, T. Todo, S. Onoue, and K. Matsumoto. Characteristics of sustainable oss projects: A theoretical and empirical study. In *2015 IEEE/ACM 8th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 15–21, 2015. doi: 10.1109/CHASE.2015.9.

- [120] L. He, Y. Cai, H. Hu, P. Su, Z. Liang, Y. Yang, H. Huang, J. Yan, X. Jia, and D. Feng. Automatically assessing crashes from heap overflows. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 274–279, 2017.
- [121] M. L. Head, L. Holman, R. Lanfear, A. T. Kahn, and M. D. Jennions. The extent and consequences of p-hacking in science. *PLoS biology*, 13(3):e1002106, 2015.
- [122] S. Hedge. Linux permissions: An introduction to chmod, 2019. URL <https://www.redhat.com/sysadmin/introduction-chmod>. [Online; Accessed: 16-Aug-2023].
- [123] S. Heelan and A. Gianni. Augmenting Vulnerability Analysis of Binary Code. In *Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC '12*, pages 199–208, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1312-4.
- [124] S. Heelan, T. Melham, and D. Kroening. Automatic Heap Layout Manipulation for Exploitation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 763–779, Baltimore, MD, Aug. 2018. USENIX Association. ISBN 978-1-939133-04-5. URL <https://www.usenix.org/conference/usenixsecurity18/presentation/heelan>.
- [125] S. Heelan, T. Melham, and D. Kroening. Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19*, pages 1689–1706, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6747-9. doi: 10.1145/3319535.3354224. URL <https://doi.org/10.1145/3319535.3354224>. event-place: London, United Kingdom.
- [126] H. Holm and K. K. Afridi. An expert-based investigation of the Common Vulnerability Scoring System. *Computers & Security*, 53:18–30, 2015. ISSN 0167-4048. doi: <https://doi.org/10.1016/j.cose.2015.04.012>. URL <https://www.sciencedirect.com/science/article/pii/S0167404815000620>.
- [127] F. Hou and S. Jansen. A systematic literature review on trust in the software ecosystem. *Empirical Software Engineering*, 28(1):8, 2023.
- [128] J. Hu, L. Zhang, C. Liu, S. Yang, S. Huang, and Y. Liu. Empirical analysis of vulnerabilities life cycle in golang ecosystem. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24*, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3639230. URL <https://doi.org/10.1145/3597503.3639230>.
- [129] C.-C. Huang, F.-Y. Lin, F. Y.-S. Lin, and Y. S. Sun. A novel approach to evaluate software vulnerability prioritization. *Journal of Systems and Software*, 86(11):2822–2840, 2013. ISSN 0164-1212. doi: <https://doi.org/10.1016/j.jss.2013.06.040>. URL <https://www.sciencedirect.com/science/article/pii/S0164121213001489>.
- [130] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and W.-M. Leong. CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 78–87, 2012. doi: 10.1109/SERE.2012.20.

- [131] S.-K. Huang, M.-H. Huang, P.-Y. Huang, H.-L. Lu, and C.-W. Lai. Software Crash Analysis for Automatic Exploit Generation on Binary Programs. *IEEE Transactions on Reliability*, 63(1):270–289, 2014. doi: 10.1109/TR.2014.2299198.
- [132] E. Iannone, D. D. Nucci, A. Sabetta, and A. De Lucia. Toward Automated Exploit Generation for Known Vulnerabilities in Open-Source Libraries. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 396–400, 2021. doi: 10.1109/ICPC52881.2021.00046.
- [133] N. Imtiaz, A. Rahman, E. Farhana, and L. Williams. Challenges with Responding to Static Analysis Tool Alerts. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR '19*, pages 245–249, Piscataway, NJ, USA, 2019. IEEE Press. doi: 10.1109/MSR.2019.00049. URL <https://doi.org/10.1109/MSR.2019.00049>. event-place: Montreal, Quebec, Canada.
- [134] ISO/IEC/IEEE. Systems and software engineering - vocabulary. ISO/IEC/IEEE 24765:2010, International Organization for Standardization (ISO), International Electrotechnical Commission (IES), and Institute of Electrical and Electronics Engineers (IEEE), Dec. 2010.
- [135] ISO/IEC/IEEE. Software and Systems Engineering — Software Testing — Part 1: concepts and definitions. ISO/IEC/IEEE 29119-1:2013, International Organization for Standardization (ISO), International Electrotechnical Commission (IES), and Institute of Electrical and Electronics Engineers (IEEE), Sept. 2013.
- [136] J. Itkonen and M. V. Mäntylä. Are test cases needed? Replicated comparison between exploratory and test-case-based software testing. *Empirical Software Engineering*, 19(2): 303–342, 2014. Publisher: Springer.
- [137] J. Itkonen, M. V. Mäntylä, and C. Lassenius. The role of the tester’s knowledge in exploratory software testing. *IEEE Transactions on Software Engineering*, 39(5):707–724, 2013. Publisher: IEEE.
- [138] J. Jacobs, S. Romanosky, I. Adjerid, and W. Baker. Improving vulnerability remediation through better exploit prediction. *Journal of Cybersecurity*, 6(1):tyaa015, 2020. Publisher: Oxford University Press.
- [139] J. Jacobs, S. Romanosky, B. Edwards, I. Adjerid, and M. Roytman. Exploit Prediction Scoring System (EPSS). *Digital Threats*, 2(3), July 2021. ISSN 2692-1626. doi: 10.1145/3436242. URL <https://doi.org/10.1145/3436242>. Place: New York, NY, USA Publisher: ACM.
- [140] Y. Jiang and Y. Atif. A selective ensemble model for cognitive cybersecurity analysis. *Journal of Network and Computer Applications*, 193:103210, 2021. ISSN 1084-8045. doi: <https://doi.org/10.1016/j.jnca.2021.103210>. URL <https://www.sciencedirect.com/science/article/pii/S1084804521002125>.
- [141] Z. Jiang, S. Gan, A. Herrera, F. Toffalini, L. Romerio, C. Tang, M. Egele, C. Zhang, and M. Payer. Evocatio: Conjuring Bug Capabilities from a Single PoC. In *Proceedings of*

- the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS '22*, pages 1599–1613, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 978-1-4503-9450-5. doi: 10.1145/3548606.3560575. URL <https://doi.org/10.1145/3548606.3560575>. event-place: Los Angeles, CA, USA.
- [142] H. Joh and Y. K. Malaiya. Defining and assessing quantitative security risk measures using vulnerability lifecycle and cvss metrics. In *The 2011 international conference on security and management (sam)*, pages 10–16, 2011.
- [143] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681. IEEE Press, 2013.
- [144] P. Johnson, R. Lagerström, M. Ekstedt, and U. Franke. Can the Common Vulnerability Scoring System be Trusted? A Bayesian Analysis. *IEEE Transactions on Dependable and Secure Computing*, 15(6):1002–1015, 2018. doi: 10.1109/TDSC.2016.2644614.
- [145] Joint Task Force Transformation Initiative. Security and privacy controls for federal information systems and organizations. NIST SP 800-53, National Institute of Standards and Technology (NIST), April 2013. URL <http://dx.doi.org/10.6028/NIST.SP.800-53r4>.
- [146] Joint Task Force Transformation Initiative. Security and Privacy Controls for Federal Information Systems and Organizations. NIST SP 800-53 Rev. 5, National Institute of Standards and Technology (NIST), Apr. 2020. URL <https://doi.org/10.6028/NIST.SP.800-53r5>.
- [147] H. J. Kang, T. G. Nguyen, B. Le, C. S. Păsăreanu, and D. Lo. Test Mimicry to Assess the Exploitability of Library Vulnerabilities. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2022*, pages 276–288, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 978-1-4503-9379-9. doi: 10.1145/3533767.3534398. URL <https://doi.org/10.1145/3533767.3534398>. event-place: Virtual, South Korea.
- [148] R. Kirk. *Experimental Design: Procedures for the Behavioral Sciences*. Thousand Oaks : Sage Publications, 4 edition, 2013.
- [149] B. Kitchenham, L. Madeyski, D. Budgen, J. Keung, P. Brereton, S. Charters, S. Gibbs, and A. Pohthong. Robust statistical methods for empirical software engineering. *Empirical Software Engineering*, 22(2):579–630, 2017. Publisher: Springer.
- [150] B. A. Kitchenham, D. Budgen, and P. Brereton. *Evidence-based software engineering and systematic reviews*, volume 4. CRC press, 2015.
- [151] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

- [152] M. Koster, G. Illyes, H. Zeller, and L. Sassman. Robots Exclusion Protocol, 2022. URL <https://www.rfc-editor.org/rfc/internet-drafts/draft-koster-rep-12.html>.
- [153] I. Kotenko, K. Izrailov, and M. Buinevich. Static Analysis of Information Systems for IoT Cyber Security: A Survey of Machine Learning Approaches. *Sensors*, 22(4), 2022. ISSN 1424-8220. doi: 10.3390/s22041335. URL <https://www.mdpi.com/1424-8220/22/4/1335>.
- [154] V. Kotov and F. Massacci. Anatomy of Exploit Kits. In J. Jürjens, B. Livshits, and R. Scandariato, editors, *Engineering Secure Software and Systems*, pages 181–196, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-36563-8.
- [155] E. Kovacs. OSVDB Shut Down Permanently. *Security Week (Web)*, Apr. 2016. URL <https://www.securityweek.com/osvdb-shut-down-permanently>.
- [156] J. R. Landis and G. G. Koch. The Measurement of Observer Agreement for Categorical Data. *Biometrics*, 33(1):159–174, 1977. ISSN 0006341X, 15410420. URL <http://www.jstor.org/stable/2529310>. Publisher: [Wiley, International Biometric Society].
- [157] E. Larios Vargas, M. Aniche, C. Treude, M. Bruntink, and G. Gousios. Selecting third-party libraries: The practitioners’ perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 245–256, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3409711. URL <https://doi.org/10.1145/3368089.3409711>.
- [158] K. Larkin, M. Wojciakowski, A. Junker, and M. Schofield. Windows package manager. <https://learn.microsoft.com/en-us/windows/package-manager/>, Feb 2023. Online; Accessed: 09-Jun-2023.
- [159] A. H. Lashkari, A. F. A. Kadir, L. Taheri, and A. A. Ghorbani. Toward Developing a Systematic Approach to Generate Benchmark Android Malware Datasets and Classification. In *2018 International Carnahan Conference on Security Technology (ICCST)*, 2018. doi: 10.1109/CCST.2018.8585560.
- [160] J. Latendresse, S. Mujahid, D. E. Costa, and E. Shihab. Not all dependencies are equal: An empirical study on production dependencies in npm. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE ’22, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394758. doi: 10.1145/3551349.3556896. URL <https://doi.org/10.1145/3551349.3556896>.
- [161] T. H. M. Le, B. Sabir, and M. A. Babar. Automated Software Vulnerability Assessment with Concept Drift. In *IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 371–382, 2019.
- [162] T. H. M. Le, H. Chen, and M. A. Babar. A Survey on Data-Driven Software Vulnerability Assessment and Prioritization. *ACM Comput. Surv.*, Mar. 2022. ISSN 0360-0300. doi:

10.1145/3529757. URL <https://doi-org.prox.lib.ncsu.edu/10.1145/3529757>. Place: New York, NY, USA Publisher: ACM.

- [163] Y. Lee, C. Min, and B. Lee. {ExpRace}: Exploiting kernel races through raising interrupts. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2363–2380, 2021.
- [164] X. Li, S. Moreschini, Z. Zhang, and D. Taibi. Exploring factors and metrics to select open source software components for integration: An empirical study. *Journal of Systems and Software*, 188:111255, 2022. ISSN 0164-1212. doi: <https://doi.org/10.1016/j.jss.2022.111255>. URL <https://www.sciencedirect.com/science/article/pii/S0164121222000267>.
- [165] M. Liu, B. Zhang, W. Chen, and X. Zhang. A survey of exploitation and detection methods of XSS vulnerabilities. *IEEE Access*, 7:182004–182016, 2019. Publisher: IEEE.
- [166] Q. Liu and Y. Zhang. VRSS: A new system for rating and scoring vulnerabilities. *Computer Communications*, 34(3):264–273, 2011. ISSN 0140-3664. doi: <https://doi.org/10.1016/j.comcom.2010.04.006>. URL <https://www.sciencedirect.com/science/article/pii/S014036641000174X>.
- [167] Q. Liu, K. Bao, and V. Hagenmeyer. Binary Exploitation in Industrial Control Systems: Past, Present and Future. *IEEE Access*, 10:48242–48273, 2022. doi: 10.1109/ACCESS.2022.3171922.
- [168] M. Lombard, J. Snyder-Duch, and C. C. Bracken. Content analysis in mass communication: Assessment and reporting of intercoder reliability. *Human communication research*, 28(4):587–604, 2002. Publisher: Wiley Online Library.
- [169] J. Lung, J. Aranda, S. Easterbrook, and G. Wilson. On the difficulty of replicating human subjects studies in software engineering. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 191–200, 2008.
- [170] J. Luo, K. Lo, and H. Qu. A software vulnerability rating approach based on the vulnerability database. *Journal of Applied Mathematics*, 2014, 2014. Publisher: Hindawi.
- [171] F. Mallet. SonarAnalyzer for Java: Tricky Bugs are Running Scared, 2016. URL <https://blog.sonarsource.com/sonaranalyzer-for-java-tricky-bugs-are-running-scared>.
- [172] G. McGraw. *Software security: building security in*. Addison-Wesley Professional, 2006.
- [173] P. Mell, K. Scarfone, and S. Romanosky. Common Vulnerability Scoring System. *IEEE Security & Privacy*, 4(6):85–89, 2006. doi: 10.1109/MSP.2006.145.
- [174] P. Mell, K. Scarfone, and S. Romansky. A Complete Guide to the Common Vulnerability Scoring System Version 2.0. Technical report, Forum of Incident Response and Security Teams (FIRST), 2007. URL <https://www.first.org/cvss/v2/cvss-v2-guide.pdf>.

- [175] R. Meloca, G. Pinto, L. Baiser, M. Mattos, I. Polato, I. S. Wiese, and D. M. German. Understanding the usage, impact, and adoption of non-osi approved licenses. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 270–280, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357166. doi: 10.1145/3196398.3196427. URL <https://doi.org/10.1145/3196398.3196427>.
- [176] Microsoft. Microsoft Exploitability Index (Website), 2022. URL <https://www.microsoft.com/en-us/msrc/exploitability-index>. [Online; Accessed: 10-Apr-2022].
- [177] Microsoft. Security Update Severity Rating System (Website), 2022. URL <https://www.microsoft.com/en-us/msrc/security-update-severity-rating-system>. [Online; Accessed: 30-Sep-2022].
- [178] T. H. Minh Le and M. A. Babar. On the Use of Fine-Grained Vulnerable Code Statements for Software Vulnerability Assessment Models. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, pages 621–633, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 978-1-4503-9303-4. doi: 10.1145/3524842.3528433. URL <https://doi.org/10.1145/3524842.3528433>. event-place: Pittsburgh, Pennsylvania.
- [179] T. H. Minh Le, D. Hin, R. Croft, and M. Ali Babar. DeepCVA: Automated Commit-level Vulnerability Assessment with Deep Multi-task Learning. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 717–729, 2021. doi: 10.1109/ASE51524.2021.9678622.
- [180] MITRE. *Common Vulnerabilities and Exposures (CVE) Numbering Authority (CNA) Rules*, 2016. URL https://cve.mitre.org/cve/cna/CNA_Rules_v1.1.pdf. [Online; Accessed: 24-July-2021].
- [181] MITRE. CVE → CWE Mapping Guidance, 2021. URL https://cwe.mitre.org/documents/cwe_usage/guidance.html.
- [182] MITRE. CWE VIEW: Weaknesses in OWASP Top Ten (2021), 2021. URL <https://cwe.mitre.org/data/definitions/1344.html>.
- [183] MITRE. Cwe-1003: Cwe view: Weaknesses for simplified mapping of published vulnerabilities, 2022. URL <https://cwe.mitre.org/data/definitions/1003.html>. [Online; Accessed 07-Jan-2022].
- [184] MITRE. CWE Common Weakness Enumeration, 2022. URL <https://cwe.mitre.org/>.
- [185] P. J. Morrison, R. Pandita, X. Xiao, R. Chillarege, and L. Williams. Are vulnerabilities discovered and resolved like other defects? *Empirical Software Engineering*, 23(3):1383–1421, 2018. Publisher: Springer.
- [186] Mozilla. HTTP messages, 2021. URL <https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages>. Publication Title: MDN Web Docs.

- [187] Mozilla. Package management basics. https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Understanding_client-side_tools/Package_management, May 2023. Online; Accessed: 09-Jun-2023.
- [188] S. Mujahid, R. Abdalkareem, and E. Shihab. What are the characteristics of highly-selected packages? a case study on the npm ecosystem. *Journal of Systems and Software*, 198:111588, 2023. ISSN 0164-1212. doi: <https://doi.org/10.1016/j.jss.2022.111588>. URL <https://www.sciencedirect.com/science/article/pii/S0164121222002643>.
- [189] S. Nadi and N. Sakr. Selecting third-party libraries: the data scientist’s perspective. *Empirical Software Engineering*, 28(1):15, 2023.
- [190] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.
- [191] National Security Agency (NSA). Software memory safety, 2023. URL https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF. Online; Accessed: 22-July-2024.
- [192] K. Nayak, D. Marino, P. Efstathopoulos, and T. Dumitraş. Some Vulnerabilities Are Different Than Others. In A. Stavrou, H. Bos, and G. Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, pages 426–446, Cham, 2014. Springer International Publishing. ISBN 978-3-319-11379-1.
- [193] NIST. Sate vi: Preliminary results, 2019. URL <https://www.nist.gov/system/files/documents/noindex/2021/03/23/13.Results.pdf>. Online; Accessed: 24-Oct-2023.
- [194] NIST. National Vulnerability Database, 2021. URL <https://nvd.nist.gov/>.
- [195] NIST. Computer security resource center (csrc) glossary, 2022. URL <https://csrc.nist.gov/glossary>. [Online; Accessed 24-Mar-2022].
- [196] NIST. General FAQs, 2022. URL <https://nvd.nist.gov/general/FAQ-Sections/General-FAQs>.
- [197] NIST. Vulnerability Metrics, 2022. URL <https://nvd.nist.gov/vuln-metrics/cvss>. [Online; Accessed: 19-Sept-2022].
- [198] NIST. Nvd - vulnerabilities (website), 2023. URL <https://nvd.nist.gov/vuln>.
- [199] NVD. CWE Over Time, 2021. URL <https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cwe-over-time>.
- [200] M. Ohm, H. Plate, A. Sykosch, and M. Meier. Backstabber’s knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*, pages 23–43. Springer, 2020.

- [201] V. Okun, R. Gaucher, and P. E. Black. Static analysis tool exposition (SATE) 2008. NIST SP 500-279, National Institute of Standards and Technology (NIST), 2009. URL <https://dx.doi.org/10.6028/NIST.SP.500-279>.
- [202] V. Okun, A. Delaitre, and P. E. Black. The Second Static Analysis Tool Exposition (SATE) 2009. NIST SP 500-287, National Institute of Standards and Technology (NIST), 2010. URL <https://dx.doi.org/10.6028/NIST.SP.500-287>.
- [203] V. Okun, A. Delaitre, and P. E. Black. Report on the Third Static Analysis Tool Exposition (SATE 2010). NIST SP 500-283, National Institute of Standards and Technology (NIST), 2011. URL <https://dx.doi.org/10.6028/NIST.SP.500-283>.
- [204] V. Okun, A. Delaitre, and P. E. Black. Report on the Static Analysis Tool Exposition (SATE) IV. NIST SP 500-297, National Institute of Standards and Technology (NIST), 2013. URL <https://dx.doi.org/10.6028/NIST.SP.500-297>.
- [205] OpenMRS. *OpenMRS Developer Manual*, 2020. URL <http://devmanual.openmrs.org/en/>. [Online; Accessed: 24-Jul-2021].
- [206] OpenMRS. OpenMRS Atlas, 2021. URL <https://atlas.openmrs.org/>. [Online; Accessed: 24-Jul-2021].
- [207] OpenSSF. About - open source security foundation, 2023. URL <https://openssf.org/about/>. Online; Accessed: 08-Jun-2023.
- [208] OpenSSF. Openssf scorecard readme (Website), 2023. URL <https://github.com/ossf/scorecard/blob/main/README.md>. [Online; Accessed: 30-Oct-2023].
- [209] OpenSSF. Openssf scorecard (Website), 2023. URL <https://securityscorecards.dev>. [Online; Accessed: 30-Sept-2023].
- [210] OSV. Open source vulnerabilities (website), 2024. URL <https://osv.dev/>. Online; Accessed: 22-July-2024.
- [211] OWASP. OWASP Top Ten - 2010, 2013. URL https://owasp.org/www-pdf-archive/OWASP_Top_10_-_2010.pdf.
- [212] OWASP. OWASP Top Ten - 2013, 2013. URL https://owasp.org/www-pdf-archive/OWASP_Top_10_-_2013.pdf.
- [213] OWASP. OWASP Top Ten - 2017, 2017. URL <https://owasp.org/www-project-top-ten/2017/>.
- [214] OWASP. OWASP Top Ten - 2021, 2021. URL <https://owasp.org/Top10/>.
- [215] OWASP. The OWASP Top Ten Application Security Risks Project, 2021. URL <https://owasp.org/www-project-top-ten/>.
- [216] OWASP, editor. *OWASP ZAP Proxy*, 2021. URL <https://www.zaproxy.org/>. [Online; Accessed 01-Nov-2021].

- [217] OWASP ZAP Dev Team. Getting Started - Features - Alerts. In *The OWASP Zed Attack Proxy (ZAP) Desktop User Guide*, 2021. URL <https://www.zaproxy.org/docs/desktop/start/features/alerts/>. [Online; Accessed 06-Dec-2021].
- [218] OWASP ZAP Dev Team. Getting Started - Features - Spider. In *The OWASP Zed Attack Proxy (ZAP) Desktop User Guide*, 2021. URL <https://www.zaproxy.org/docs/desktop/start/features/spider/>. [Online; Accessed 20-Jul-2021].
- [219] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '18*, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450358231. doi: 10.1145/3239235.3268920. URL <https://doi.org/10.1145/3239235.3268920>.
- [220] I. Pashchenko, D.-L. Vu, and F. Massacci. A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, CCS '20*, page 1513–1531, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370899. doi: 10.1145/3372297.3417232. URL <https://doi.org/10.1145/3372297.3417232>.
- [221] M. Pendleton, R. Garcia-Lebron, J.-H. Cho, and S. Xu. A Survey on Systems Security Metrics. *ACM Comput. Surv.*, 49(4), Dec. 2016. ISSN 0360-0300. doi: 10.1145/3005714. URL <https://doi.org/10.1145/3005714>. Place: New York, NY, USA Publisher: Association for Computing Machinery.
- [222] S. Peroni and D. Shotton. OpenCitations, an infrastructure organization for open scholarship. *Quantitative Science Studies*, 1(1):428–444, 2020. Publisher: MIT Press One Rogers Street, Cambridge, MA 02142-1209, USA journals-info
- [223] S. Perugini. Revitalizing the linux programming course with go. *J. Comput. Sci. Coll.*, 35(5):61–69, oct 2019. ISSN 1937-4771.
- [224] A. Peruma and D. Krutz. Understanding the Relationship between Quality and Security: A Large-Scale Analysis of Android Applications. In *2018 IEEE/ACM 1st International Workshop on Security Awareness from Design to Deployment (SEAD)*, 2018.
- [225] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson. Systematic mapping studies in software engineering. In *12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12*, pages 1–10, 2008.
- [226] D. Pfahl, H. Yin, M. V. Mäntylä, and J. Münch. How is exploratory testing used? A state-of-the-practice survey. In *Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement*, page 5. ACM, 2014.
- [227] PGX. pgx readme, 2024. URL <https://github.com/jackc/pgx/blob/masterREADME.md>. Online; Accessed: 22-July-2024.

- [228] R. Pike. Go at google. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12*, page 5–6, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315630. doi: 10.1145/2384716.2384720. URL <https://doi.org/10.1145/2384716.2384720>.
- [229] H. Plate, S. E. Ponta, and A. Sabetta. Impact assessment for vulnerabilities in open-source software libraries. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 411–420, 2015.
- [230] S. E. Ponta, H. Plate, and A. Sabetta. Beyond Metadata: Code-Centric and Usage-Based Analysis of Known Vulnerabilities in Open-Source Software. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 449–460, 2018. doi: 10.1109/ICSME.2018.00054.
- [231] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont. A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 383–387, 2019. doi: 10.1109/MSR.2019.00064.
- [232] S. E. Ponta, H. Plate, and A. Sabetta. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, 25(5):3175–3215, 2020. Publisher: Springer.
- [233] A. Prasertsang and D. Pradubsuwun. Formal verification of concurrency in go. In *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 1–4, 2016. doi: 10.1109/JCSSE.2016.7748882.
- [234] S. Purkayastha, S. Goyal, T. Phillips, H. Wu, B. Haakenson, and X. Zou. Continuous Security through Integration Testing in an Electronic Health Records System. In *2020 International Conference on Software Security and Assurance (ICSSA)*, pages 26–31. IEEE, 2020.
- [235] H. S. Qiu, Y. L. Li, S. Padala, A. Sarma, and B. Vasilescu. The signals that potential contributors look for when choosing open-source projects. 3(CSCW), nov 2019. doi: 10.1145/3359224. URL <https://doi.org/10.1145/3359224>.
- [236] Qualtrics. Qualtrics (website), 2024. URL <https://www.qualtrics.com/>. Online; Accessed: 22-July-2024.
- [237] Radio New Zealand (RNZ). Health Ministry announces \$75m to plug cybersecurity gaps, 2021. URL <https://www.rnz.co.nz/news/national/458331/health-ministry-announces-75m-to-plug-cybersecurity-gaps>. [Online; Accessed: 21-Dec-2021].
- [238] A. A. U. Rahman, E. Helms, L. Williams, and C. Parnin. Synthesizing continuous deployment practices used in software development. In *2015 Agile Conference*, pages 1–10. IEEE, 2015.

- [239] S. M. Rajasooriya, C. P. Tsokos, and P. K. Kaluarachchi. Stochastic Modelling of Vulnerability Life Cycle and Security Risk Evaluation. *Journal of information Security*, 7(4):269–279, 2016. Publisher: Scientific Research Publishing.
- [240] S. M. Rajasooriya, C. P. Tsokos, and P. K. Kaluarachchi. Cyber security: Nonlinear stochastic models for predicting the exploitability. *Journal of information Security*, 8(2):125–140, 2017. Publisher: Scientific Research Publishing.
- [241] P. Ralph and E. Tempero. Construct validity in software engineering research and software metrics. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, pages 13–23, 2018.
- [242] N. M. Razali, Y. B. Wah, and others. Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. *Journal of statistical modeling and analytics*, 2(1):21–33, 2011.
- [243] G. Reid, P. Mell, and K. Scarfone. CVSS-SIG Version 2 History. Technical report, Forum of Incident Response and Security Teams (FIRST), 2007. URL <https://www.first.org/cvss/v2/history>.
- [244] D. Repel, J. Kinder, and L. Cavallaro. Modular Synthesis of Heap Exploits. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security, PLAS '17*, pages 25–35, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5099-0.
- [245] Y. Roumani and J. Nwankpa. Examining exploitability risk of vulnerabilities: a hazard model. *Communications of the Association for Information Systems*, 46(1):18, 2020.
- [246] C. Sabottke, O. Suciu, and T. Dumitraş. Vulnerability disclosure in the age of social media: exploiting twitter for predicting real-world exploits. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 1041–1056, 2015.
- [247] N. A. M. Saffie, N. M. Shukor, and K. A. Rasmani. Fuzzy delphi method: Issues and challenges. In *2016 International Conference on Logistics, Informatics and Service Sciences (LISS)*, pages 1–7, 2016. doi: 10.1109/LISS.2016.7854490.
- [248] R. Scandariato, J. Walden, and W. Joosen. Static analysis versus penetration testing: A controlled experiment. In *2013 IEEE 24th international symposium on software reliability engineering (ISSRE)*, pages 451–460. IEEE, 2013.
- [249] T. Scanlon. 10 Types of Application Security Testing Tools: When and How to Use Them. Blog, Software Engineering Institute, Carnegie Mellon University, July 2018. URL <https://insights.sei.cmu.edu/blog/10-types-of-application-security-testing-tools-when-and-how-to-use-them>.
- [250] K. Scarfone and P. Mell. An analysis of CVSS version 2 vulnerability scoring. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 516–525, 2009. doi: 10.1109/ESEM.2009.5314220.

- [251] F. Schmager, N. Cameron, and J. Noble. Gohotdraw: evaluating the go programming language with design patterns. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450305471. doi: 10.1145/1937117.1937127. URL <https://doi.org/10.1145/1937117.1937127>.
- [252] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit Hardening Made Easy. In *20th USENIX Security Symposium (USENIX Security 11)*, San Francisco, CA, Aug. 2011. USENIX Association.
- [253] SCImago. SCImago Journal & Country Rank [Portal]., 2023. URL <https://www.scimagojr.com>.
- [254] SciTools. What metrics does understand have?, 2021. URL <https://support.scitools.com/support/solutions/articles/70000582223-what-metrics-does-understand-have->. Online; Accessed: 27-Oct-2022.
- [255] Semantic Scholar. Semantic scholar academic graph api, 2023. URL <https://www.semanticscholar.org/product/api>. [Online; Accessed: 18-July-2023].
- [256] R. Sen and G. R. Heim. Managing enterprise risks of technological systems: An exploratory empirical analysis of vulnerability characteristics as drivers of exploit publication. *Decision Sciences*, 47(6):1073–1102, 2016. Publisher: Wiley Online Library.
- [257] J. Sheoran, K. Blincoe, E. Kalliamvakou, D. Damian, and J. Ell. Understanding "watchers" on github. In *Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014*, page 336–339, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328630. doi: 10.1145/2597073.2597114. URL <https://doi.org/10.1145/2597073.2597114>.
- [258] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.
- [259] S. d. Smale, R. v. Dijk, X. Bouwman, J. v. d. Ham, and M. v. Eeten. No One Drinks From the Firehose: How Organizations Filter and Prioritize Vulnerability Information. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1980–1996, Los Alamitos, CA, USA, May 2023. IEEE Computer Society. doi: 10.1109/SP46215.2023.00012. URL <https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00012>.
- [260] B. Smith and L. Williams. On the effective use of security test patterns. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 108–117. IEEE, 2012.
- [261] B. Smith and L. A. Williams. Systematizing security test planning using functional requirements phrases. Technical Report TR-2011-5, North Carolina State University. Dept. of Computer Science, 2011.

- [262] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 248–259. ACM, 2015.
- [263] J. Smith, L. N. Q. Do, and E. Murphy-Hill. Why Can't Johnny Fix Vulnerabilities: A Usability Evaluation of Static Analysis Tools for Security. In *Sixteenth Symposium on Usable Privacy and Security ({SOUPS} 2020)*, pages 221–238. USENIX, 2020.
- [264] Snyk. Software dependencies: How to manage dependencies at scale, 2023. URL <https://snyk.io/series/open-source-security/software-dependencies/>. Online; Accessed: 08-Jun-2023.
- [265] SonarSource. SonarQube Documentation: Security-related Rules, 2019. URL <https://docs.sonarqube.org/8.2/user-guide/security-rules/>. [Online; Accessed: 06-Dec-2021].
- [266] E. Sotos-Martínez, N. M. Villanueva, and L. A. Orellana. A Survey on the State of the Art of Vulnerability Assessment Techniques. In J. J. Gude Prego, J. G. de la Puerta, P. García Bringas, H. Quintián, and E. Corchado, editors, *14th International Conference on Computational Intelligence in Security for Information Systems and 12th International Conference on European Transnational Educational (CISIS 2021 and ICEUTE 2021)*, pages 203–213, Cham, 2021. Springer International Publishing. ISBN 978-3-030-87872-6.
- [267] G. Spanos and L. Angelis. A multi-target approach to estimate software vulnerability characteristics and severity scores. *Journal of Systems and Software*, 146:152–166, 2018. ISSN 0164-1212. doi: <https://doi.org/10.1016/j.jss.2018.09.039>. URL <https://www.sciencedirect.com/science/article/pii/S0164121218302061>.
- [268] G. Spanos, A. Sioziou, and L. Angelis. WIVSS: A New Methodology for Scoring Information Systems Vulnerabilities. In *Proceedings of the 17th Panhellenic Conference on Informatics, PCI '13*, pages 83–90, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 978-1-4503-1969-0. doi: 10.1145/2491845.2491871. URL <https://doi.org/10.1145/2491845.2491871>. event-place: Thessaloniki, Greece.
- [269] StackOverflow. 2021 Developer Survey, 2021. URL <https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies>. [Online; Accessed: 07-Dec-2021].
- [270] O. Suciú, C. Nelson, Z. Lyu, T. Bao, and T. Dumitras. Expected Exploitability: Predicting the Development of Functional Vulnerability Exploits. In *31st USENIX Security Symposium*, pages 377–394, Boston, MA, 2022. USENIX Association. ISBN 978-1-939133-31-1.
- [271] D. Toloudis, G. Spanos, and L. Angelis. Associating the Severity of Vulnerabilities with their Description. In J. Krogstie, H. Mouratidis, and J. Su, editors, *Advanced Information Systems Engineering Workshops*, pages 231–242, Cham, 2016. Springer International Publishing.

- [272] TrendMicro. Trendmicro: exploit, 2023. URL <https://www.trendmicro.com/vinfo/us/security/definition/exploit>. [Online; Accessed: 17-May-2023].
- [273] S. Tripathi, G. Grieco, and S. Rawat. Exniffer: Learning to Prioritize Crashes by Assessing the Exploitability from Memory Dump. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 239–248, 2017.
- [274] A. Trockman, S. Zhou, C. Kästner, and B. Vasilescu. Adding sparkle to social coding: An empirical study of repository badges in the npm ecosystem. In *Proceedings of the 40th International Conference on Software Engineering, ICSE '18*, page 511–522, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. doi: 10.1145/3180155.3180209. URL <https://doi.org/10.1145/3180155.3180209>.
- [275] I. A. Tøndel, M. G. Jaatun, D. S. Cruzes, and L. Williams. Collaborative security risk estimation in agile software development. *Information & Computer Security*, 2019. Publisher: Emerald Publishing Limited.
- [276] U.S. Cybersecurity and Infrastructure Security Agency (CISA). Provide medical care is in critical condition: Analysis and stakeholder decision support to minimize further harm, Sept. 2021. URL https://www.cisa.gov/sites/default/files/publications/Insights_MedicalCare_FINAL-v2_0.pdf. [Online; Accessed: 21-Dec-2021].
- [277] US Dept of Veterans Affairs, Office of Information and Technology, Enterprise Program Management Office. VA Monograph, Apr. 2021. URL https://www.va.gov/vd1/documents/Monograph/Monograph/VistA_Monograph_0421_REDACTED.pdf.
- [278] A. van der Stock, D. Cuthbert, J. Manico, J. C. Grossman, and M. Burnett. Application Security Verification Standard. Rev. 4.0.1, Open Web Application Security Project (OWASP), Mar. 2019. URL <https://github.com/OWASP/ASVS/tree/v4.0.1/4.0>.
- [279] M. van Haastrecht, I. Sarhan, B. Yigit Ozkan, M. Brinkhuis, and M. Spruit. SYMBALS: A systematic review methodology blending active learning and snowballing. *Frontiers in research metrics and analytics*, 6:685591, 2021. Publisher: Frontiers Media SA.
- [280] D. Votipka, R. Stevens, E. Redmiles, J. Hu, and M. Mazurek. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 374–391. IEEE, 2018.
- [281] D. Walkowski. f5: Threats, Vulnerabilities, Exploits and Their Relationship to Risk, 2021. URL <https://www.f5.com/labs/learning-center/threats-vulnerabilities-exploits-and-their-relationship-to-risk>.
- [282] C. Wang, J. Gao, Y. Jiang, Z. Xing, H. Zhang, W. Yin, M. Gu, and J. Sun. Go-clone: graph-embedding based clone detector for golang. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019*, page 374–377, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362245. doi: 10.1145/3293882.3338996. URL <https://doi.org/10.1145/3293882.3338996>.

- [283] C. Wang, H. Sun, Y. Xu, Y. Jiang, H. Zhang, and M. Gu. Go-sanitizer: Bug-oriented assertion generation for go. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 36–41, 2019. doi: 10.1109/ISSREW.2019.00039.
- [284] Y. Wang, F. Yang, and Q. Sun. Measuring Network Vulnerability Based on Pathology. In *2008 The Ninth International Conference on Web-Age Information Management*, pages 640–646, 2008. doi: 10.1109/WAIM.2008.66.
- [285] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou. Revery: From Proof-of-Concept to Exploitable. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, pages 1914–1927, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5693-0.
- [286] Y. Wang, W. Wu, C. Zhang, X. Xing, X. Gong, and W. Zou. From proof-of-concept to exploitable. *Cybersecurity*, 2(1):1–25, 2019. Publisher: SpringerOpen.
- [287] D. Wermke, J. H. Klemmer, N. Wöhler, J. Schmüser, H. S. Ramulu, Y. Acar, and S. Fahl. “always contribute back”: A qualitative study on security challenges of the open source supply chain. In *2023 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 1545–1560, Los Alamitos, CA, USA, may 2023. IEEE Computer Society. doi: 10.1109/SP46215.2023.00191. URL <https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00191>.
- [288] R. R. Wilcox and H. Keselman. Modern robust data analysis methods: measures of central tendency. *Psychological methods*, 8(3):254, 2003. Publisher: American Psychological Association.
- [289] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.
- [290] T. Wood, S. Perera, S. Yan, L. Padgha, and A. Moffat. CORE: Conference Details, 2022. URL https://www.core.edu.au/conference-portal#h.p_ID_44.
- [291] Q. Wu, Y. Xiao, X. Liao, and K. Lu. {OS-Aware} Vulnerability Prioritization via Differential Severity Analysis. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 395–412, 2022.
- [292] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 781–797, Baltimore, MD, Aug. 2018. USENIX Association. ISBN 978-1-939133-04-5.
- [293] W. Wu, Y. Chen, X. Xing, and W. Zou. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1187–1204, Santa Clara, CA, 2019. USENIX Association. ISBN 978-1-939133-06-9.

- [294] C. Xiao, A. Sarabi, Y. Liu, B. Li, M. Liu, and T. Dumitras. From Patching Delays to Infection Symptoms: Using Risk Profiles for an Early Discovery of Vulnerabilities Exploited in the Wild. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 903–918, Baltimore, MD, 2018. USENIX Association. ISBN 978-1-939133-04-5.
- [295] B. Xu, L. An, F. Thung, F. Khomh, and D. Lo. Why reinventing the wheels? an empirical study on library reuse and re-implementation. *Empirical Software Engineering*, 25: 755–789, 2020.
- [296] Y. Yamamoto, D. Miyamoto, and M. Nakayama. Text-Mining Approach for Estimating Vulnerability Score. In *2015 4th International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, pages 67–73, 2015. doi: 10.1109/BADGERS.2015.018.
- [297] G. Yan, J. Lu, Z. Shu, and Y. Kucuk. ExploitMeter: Combining Fuzzing with Machine Learning for Automated Evaluation of Software Exploitability. In *2017 IEEE Symposium on Privacy-Aware Computing (PAC)*, pages 164–175, 2017. doi: 10.1109/PAC.2017.10.
- [298] H. Yang, S. Park, K. Yim, and M. Lee. Better Not to Use Vulnerability’s Reference for Exploitability Prediction. *Applied Sciences*, 10(7):2555, 2020. Publisher: Multidisciplinary Digital Publishing Institute.
- [299] F. Yilmaz, M. Sridhar, and W. Choi. Guide Me to Exploit: Assisted ROP Exploit Generation for ActionScript Virtual Machine. In *Annual Computer Security Applications Conference*, pages 386–400, New York, USA, 2020. ACM. ISBN 978-1-4503-8858-0.
- [300] A. Younis, Y. Malaiya, C. Anderson, and I. Ray. To Fear or Not to Fear That is the Question: Code Characteristics of a Vulnerable Function with an Existing Exploit. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 97–104, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3935-3.
- [301] A. Younis, Y. K. Malaiya, and I. Ray. Assessing vulnerability exploitability risk using software properties. *Software Quality Journal*, 24(1):159–202, 2016. Publisher: Springer.
- [302] A. A. Younis and Y. K. Malaiya. Using Software Structure to Predict Vulnerability Exploitation Potential. In *2014 IEEE Eighth International Conference on Software Security and Reliability-Companion*, pages 13–18, 2014.
- [303] A. A. Younis and Y. K. Malaiya. Comparing and Evaluating CVSS Base Metrics and Microsoft Rating System. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 252–261, 2015.
- [304] A. A. Younis, Y. K. Malaiya, and I. Ray. Using Attack Surface Entry Points and Reachability Analysis to Assess the Risk of Software Vulnerability Exploitability. In *IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 1–8, 2014.
- [305] Z. Yu and T. Menzies. FAST2: An intelligent assistant for finding relevant papers. *Expert Systems with Applications*, 120:57–71, 2019. ISSN 0957-4174.

- [306] Z. Yu, N. A. Kraft, and T. Menzies. Finding better active learners for faster literature reviews. *Empirical Software Engineering*, 23:3161–3186, 2018. Publisher: Springer.
- [307] N. Zahan, P. Kanakiya, B. Hambleton, S. Shohan, and L. Williams. Openssf scorecard: On the path toward ecosystem-wide automated security metrics. *IEEE Security & Privacy*, 21(6):76–88, 2023. doi: 10.1109/MSEC.2023.3279773.
- [308] N. Zahan, S. Shohan, D. Harris, and L. Williams. Do software security practices yield fewer vulnerabilities? In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 292–303, 2023. doi: 10.1109/ICSE-SEIP58684.2023.00032.
- [309] K. Zeng, Y. Chen, H. Cho, X. Xing, A. Doupé, Y. Shoshitaishvili, and T. Bao. Playing for {K (H) eaps}: Understanding and improving linux kernel exploit reliability. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 71–88, 2022.
- [310] A. Zerouali, T. Mens, A. Decan, and C. De Roover. On the impact of security vulnerabilities in the npm and rubygems dependency networks. *Empirical Software Engineering*, 27(5): 107, 2022.
- [311] F. Zhang and Q. Li. Dynamic Risk-Aware Patch Scheduling. In *2020 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9, 2020.
- [312] Z. Zhao, Y. Wang, and X. Gong. HAEPG: An Automatic Multi-hop Exploitation Generation Framework. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 89–109, Cham, 2020. Springer. ISBN 978-3-030-52683-2.
- [313] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In *19th Annual Network and Distributed System Security (NDSS) Symposium*, 2012.
- [314] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *USENIX security symposium*, volume 17, 2019.
- [315] D. Zou, J. Yang, Z. Li, H. Jin, and X. Ma. Autocvss: An approach for automatic assessment of vulnerability severity based on attack process. In *International Conference on Green, Pervasive, and Cloud Computing*, pages 238–253. Springer, 2019.

APPENDICES

APPENDIX

A

ACRONYMS

A summary of common acronyms is documented in Table A.1.

Table A.1: A summary of acronyms used in alphabetical order.

Acronym	Abbreviation
Bi-directional Long Short Term Memory	BiLSTM
Convolutional Neural Network	CNN
Dynamic Analysis Security Testing	DAST
Decision Tree	DT
Gradient Boosting Machine	GBM
Gated Recurrent Unit	GRU
K-Nearest Neighbor	KNN
Latent Dirichlet Allocation	LDA
Light Gradient Boosting Machine	LGBM
Logistic Regression	LR
Long Short-Term Memory	LSTM
LSTM based Artificial Neural Network	LSTM-ANN
Multi-Layer Perception	MLP
Naïve Bayes	NB

Table A.1 (Continued)

(US) National Institute of Standards and Technology	NIST
National Vulnerability Database	NVD
Open Source Security Foundation	OpenSSF
Random Forest	RF
Static Analysis Security Testing	SAST
Supervised Latent Dirichlet Allocation	SLDA
Support Vector Machine	SVM
Extreme Gradient Boosting	XGBoost

APPENDIX

B

VULNERABILITY DETECTION TECHNIQUE COMPARISON - REPLICATION DETAILS

In this Appendix we include details which may be useful for some readers or necessary for an exact replication of our vulnerability detection comparison in Chapter 3, but which may detract from the primary report, such as with large, multi-page tables.

B.1 Appendix - Automated Technique CWEs

Table B.1: CWEs covered in the rules implemented by automated techniques

CWE ID	CWE Name	OWASP Top Ten	ZAP	DA2	Sonar	SA2
22	Path Traversal	A01	X	X		X
23	Relative Path Traversal	A01		X		
200	Exposure of Sensitive Information to an Unauthorized Actor	A01	X			X

Table B.1 (Continued)

ID	Name	Top Ten	ZAP	DA2	Sonar	SA2
201	Insertion of Sensitive Information Into Sent Data	A01				X
264	Permissions, Privileges, and Access Controls	A01	X			
284	Improper Access Control	A01	X			X
285	Improper Authorization	A01				X
352	Cross-Site Request Forgery (CSRF)	A01	X	X	X	X
359	Exposure of Private Personal Information to an Unauthorized Actor	A01				X
425	Forced Browsing	A01				X
601	Open Redirect	A01	X			X
668	Exposure of Resource to Wrong Sphere	A01				X
862	Missing Authorization	A01				X
863	Incorrect Authorization	A01				X
1275	Sensitive Cookie with Improper SameSite Attribute	A01	X			
296	Improper Following of a Certificate's Chain of Trust	A02				X
319	Cleartext Transmission of Sensitive Information	A02				X
321	Use of Hard-coded Cryptographic Key	A02		X		X
322	Key Exchange without Entity Authentication	A02		X		
325	Missing Cryptographic Step	A02		X		
326	Inadequate Encryption Strength	A02	X	X	X	
327	Use of a Broken or Risky Cryptographic Algorithm	A02			X	
328	Use of Weak Hash	A02			X	
330	Use of Insufficiently Random Values	A02		X	X	X

Table B.1 (Continued)

ID	Name	Top Ten	ZAP	DA2	Sonar	SA2
336	Same Seed in Pseudo-Random Number Generator (PRNG)	A02			X	X
337	Predictable Seed in Pseudo-Random Number Generator (PRNG)	A02			X	X
523	Unprotected Transport of Credentials	A02		X		
760	Use of a One-Way Hash with a Predictable Salt	A02				X
916	Use of Password Hash With Insufficient Computational Effort	A02				X
20	Improper Input Validation	A03				X
74	Injection	A03				X
78	OS Command Injection	A03	X	X		X
79	Cross-site Scripting	A03	X	X		X
83	Improper Neutralization of Script in Attributes in a Web Page	A03				X
88	Argument Injection	A03				X
89	SQL Injection	A03	X	X		X
90	LDAP Injection	A03		X		X
91	XML Injection (aka Blind XPath Injection)	A03		X		
93	CRLF Injection	A03	X			
94	Code Injection	A03	X			X
95	Eval Injection	A03				X
97	Improper Neutralization of Server-Side Includes (SSI) Within a Web Page	A03	X			
98	PHP Remote File Inclusion	A03	X			
99	Resource Injection	A03				X
113	HTTP Response Splitting	A03				X
184	Incomplete List of Disallowed Inputs	A03				X
470	Unsafe Reflection	A03				X

Table B.1 (Continued)

ID	Name	Top Ten	ZAP	DA2	Sonar	SA2
610	Externally Controlled Reference to a Resource in Another Sphere	A03				X
643	XPath Injection	A03				X
917	Expression Language Injection	A03				X
73	External Control of File Name or Path	A04				X
183	Permissive List of Allowed Inputs	A04				X
209	Generation of Error Message Containing Sensitive Information	A04		X		X
311	Missing Encryption of Sensitive Data	A04	X			
313	Cleartext Storage in a File or on Disk	A04				X
472	External Control of Assumed-Immutable Web Parameter	A04	X			
501	Trust Boundary Violation	A04				X
522	Insufficiently Protected Credentials	A04			X	
525	Use of Web Browser Cache Containing Sensitive Info.	A04	X			
642	External Control of Critical State Data	A04	X			X
646	Reliance on File Name or Extension of Externally-Supplied File	A04				X
650	Trusting HTTP Permission Methods on the Server Side	A04				X
770	Allocation of Resources Without Limits or Throttling	A04				X
807	Reliance on Untrusted Inputs in a Security Decision	A04			X	
927	Use of Implicit Intent for Sensitive Communication	A04				X
1021	Improper Restriction of Rendered UI Layers or Frames	A04	X			

Table B.1 (Continued)

ID	Name	Top Ten	ZAP	DA2	Sonar	SA2
7	J2EE Misconfiguration: Missing Custom Error Page	A05				X
315	Cleartext Storage of Sensitive Information in a Cookie	A05				X
541	Inclusion of Sensitive Information in an Include File	A05	X			
548	Exposure of Information Through Directory Listing	A05	X			
611	Improper Restriction of XML External Entity Reference	A05			X	X
614	Sensitive Cookie in HTTPS Session Without 'Secure' Attribute	A05	X			X
776	XML Entity Expansion	A05				X
933	Security Misconfiguration	A05	X			
942	Permissive Cross-domain Policy with Untrusted Domains	A05				X
1004	Sensitive Cookie Without 'HttpOnly' Flag	A05	X			X
259	Use of Hard-coded Password	A07				X
263	Password Aging with Long Expiration	A07				X
287	Improper Authentication	A07		X		X
288	Authentication Bypass Using an Alternate Path or Channel	A07				X
295	Improper Certificate Validation	A07		X	X	X
297	Improper Validation of Certificate with Host Mismatch	A07				X
300	Channel Accessible by Non-Endpoint	A07				X
307	Improper Restriction of Excessive Authentication Attempts	A07				X
346	Origin Validation Error	A07				X

Table B.1 (Continued)

ID	Name	Top Ten	ZAP	DA2	Sonar	SA2
384	Session Fixation	A07				X
521	Weak Password Requirements	A07			X	X
613	Insufficient Session Expiration	A07				X
798	Use of Hard-coded Credentials	A07				X
345	Insufficient Verification of Data Authenticity	A08	X			X
502	Deserialization of Untrusted Data	A08			X	X
565	Reliance on Cookies without Validation and Integrity Checking	A08	X			X
829	Inclusion of Functionality from Untrusted Control Sphere	A08	X			X
915	Improperly Controlled Modification of Dynamically-Determined Object Attributes	A08			X	X
532	Insertion of Sensitive Information into Log File	A09				X
778	Insufficient Logging	A09				X
4	J2EE Environment Issues (Deprecated)	NM				X
36	Absolute Path Traversal	NM		X		
41	Improper Resolution of Path Equivalence	NM		X		
67	Improper Handling of Windows Device Names	NM		X		
102	Struts: Duplicate Validation Forms	NM			X	
112	Missing XML Validation	NM		X		
118	Range Error	NM		X		
120	Buffer Overflow	NM	X	X		
124	Buffer Underflow	NM		X		
134	Use of Externally-Controlled Format String	NM	X	X		

Table B.1 (Continued)

ID	Name	Top Ten	ZAP	DA2	Sonar	SA2
140	Improper Neutralization of Delimiters	NM		X		
144	Improper Neutralization of Line Delimiters	NM		X		
149	Improper Neutralization of Quoting Syntax	NM		X		
150	Improper Neutralization of Escape, Meta, or Control Sequences	NM		X		
154	Improper Neutralization of Variable Name Delimiters	NM		X		
156	Improper Neutralization of Whitespace	NM		X		
157	Failure to Sanitize Paired Delimiters	NM		X		
158	Improper Neutralization of Null Byte or NUL Character	NM		X		
166	Improper Handling of Missing Special Element	NM		X		
172	Encoding Error	NM		X		
174	Double Decoding of the Same Data	NM		X		
175	Improper Handling of Mixed Encoding	NM		X		
176	Improper Handling of Unicode Encoding	NM		X		
177	Improper Handling of URL Encoding (Hex Encoding)	NM		X		
185	Incorrect Regular Expression	NM		X		X
189	Numeric Error	NM		X		
190	Integer Overflow or Wraparound	NM				X
194	Unexpected Sign Extension	NM		X		
215	Insertion of Sensitive Information Into Debugging Code	NM				X
242	Use of Inherently Dangerous Function	NM				X
252	Unchecked Return Value	NM				X

Table B.1 (Continued)

ID	Name	Top Ten	ZAP	DA2	Sonar	SA2
253	Incorrect Check of Function Return Value	NM				X
289	Authentication Bypass by Alternate Name	NM				X
299	Improper Check for Certificate Revocation	NM				X
314	Cleartext Storage in the Registry	NM				X
317	Cleartext Storage of Sensitive Info. in GUI	NM				X
332	Insufficient Entropy in PRNG	NM			X	
366	Race Condition within a Thread	NM				X
369	Divide By Zero	NM				X
390	Detection of Error Condition Without Action	NM				X
398	Code Quality	NM				X
399	Resource Management Error	NM		X		
400	Uncontrolled Resource Consumption	NM				X
403	File Descriptor Leak	NM				X
404	Improper Resource Shutdown or Release	NM				X
406	Insufficient Control of Network Message Volume (Network Amplification)	NM		X		
427	Uncontrolled Search Path Element	NM				X
436	Interpretation Conflict	NM	X			
476	NULL Pointer Dereference	NM				X
480	Use of Incorrect Operator	NM				X
483	Incorrect Block Delimitation	NM				X
484	Omitted Break Statement in Switch	NM				X
489	Active Debug Code	NM			X	

Table B.1 (Continued)

ID	Name	Top Ten	ZAP	DA2	Sonar	SA2
493	Critical Public Variable Without Final Modifier	NM			X	
500	Public Static Field Not Marked Final	NM			X	
543	Use of Singleton Pattern Without Synchronization in a Multithreaded Context	NM				X
561	Dead Code	NM				X
563	Assignment to Variable without Use	NM				X
567	Unsynchronized Access to Shared Data in a Multithreaded Context	NM				X
568	finalize() Method Without super.finalize()	NM				X
569	Expression Issue	NM				X
573	Improper Following of Specification by Caller	NM				X
580	clone() Method Without super.clone()	NM				X
582	Array Declared Public, Final, and Static	NM			X	
599	Missing Validation of OpenSSL Certificate	NM				X
600	Uncaught Exception in Servlet	NM			X	
607	Public Static Final Field References Mutable Object	NM			X	
615	Inclusion of Sensitive Information in Source Code Comments	NM				X
628	Function Call with Incorrectly Specified Arguments	NM				X
661	Weaknesses in Software Written in PHP	NM				X
665	Improper Initialization	NM				X
670	Always-Incorrect Control Flow Implementation	NM				X

Table B.1 (Continued)

ID	Name	Top Ten	ZAP	DA2	Sonar	SA2
683	Function Call With Incorrect Order of Arguments	NM				X
688	Function Call With Incorrect Variable or Reference as Argument	NM				X
693	Protection Mechanism Failure	NM	X			
704	Incorrect Type Conversion or Cast	NM				X
754	Improper Check for Unusual or Exceptional Conditions	NM			X	
755	Improper Handling of Exceptional Conditions	NM				X
777	Regular Expression without Anchors	NM				X
779	Logging of Excessive Data	NM				X
783	Operator Precedence Logic Error	NM				X
827	Improper Control of Document Type Definition	NM			X	X
833	Deadlock	NM				X
835	Infinite Loop	NM				X
1022	Use of Web Link to Untrusted Target with window.opener Access	NM			X	X
1023	Incomplete Comparison with Missing Factors	NM				X

B.2 Appendix - Student Experience Questionnaire

At the beginning of the course, students were asked to fill out a survey about their experience relevant to the course. The four questions asked to students were as follows:

1. How much time have you spent working at a professional software organization – including internships – in terms of the # of years and the # of months?
2. On a scale from 1 (none) to 5 (fully), how much of the time has your work at a professional software organization involved cybersecurity?
3. Which of the follow classes have you already completed?
4. Which of the following classes are you currently taking?

Q1 was short answer. For Q2, students selected a single number between 1 and 5. For Q3, the students could check any number of checkboxes corresponding to a list of the security and privacy courses offered at the institution. For Q4, the students selected from the subset of classes from question 4 that were being offered the semester in which the survey was given.

Fifty-nine of the sixty-three students who agreed to let their data be used for the study responded to the survey. Of these 59 responses, four students responses to Q1 provided a numeric value, e.g. “3”, but did not specify whether the numeric value indicated years or months. We considered this invalid and summarize experience from the remaining 55 participants in Section 3.5.2

B.3 Appendix - Student Assignments

The following are the verbatim assignments for the Course Project that guided the tasks performed by students. We have removed sections of the assignment that are not relevant to this project. Additionally, information that is specific to the tools used, such as UI locations, has also been removed. Text that has been removed is indicated by square brackets [].

B.3.1 Project Part 1

Throughout the course of this semester, you will perform and document a technical security review of OpenMRS (<http://openmrs.org>). This open-source systems provides electronic health care functionality for “resource-constrained environments”. While the system has not been designed for deployment within the United States, security and privacy concerns are still a paramount security concern for any patient.

Software:

OpenMRS 2.9.0. There is no need to install OpenMRS. You will use the VCL image CSC515_SoftwareSecurity_Ubuntu.

Deliverables:

Submit a PDF with all deliverables in Gradescope. Only one submission should be performed per team. Do not include your names/IDs/team name on the report to facilitate the peer evaluation of your assignment (see Part 3 of this assignment).

1. Security test planning and execution (45 points)

a. Record how much total time (hours and minutes) your team spends to complete this activity (test planning and test execution). Compute a metric of how many true positive defects you found per hour of total effort.

b. **Test planning.** Create 15 black box test cases to start a repeatable black box test plan for the OpenMSR (Version 2.9). You may find the OWASP Testing Guide and OWASP Proactive Controls helpful references in addition to the references provided throughout the ASVS document.

For each test case, you must specify:

- A unique test case id that maps to the ASVS, sticking to Level 1 and Level 2. Provide the name/description of the ASVS control. Only one unique identifier is needed (as opposed to the example in the lecture slides). The ASVS number should be part of the one unique identifier.
- Detailed and repeatable (the same steps could be done by anyone who reads the instructions) instructions for how to execute the test case
- Expected results when running the test case. A passing test case would indicate a secure system.
- Actual results of running the test case.

- Indicate the CWE (number and name) for the vulnerability you are testing for.

In choosing your test cases, we are looking for you to demonstrate your understanding of the vulnerability and what it would take to stress the system to see if the vulnerability exists. You may have only one test case per ASVS control.

c. Extra credit (up to 5 points): Create a black box test case that will reveal the vulnerability reported by the static analysis tool (Part 2 of this assignment) for up to 5 vulnerabilities (1 point per vulnerability). Provide the tool output (screen shot of the alert) from each tool.

2. Static analysis (45 points)

a. Record how much total time (hours and minutes) your team spends to complete this activity (test planning and test execution). Compute a metric of how many defects you found per hour of total effort.

b. For each of the three tools (below), review the security reports. Based upon these reports:

- References:
 - Troubleshooting VCL
 - Opening OpenMRS on VCL
- Randomly choose 10 security alerts and provide a cross-reference back to the originating report(s) where the alert was documented. Explore the code to determine if the alert is a false positive or a true positive. The alerts analyzed MUST be security alerts even though the tools will report “regular quality” alerts – you need to choose security alerts.
- If the alert is a false positive, explain why. If you have more than 5 false positives, keep choosing alerts until you have 5 true positives while still reporting the false positives (which may make you go above a total of 10).
- If the alert is a true positive, (1) explain how to fix the vulnerability; (2) map the vulnerability to a CWE; (3) map the vulnerability to the ASVS control.
- Find the instructions for getting [SAST-3] going on OpenMRS here[[hyperlink removed](#)]. [Tool-specific instructions]
- Find the instructions for getting [SAST-2] going on OpenMRS here[[hyperlink removed](#)]. [Tool-specific instructions]

c. Extra credit (up to 5 points): Find 5 instances (1 point per instance) of a potential vulnerability being reported by multiple tools. Provide the tool output (screen shot of the alert) from each tool. Explore the code to determine if the alert is a false positive or a true positive. If the alert is a false positive, explain why. If the alert is a true positive, explain how to fix the vulnerability.

3. Peer evaluation (10 points)

Perform a peer evaluation on another team. Produce a complete report of feedback for the other team using this rubric [to be supplied]. **Note: For any part of this course-long project, you may not directly copy materials from other sources. You need to adapt and make unique**

to OpenMRS. You should provide references to your sources. Copying materials without attribution is plagiarism and will be treated as an academic integrity violation.

B.3.2 Project Part 2

The fuzzing should be performed on the VCL Class Image (“CSC 515 Software Security Ubuntu”).

0. Black Box Test Cases

Parts 1 (OWASP ZAP) and 2 ([DAST-2]) ask for you to write a black box test case. We use the same format as was used in Project Part. For each test case, you must specify:

- A unique test case id that maps to the ASVS, sticking to Level 1 and Level 2. Provide the name/description of the ASVS control. Only one unique identifier is needed (as opposed to the example in the lecture slides). The ASVS number should be part of the one unique identifier.
- Detailed and repeatable (the same steps could be done by anyone who reads the instructions) instructions for how to execute the test case
- Expected results when running the test case. A passing test case would indicate a secure system.
- Actual results of running the test case.
- Indicate the CWE (number and name) for the vulnerability you are testing for.

1. OWASP ZAP (30 points, 3 points for each of the 5 test cases in the two parts)

Client-side bypassing

- Record how much total time (hours and minutes) your team spends to complete this activity. Provide:
 - Total time to plan and run the 5 black box test cases.
 - Total number of vulnerabilities found.
- Plan 5 black box test cases (using format provided in Part 0 above) in which you stop user input in OpenMRS with OWASP ZAP and change the input string to an attack. (Consider using the strings that can be found in the ZAP rulesets, such as jbrofuzz) Use these instructions as a guide.
- In your test case, be sure to document the page URL, the input field, the initial user input, and the malicious input. Describe what “filler” information is used for the rest of the fields on the page (if necessary).
- Run the test case and document the results.

Fuzzing

- Record how much total time (hours and minutes) your team spends to complete this activity.
 - Do not include time to run ZAP
 - Provide:
 - * Total time to work with the ZAP output to identify the 5 vulnerabilities.
 - * Total time to plan and run the 5 black box test cases.
- Use the 5 client-side bypassing testcases (above) for this exercise.
- Use the jbrofuzz rulesets to perform a fuzzing exercise on OpenMRS with the following vulnerability types: Injection, Buffer Overflow, XSS, and SQL Injection.
- Take a screen shot of ZAP information on the five test cases.
- Report the fuzzers you chose for each vulnerability type along with the results, and what you believe the team would need to do to fix any vulnerabilities you find. If you don't find any vulnerabilities, provide your reasoning as to why that was the case, and describe and what mitigations the team must have in place such that there are no vulnerabilities.

2. DAST-2 (25 points)

[DAST-2] FAQ [hyperlink removed] and [DAST-2] Troubleshooting [hyperlink removed]

- Record how much total time (hours and minutes) your team spends to complete this activity.
 - Do not include time to run [DAST-2].
 - Provide:
 - * Total time to work with the [DAST-2] output to identify the 5 vulnerabilities.
 - * Total time to plan and run the 5 black box test cases.
- Run [DAST-2] on OpenMRS. Run any 5 of your test cases from Project Part 1 to seed the [DAST-2] run. Run [DAST-2] long enough that you feel you have captured enough true positive vulnerabilities that you can complete five test case plans. Note: [DAST-2] will like run out of memory if you run all 5 together. It is best to run each one separately. Also, make sure you capture only the steps for your test cases, not other unnecessary steps.
- Export your results.
- Take a screen shot of [DAST-2] information on the five vulnerabilities you will explore further. Write five black box test plans (using format provided in Part 0 above) to expose five vulnerabilities detected by [DAST-2] (which may use a proxy). Hint: Your expected results should be different from the actual results since these test cases should be failing test cases.

3. Vulnerable Dependencies (35 points)

[Assignment Section not Relevant]

4. Peer evaluation (10 points)

Perform a peer evaluation on another team. Produce a complete report of feedback for the other team using this rubric (to be supplied).

B.3.3 Project Part 3

The project can be done on the VCL Class Image (“CSC 515 Software Security Ubuntu”).

0. Black Box Test Cases

Parts 1 (Logging), 2 ([Interactive Testing]), and 3 (Test coverage) ask for you to write black box test cases. We use the same format as was used in Project Part 1. For each test case, you must specify:

- A unique test case id that maps to the ASVS, sticking to Level 1 and Level 2. Provide the name/description of the ASVS control. Only one unique identifier is needed (as opposed to the example in the lecture slides). The ASVS number should be part of the one unique identifier.
- Detailed and repeatable (the same steps could be done by anyone who reads the instructions) instructions for how to execute the test case
- Expected results when running the test case. A passing test case would indicate a secure system.
- Actual results of running the test case.
- Indicate the CWE (number and name) for the vulnerability you are testing for.

1. Logging (25 points)

Where are the Log files? Check out the OpenMRS FAQ

- Record how much total time (hours and minutes) your team spends to complete this activity (test planning and test execution). Compute a metric of how many true positive defects you found per hour of total effort.
- Write 10 black box test cases for ASVS V7 Levels 1 and 2. You can have multiple test cases for the same control testing for logging in multiple areas of the application. What should be logged to support non-repudiation/accountability should be in your expected results.
- Run the test. Find and document the location of OpenMRS’s transaction logs.

- Write what is logged in the actual results column. The test case should fail if non-repudiation/accountability is not supported (see the 6 Ws on page 3 of the lecture notes).
- Comment on the adequacy of OpenMRS's logging overall based upon these 10 test cases.

2. Interactive Application Security Testing (25 points)

[Assignment Section not Relevant]

3. Test Coverage (25 points)

This test coverage relates to all work you have done in Project Parts 1, 2, and 3.

1. Compute your black box test coverage for each section of the ASVS (i.e. V1, V2, etc.) which includes the black box tests you write for Part 2 (Seeker) for Level 1 and Level 2 controls. You get credit for a control (e.g. V1.1) if you have a test case for it. If you have more than one test case for a control, you do not get extra credit –coverage is binary. Coverage is computed as # of test cases / # of requirements.
2. (15 points, 3 points each) Write 5 more black box tests to increase your coverage of controls you did not have a test case for.
3. (5 points) Recompute your test coverage. Report as below. Record how much total time (hours and minutes) your team spends to complete this activity (test planning and test execution). Compute a metric of how many true positive defects you found per hour of total effort.
4. (5 points) Reflect on the controls you have lower coverage for. Are these controls particularly hard to test, we didn't cover in class, you just didn't get to it, etc.

Control	# of test cases	# of L1 and L2 controls	Coverage
V1.1: Secure development lifecycle	?	7	?/7
...			
Total			

4. Vulnerability Discovery Comparison (15 points)

1. (5 points) Compare the five vulnerability detection techniques you have used this semester by first completing the table below.
 - A: total number # of true positives for this detection type for all activities (Project Parts 1-3)

- B: total time spent on all for all activities (Project Parts 1-3)
- Efficiency is A/B
- Exploitability: give a relative rating of the ability for this technique to find exploitable vulnerabilities
- Provide the CWE number for all the true positive vulnerabilities detected by this technique. (This information will help you address the “wide range of vulnerability types” question below.)

Technique	# of true positive vulnerabilities discovered	Total time (hours)	Efficiency: # vulnerabilities / total time	Detecting Exploitable vulnerabilities? (High-/Med/Low)	Unique CWE numbers
Manual black box					
Static analysis					
Dynamic analysis					
Interactive testing					

2. (10 points) Use this data to re-answer the question that was on the midterm (that people generally didn't do too well on). Being able to understand the tradeoffs between the techniques is a major learning objective of the class.

As efficiently and effectively as possible, companies want to detect a wide range of exploitable vulnerabilities (both implementation bugs and design flaws). Based upon your experience with these techniques, compare their ability to efficiently and effectively detect a wide range of types of exploitable vulnerabilities.

5. Peer evaluation (10 points)

Perform a peer evaluation on another team. Produce a complete report of feedback for the other team using this rubric [to be supplied].

B.3.4 Project Part 4

1. Protection Poker (20 points)

[Assignment Section not Relevant]

2. Vulnerability Fixes (35 points)

[Assignment Section not Relevant]

3. Exploratory Penetration Testing (35 points)

Each team member is to perform 3 hours of exploratory penetration testing on OpenMRS. This testing is to be done opportunistically, based upon your general knowledge of OpenMRS but without a test plan, as is done by professional penetration testers. **DO NOT USE YOUR OLD BLACK BOX TESTS FROM PRIOR MODULES.** Use a screen video/voice screen recorder to record your penetration testing actions. Speak aloud as you work to describe your actions, such as, “I see the input field for logging in. I’m going to see if 1=1 works for a password.” or “I see a parameter in the URL, I’m going to see what happens if I change the URL.” **You should be speaking around once/minute to narrate what you are attempting.** You don’t have to do all 3 hours in one session, but you should have 3 hours of annotated video to document your penetration testing. There’s lots of screen recorders available – if you know of a free one and can suggest it to your classmates, please post on Piazza.

Pause the recording every time you have a true positive vulnerability. Note how long you have been working so a log of your work and the time between vulnerability discovery is created (For example, Vulnerability #1 was found at 1 hour and 12 minutes, Vulnerability #2 was found at 1 hour and 30 minutes, etc.) If you work in multiple sessions, the elapsed time will pick up where you left off the prior session – like if you do one session for 1 hour 15 minutes, the second session begins at 1 hour 16 minutes. Take a screen shot and number each true positive vulnerability . Record your actions such that this vulnerability could be replicated by someone else via a black box test case. Record the CWE for your true positive vulnerability. Record your work as in the following table. The reference info for video traceability is to aid a reviewer in watching you find the vulnerability. If you have one video, the “time” should aid in finding the appropriate part of the video. If you have multiple videos, please specify which video and what time on that video.

Vulnerability #	Elapsed Time	Ref Info for Video Traceability	CWE	Commentary

Replication instructions via a black box test and the screenshots for each true positive vulnerability should appear below the table, labeled with the vulnerability number. Since you are not recording all your steps, the replication instructions may not work completely since you

may change the state of the software somewhere along the line – document what you can via a black box test and say the actual results don't match your screenshot.

After you are complete, compute an efficiency metric (true positive vulnerability/hour) metric for each student. Submit a table:

	# vuln	Time	Efficiency
Name 1			
Name 2			
Name 3			
Name 4			
Total			

Copy the efficiency table you turned in for Project Part 3 #4. Add an additional line for Penetration testing. Compare and comment on this efficiency rate with the other vulnerability discovery techniques in the table you input in #4 of Project Part 3.

- Each person on the team should submit one or more videos by uploading it/them to your own google drive and providing a link to the video(s), sharing the video with anyone who has the link and an NCSU login (which will allow peer evaluation and grading). The video(s) should be approximately 3 hours in length.
- A person who does not submit a video can not be awarded the points for this part of the project while the rest of the team can.
- It is possible to work for 3 hours and find 0 vulnerabilities – real penetration tests constantly work more than 3 hours without finding anything. That's part of the reason for documenting your work via video.
- For those team members who do submit videos, the grade will be an overall team grade.

Submission: The team submits one file with the links to the team member's files.

4. Peer Evaluation (10 points)

Perform a peer evaluation on another team. Produce a complete report of feedback for the other team using this rubric [to be supplied].

B.4 Appendix - Equipment Specifications

In this appendix we provide additional details of the equipment used in our case study. As noted in Section 3.9, a key resource used in this project was the school's Virtual Computing Lab¹ (VCL), which provided virtual machine (VM) instances. Researchers used VCL when applying EMPT, SMPT, and DAST as part of data collection for RQ1. All student tasks were performed using VCL for RQ1 and RQ2. Researchers created a system image including the SUT (OpenMRS) as well as SAST and DAST tools. The base image was assigned 4-cores, 8G RAM, and 40G disk space. An instance of this image could be checked out by students and researchers and accessed remotely through a terminal using ssh or graphically using Remote Desktop Protocol (RDP). Researchers also used two expanded instances of the base image with 16 CPUs, 32GB RAM, and 80G disk space. For client-server tools, a server was setup in a separate VCL instance by researchers with assistance from the teaching staff of the course. The server UI was accessible from VCL instances of the base image, while the server instance itself was only accessible to researchers and teaching staff. The server instance had 4 cores, 8G RAM, and 60G disk space, and contained the server software for SAST-1 used to answer RQ2. All VCL instances in this study used the Ubuntu operating system.

The VCL alone was used for data collection for RQ2. However, the base VCL images were small, and the remote connection to VCL could lag. Researchers used two used additional resources as needed for RQ1 data collection. First, we created a VM in VirtualBox using the same operating system (Ubuntu 18.04 LTS) and OpenMRS version (Version 2.9) as the VCL images. This VM was used by researchers for SMPT and EMPT data collection, particularly when reviewing the output of each technique where instances of the SUT were needed on an ad hoc basis. The VM was assigned 2 CPUs, 4GB RAM, and 32G disk space and could be copied and shared amongst researchers to run locally. Researchers increased the size of the VM as needed, up to 8 CPUs and 16GB RAM when the host system could support the VM size. A second VM was created in VirtualBox with the same specifications and operating system, but with the server software for Sonarqube installed. We also used a desktop machine with 24 CPUs, 32G RAM, and 500G disk space. The desktop was running the Ubuntu operating system. This machine was accessible through the terminal via ssh and graphically using x2go². For RQ1 data collection we ran the SAST-1 server software directly on this machine. The desktop was also used to run VirtualBox VMs for resource-intensive activity such as running Sonarqube and DAST-2.

While equipment constraints impacted both SAST and DAST, available equipment and intended use also impacts how SAST tools are setup. SAST tools can be setup and configured according to different architectures. The SAST tools used in this study could be setup as client-server tools where the SUT code is scanned on the "client" machine, and information is sent to a "server". The analyst then reviews the results through the server. For some tools, the automated analysis and rules are applied on the client, while for other tools the automated analysis and rules are applied on the server. The SAST tools used in this study also included an optional plugin for Integrated Development Environments (IDEs) such as Eclipse³. The plugin allows

¹<https://vcl.apache.org/>

²<https://wiki.x2go.org/doku.php>

³<https://www.eclipse.org/ide/>

developers to initialize SAST analysis and in some cases view alerts from the tool within the IDE itself. Some tools can be run without a server using only IDE plugins. Other tools require a server. Similar to the previous work by Austin et al. [22, 23], we found that the server GUI was easier to use when aggregating and analyzing all system vulnerabilities for RQ1. Consequently, a client-server configuration was used with SAST-2 and Sonarqube to answer RQ1. SAST-2 and SAST-3 were more easily configured to use locally within an IDE, as was done in for the class with RQ2 and RQ3.

APPENDIX

C

VULNERABILITY DETECTION TECHNIQUE COMPARISON - ALL CWES TABLE

Table C.1 shows the CWE for high and medium severity vulnerabilities found. Table C.2 provides the same information for low severity vulnerabilities. The first column of the table indicates the CWE number. The CWEs are organized based on the OWASP Top Ten Categories. The second column of the table indicates which, if any, of the OWASP Top Ten the vulnerability maps to. Columns three and four of the table are the number of vulnerabilities found by the techniques SMPT and EMPT. Columns five through eight break down the vulnerabilities found by DAST and SAST by tool (ZAP, DA-2, Sonar, and SA-2). Column nine of Table C.1 shows the total number of vulnerabilities found of each CWE type. The Total column is not the same as the sum of the previous six columns. Some vulnerabilities were found using more than one technique. Similarly, 20 Vulnerabilities were associated with more than one CWE; therefore the total vulnerabilities for each technique as shown in Table 3.5 may be lower than the sum of each column in Table C.1.

Table C.1: CWEs associated with *more severe* Vulnerabilities

CWE	Top Ten	SMPT	EMPT	DAST		SAST		Total
				ZAP	DA2	Sonar	SA2	
A01 Broken Access Control								

Table C.1 (Continued)

CWE	TT	SMPT	EMPT	ZAP	DA2	Sonar	SA2	Total
922 - Insecure Storage of Sensitive Information	A01	1						1
200 - Exposure of Sensitive Information to an Unauth. Actor	A01		2					2
601 - URL Redirection to Untrusted Site ('Open Redirect')	A01						9	9
285 - Improper Authorization	A01	1	13					13
22 - Path Traversal	A01						19	19
A02 Cryptographic Failures								
326 - Inadequate Encryption Strength	A02			1				1
319 - Cleartext Transmission of Sensitive Information	A02	1	1					1
327 - Use of a Broken or Risky Cryptographic Algorithm	A02					2		2
A03 Injection								
643 - XPath Injection	A03						1	1
89 - SQL Injection	A03						4	4
20 - Improper Input Validation	A03	3	19		2			21
79 - Cross-site Scripting	A03	2	100	3	7		19	124
A04 Insecure Design								
269 - Improper Privilege Management	A04		1					1
313 - Cleartext Storage in a File or on Disk	A04						1	1
770 - Allocation of Resources Without Limits or Throttling	A04	1	1					1
419 - Unprotected Primary Channel	A04	2	1					2
807 - Reliance on Untrusted Inputs in a Security Decision	A04					3		3

Table C.1 (Continued)

CWE	TT	SMPT	EMPT	ZAP	DA2	Sonar	SA2	Total
73 - External Control of File Name or Path	A04						4	4
598 - Use of GET Request Method With Sensitive Query Strings	A04	2	5		1			6
A05 Security Misconfiguration								
548 - Exposure of Information Through Directory Listing	A05			1				1
614 - Sensitive Cookie in HTTPS Session Without 'Secure' Attribute	A05	1	1					1
16 - Configuration	A05	1	1	1				3
611 - Improper Restriction of XML External Entity Reference	A05					13	1	14
A06 Vulnerable and Outdated Components								
A07 Identification and Authentication Failures								
308 - Use of Single-factor Authentication	A07		1					1
384 - Session Fixation	A07	1	1					1
620 - Unverified Password Change	A07	1						1
346 - Origin Validation Error	A07						2	2
613 - Insufficient Session Expiration	A07	1	1		1			2
521 - Weak Password Requirements	A07	10	7					10
A08 Software and Data Integrity Failures								
829 - Inclusion of Functionality from Untrusted Control Sphere	A08	1						1
502 - Deserialization of Untrusted Data	A08						10	10

Table C.1 (Continued)

CWE	TT	SMPT	EMPT	ZAP	DA2	Sonar	SA2	Total
A09 Security Logging and Monitoring Failures								
532 - Insertion of Sensitive Information into Log File	A09	1	1					1
778 - Insufficient Logging	A09	2	9					11
A10 Server-Side Request Forgery (SSRF)								
918 - Server-Side Request Forgery (SSRF)	A10	1						1
Not Mapped to OWASP Top Ten								
509 - Replicating Malicious Code (Virus or Worm)	NA	1	1					1
1022 - Use of Web Link to Untrusted Target with window.opener Access	NA					1		1
674 - Uncontrolled Recursion	NA						2	2
567 - Unsynchronized Access to Shared Data in a Multithreaded Context	NA						4	4
543 - Use of Singleton Pattern Without Synchronization in a Multithreaded Context	NA						8	8
827 - Improper Control of Document Type Definition ¹	NA					13	1	14
404 - Improper Resource Shutdown or Release	NA						39	39

¹If comparing the current Table (Table C.1) to Table 3.6 in Chapter 3, it should be noted that all 14 vulnerabilities classified under CWE 827 were also classified under CWE 611. Since CWE 611 maps to A05 in the OWASP Top Ten, these vulnerabilities are classified under A05 in Table 3.6. They are not classified as “No Mapping” in Table 3.6 since they are mapped through another CWE.

Table C.2: Low Severity Vulnerability CWEs

CWE	Top Ten	SMPT	EMPT	DAST		SAST		Total
				ZAP	DA2	Sonar	SA2	
A01 Broken Access Control								
352 - Cross-Site Request Forgery	A01			1		220	20	234
A02 Cryptographic Failures								
760 - Use of a One-Way Hash with a Predictable Salt	A02						2	2
A03 Injection								
470 - Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')	A03						34	34
A04 Insecure Design								
209 - Generation of Error Message Containing Sensitive Information	A04	2	18		1			18
501 - Trust Boundary Violation	A04						28	28
A05 Security Misconfiguration								
7 - Missing Custom Error Page	A05	1	1	1	1		1	1
933 - Security Misconfiguration	A05			1				1
16 - Configuration	A05	2	1	2				2
A06 Vulnerable and Outdated Components								
A07 Identification and Authentication Failures								
A08 Software and Data Integrity Failures								
345 - Insufficient Verification of Data Authenticity	A08			1				1
502 - Deserialization of Untrusted Data	A08					1		1
A09 Security Logging and Monitoring Failures								
A10 Server-Side Request Forgery (SSRF)								
Not Mapped to OWASP Top Ten								

Table C.2 (Continued)

CWE	TT	SMPT	EMPT	ZAP	DA2	Sonar	SA2	Total
242 - Use of Inherently Dangerous Function	NA						2	2
615 - Inclusion of Sensitive Information in Source Code Comments	NA						5	5
404 - Improper Resource Shutdown or Release	NA						17	17
489 - Active Debug Code	NA					26		26
582 - Array Declared Public, Final, and Static	NA					31		31
754 - Improper Check for Unusual or Exceptional Conditions	NA					31		31
600 - Uncaught Exception in Servlet	NA					60	3	60
493 - Critical Public Variable Without Final Modifier	NA					210		210

APPENDIX

D

OPENSSEF MEASUREMENT METHOD DETAILS

Table D.1 contains additional algorithmic details of how each of the OpenSSF checks are implemented, to supplement the information provided previously in Table 5.1 in Chapter 5. The first column of Table D.1 indicates the name of the check. The second column indicates the standardization (Std.) method used to transform the score into the [0-10] range. The *Details* of how the check is computed are in the third and final column of Table D.1. A value of -1 , such as shown in one of the cases for the *Maintained* check in Table D.1 is output when a particular check cannot be correctly applied. Detailed information is derived from the documentation and source code of the openSSF Repository¹

As discussed in Chapter 5 Section 5.1.1.4, each check is standardized to a value between 0 and 10 (inclusive), using one of four standardization methods: Binary (B), Deduction (D), Points-Based (Pt), and Proportional (Pr). In Table D.1, for checks which use different standardization methods on a case-by-case basis the standardization methods are separated by a |. We include the standardization methods associated with each check in Table D.1, since the standardization method is reflected in the math used to compute each check.

¹<https://github.com/ossf/scorecard/>

Table D.1: OpenSSF Check Details

Name	Std.	Details
GROUP: Holistic Security Practices - Code Vulnerabilities		
Vulnerabilities	D	$\max(0, 10 - \#VULN)$ <p>where: $\#VULN$ the # vulnerabilities in the project and its dependencies, as detected by the OSV scanner</p>
GROUP: Holistic Security Practices - Maintenance		
Dependency Update Tool	B	$\begin{cases} 10 & \text{tool detected} \\ 0 & \text{tool NOT detected} \end{cases}$ <p>Checks to see if Dependabot, Renovate bot, Sonatype Lift, or PyUp are used via Github Apps or Github Actions.</p>
Maintained	Pr B	$\begin{cases} 0 & \text{project archived} \\ -1 & \text{project} < 90 \text{ days old} \\ \min\left(10, 10 * \frac{\#commits + \#issuesUpdated}{\text{expectedActivity}}\right) & \text{otherwise} \end{cases}$ <p>where: $\#commits$ the # of commits in the past 90 days $\#issuesUpdated$ is the # of issues where a project collaborator has commented in the past 90 days, and expectedActivity is the $\#commits + \#updates$ expected for the 90 day period, currently 12.86 (1 per week)</p>

Table D.1 (Continued)

Name	Std.	Details
Security Policy	Pt	<p style="text-align: center;"><i>LINK + TXT + SECURITY_TXT</i></p> <p>where:</p> <p><i>LINK</i> = 6 if SECURITY.md includes a valid address to contact for vulnerabilities (<i>link</i> = 0, otherwise)</p> <p><i>TXT</i> = 3 if SECURITY.md includes any free form text beyond the contact links (<i>TXT</i> = 0, otherwise)</p> <p><i>SECURITY_TXT</i> = 1 if SECURITY.md includes specific text about vulnerability management (<i>SECURITY_TXT</i> = 0, otherwise)</p> <p>All values = 0 if there is no SECURITY.md file.</p>
License	Pt	<p style="text-align: center;"><i>DETECTED + TOPLEVEL + FSF_OSI</i></p> <p>where:</p> <p><i>DETECTED</i> = 6 if a LICENSE, COPYRIGHT, or COPYING filename is detected with static analysis (<i>DETECTED</i> = 0, otherwise)</p> <p><i>TOPLEVEL</i> = 3 if the detected file is at the top-level directory (<i>TOPLEVEL</i> = 0, otherwise)</p> <p><i>FSF_OSI</i> = 1 if a FSF or OSI license is specified (<i>FSF_OSI</i> = 0, otherwise)</p>
Best Practices	Pt	<p style="text-align: center;"> $\left. \begin{array}{l} 2 \\ PASS + SILVER + GOLD \end{array} \right\} \begin{array}{l} badge \text{ is "inprogress"} \\ otherwise \end{array}$ </p> <p>where:</p> <p><i>PASS</i> = 5 if the project has a passing badge (<i>PASS</i> = 0, otherwise)</p> <p><i>SILVER</i> = 2 if the project has a "silver" or better badge (<i>SILVER</i> = 0, otherwise)</p> <p><i>GOLD</i> = 3 if the project has a "gold" or better badge (<i>GOLD</i> = 0, otherwise)</p> <p>A gold badge will earn all 10 points (gold > silver)</p> <p>If the project does NOT have a badge, the project will have 0 points.</p>

GROUP: Holistic Security Practices - Continuous Testing

Table D.1 (Continued)

Name	Std.	Details
CI Tests	Pr B	$10 * \frac{\#TESTED}{\#MERGED}$ <p>where: $\#TESTED$ the # of pull requests whose Github-based CI/CD pipeline includes a check whose name contains one of the following substrings: appveyor, buildkite, circleci, e2e, github-actions, jenkins,mergeable, packit-as-a-service, semaphoreci, test, travis-ci,flutter-dashboard,Cirrus CI, azure-pipelines $\#MERGED$ is the # pull requests merged into the repository</p>
Fuzzing	B	$\begin{cases} 10 & \text{tool detected} \\ 0 & \text{tool NOT detected} \end{cases}$ <p>Checks to see if the project is on the OSS-Fuzz list. Additionally, uses regular-expressions-based matching on filenames and contents to detect use of one or more of the following fuzzing tools: ClusterFuzzLite, Go Fuzzing, QuickCheck, Hedgehog, Validity, SmallCheck, Fast-Check, OneFuzz, Atheris,LibFuzzer, Cargo-Fuzz</p>
SAST	B Pr	$\begin{cases} 10 & \text{sonar configuration filename detected} \\ 3 + \left(7 * \frac{\#CHECKED}{\#MERGED}\right) & \text{github/codeql-action in Github Workflow} \\ \frac{\#CHECKED}{\#MERGED} & \text{otherwise} \end{cases}$ <p>where: $\#CHECKED$ the # of pull requests whose Github-based CI/CD pipeline includes a check whose name contains one of the following substrings: github-advanced-security, github-code-scanning, lgtm-com, sonarcloud $\#MERGED$ is the # pull requests merged into the repository Value will be 0 if no SAST tools are detected, since $\#CHECKED = 0$</p>

GROUP: Source Risk Assessment

Table D.1 (Continued)

Name	Std.	Details
Binary Artifacts	D	$\max(0, 10 - \#BIN)$ <p>where: $\#BIN$ the # binaries in the project repository</p>
Branch Protection	Pt	$TIER1 + TIER2 + TIER3 + TIER4 + TIER5$ <p>where: $TIER1 = 3$ force push and branch deletion are prevented (otherwise $TIER1 = 0$) $TIER2 = 3$ $TIER1$ is met and the repository requires reviewer approval before merging, requires the branch to be up-to-date before merging from administrators, and requires administrator approval of the most recent reviewable push (otherwise $TIER2 = 0$) $TIER3 = 2$ $TIER2$ is met and a branch must pass at least 1 status check to be merged (otherwise $TIER3 = 0$) $TIER4 = 1$ $TIER3$ is met, code must have at least 2 reviewers before merging, and all code is reviewed by the code owners (otherwise $TIER4 = 0$) $TIER5 = 1$ $TIER4$ is met. Additionally, administrators are included in code review and dismiss stale reviews and approvals when new commits are pushed (otherwise $TIER5 = 0$).</p>
Dangerous Workflow	B	$\begin{cases} 10 & \text{NO dangerous patterns in Workflow} \\ 0 & \text{dangerous patterns in Workflow} \end{cases}$ <p>Detects dangerous patterns in workflow scripts using static analysis</p>

Table D.1 (Continued)

Name	Std.	Details
Code Review	Pr	$10 * \frac{\#REVIEWED}{\#CHANGES}$ <p>where: $\#REVIEWED$ the # of merged changes where one or more of the following is true:</p> <ul style="list-style-type: none"> • Merge Request labels include phrases associated with the Prow review tool (lgtm, approved) • the commit message associated with the merge include phrases associated with Gerrit (Reviewed-on:, Reviewed-by:), Phabricator (Differential Revision:), Piper (PiperOrigin-RevID:) • the merge request status is Approved on github • the Github user associated with the merge request is different from the Github user associated with the original commit <p>$\#CHANGES$ is the total # changes merged</p>
Contributors	Pr	$\min\left(10, 10 * \frac{\#ORGS}{EXPECTED_ORGS}\right)$ <p>where: $\#ORGS$ the # distinct organizations associated with project contributors $EXPECTED_ORGS$ is number of distinct organizations needed to earn a perfect score on the check (currently set to 3)</p>

GROUP: Build Risk Assessment

Table D.1 (Continued)

Name	Std.	Details
Pinned Dependencies	Pr	$\left. \begin{aligned} & \left(7 * \frac{\#GHP}{\#TGH} \right) + \left(2 * \frac{\#TPP}{\#TPT} \right) && \textit{Repository uses Github Actions} \\ & 10 * \frac{\#PINNED}{\#TOTAL_DEP} && \textit{Other dependency ecosystems} \end{aligned} \right\}$ <p>where:</p> <p><i>#GHP</i> the # Github-owned pinned dependencies (in Github Actions) <i>#TGH</i> the total # Github-owned dependencies (in Github Actions) <i>#TPP</i> the # third-party pinned dependencies (in Github Actions) <i>#TPT</i> the total # third-party dependencies (in Github Actions) <i>#PINNED</i> the # pinned dependencies (in other ecosystems) <i>#TOTAL_DEP</i> the total # dependencies (in other ecosystems)</p>
Token Permissions	D	$\left. \begin{aligned} & 0 && \textit{all perms. by default} \\ & \max\left(0, 10 - (0.5 * STATUS) - (0.5 * CHECK) \right. && \textit{otherwise} \\ & \quad \quad \quad - SECURITY - DEPLOYMENT \\ & \quad \quad \quad - CONTENT - PACKAGE \\ & \quad \quad \quad \left. - ACTIONS \right) \end{aligned} \right\}$ <p>where:</p> <p><i>STATUS</i> # of files whose default permissions include statuses <i>CHECK</i> # of files whose default permissions include checks <i>SECURITY</i> # of files whose default permissions include security-events <i>DEPLOYMENT</i> # of files whose default permissions include deployments <i>CONTENT</i> # of files whose default permissions include contents <i>PACKAGE</i> # of files whose default permissions include packages <i>ACTION</i> # of files whose default permissions include actions Note: Value is 0 if all permissions are enabled by default for ANY file</p>

Table D.1 (Continued)

Name	Std.	Details
Packaging	B	$\begin{cases} 10 & \text{Repository uses Github packaging workflows} \\ 0 & \text{Repository does NOT use Github packaging workflows} \end{cases}$
Signed Releases	Pr	$\frac{\sum_{i=0}^{REL} \max(PROV(r_i), SIG(r_i))}{10 * REL}$ <p>where: REL is the # releases from the Github Release tool² $PROV(r_i) = 10$ if the release assets for release r_i from the Github Release tool include files with an extension associated with SLSA provenance files, currently [.intoto.jsonl] (otherwise $PROV(r_i) = 0$) $SIG(r_i) = 8$ if the release assets for release r_i from the Github Release tool include files with an extension associated with a release signature provenance files, currently [.asc, .minisig, .sig, .sign] (otherwise $SIG(r_i) = 0$)</p>

²<https://docs.github.com/en/repositories/releasing-projects-on-github/about-releases>

APPENDIX

E

SURVEY EXAMPLE

In this section, we show the survey as it was presented to dependents of the `pgx` package. In the example, we include the questions for the 6 checks examined in our study, followed by the 3 questions that were later determined to be out of scope. For each participant, 7 of the 9 checks would be randomly selected for inclusion in the survey.

The `pgx` package was one of the example dependencies around which we built our survey, as discussed in Section 5.6.2.1. The dependencies that were included as examples in the survey were `cobra`, `cgroups`, `chimeracoder/anaconda`, `fasthttp`, and `pgx`. The `cobra` package is a user-interface package for command-line tools[45], used by the OpenSSF Scorecard tool itself among other dependents. The `cgroups` package is used to help facilitate container configurations[39]. The `chimeracoder/anaconda` package provides functionality for interfacing with the twitter API[18]. The `fasthttp` package is an http client and server API which is more efficient than the built-in http functionality for some use-cases[89], such as applications requiring “thousands of small to medium requests per second”. Finally, `pgx` is an alternative postgresql (database) driver which claims to be more efficient than the default Postgres driver for the functionality it supports, but which does not support all of the functionality supported by the default drive [227]. For `pgx` we focused our analysis on v4 rather than the latest version (v5), since at the time of survey deployment, v4 was more popular, suggesting some maintainers had not yet upgraded. For the other packages we used the latest version. Individuals who selected “None of the above” for the package they were associated with were randomly presented with either `fasthttp` or `pgx`, since many software developers will be familiar with the concept of http interfaces and database drivers, even if they do not use them.

All of the scores presented in the survey were based on the real scores of the example

package. However, in some cases we used hypothetical values to better illustrate the potential differences in approach. As we discuss in Section 5.8.1.2, the original (non-aggregated) and aggregate scores were the same in the majority of cases. In our pilot study, we found this to be confusing for participants. Consequently, for each example dependent we altered the scores so that at-most two of the checks would have the same score for the original and aggregate values. Each participant was only presented with 7 out of the 9 checks, and so in some cases the participant would not be presented with both of the checks that had the same score, and therefore would have less than two checks with the same score.

Example Survey

Thank you for participating in this survey! As a reminder, information about the IRB protocol for our survey, including information about whom to contact if you have questions and how you can withdraw from this survey at any time, can be found at the following link: https://github.com/RealsearchGroup/RealsearchGroup.github.io/blob/main/misc/26676%20Survey%20Consent%20Form_v4a_release.pdf

A system you maintain and/or are familiar with should be shown in the dropdown below. If not, please select a system you are familiar with: <dropdown with dependent the respondent maintains selected, if applicable, otherwise the respondent can select “None of these apply (your survey will not be specific to your project)”, which will select a random dependency>

Prior to hearing about our study, were you familiar with the OpenSSF Scorecard tool?

- Yes, and I have experience using the tool
- Yes, but I have never used the tool
- No, I had never heard of the OpenSSF Scorecard tool

We will be examining a number of security-related metrics. Specifically - we will be using the metrics collected by the OpenSSF Scorecard tool (also referred to as "checks"). While we are in discussions with OpenSSF, this research is performed independently of OpenSSF.

OpenSSF Scorecard is an automated tool to collect information reflecting a series of concepts such as whether a project is actively maintained, and what security-related tools are being used. A full list of the metrics and more information about the tool itself is available in the [OpenSSF Scorecard documentation](#). Each metric provides a score value between 0 and 10 where 0 is the worst possible score and 10 is the best possible score, e.g. “no problems detected”. We will include an overview of the relevant metric in each of our questions.

The score values discussed in this survey are hypothetical, based on real scores but altered slightly to illustrate the concepts we are asking questions about.

We will be using the security metrics from OpenSSF Scorecard tool to understand what information from a package's dependency tree needs to be surfaced when making decisions about whether to include or replace the component.

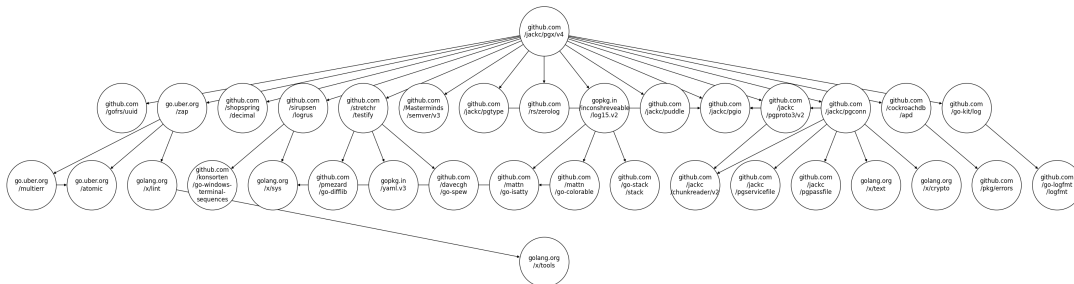
Specifically, we will ask:

When determining whether to include or replace the package, which of the following scores would you most prefer to know:

We will then present three options:

1. The current score, **which only accounts for the package to be included, NOT its dependencies**
2. A proposed score, **which accounts for more of the package's dependency tree**. This will vary based on the metric we examine, and will be explained fully in each question
3. **Other** if neither the original score, nor our proposed score is sufficient for making a decision (including space to explain what would work better)

As part of this survey, we will be discussing the `github.com/jackc/pgx/v4`, a popular GO package for using PostgreSQL with GO. We have selected this package since it is a popular alternative to the standard database interfaces, and likely familiar to many survey participants. Even if you are not familiar with the package, it may be a helpful reference. The package and its dependency tree are shown below.



The dependencies that are explicitly referenced by the root node (`pgx`), are referred to as **direct** dependencies. In the example above, the direct dependencies are `uuid`, `zap`, `decimal`, `logrus`, `testify`, `semver`, `pgtype`, `zerolog`, `log15`, `puddle`, `pgio`, `pgproto3`, `pgconn`, `cockroachdb/apd`, and `go-kit/log`

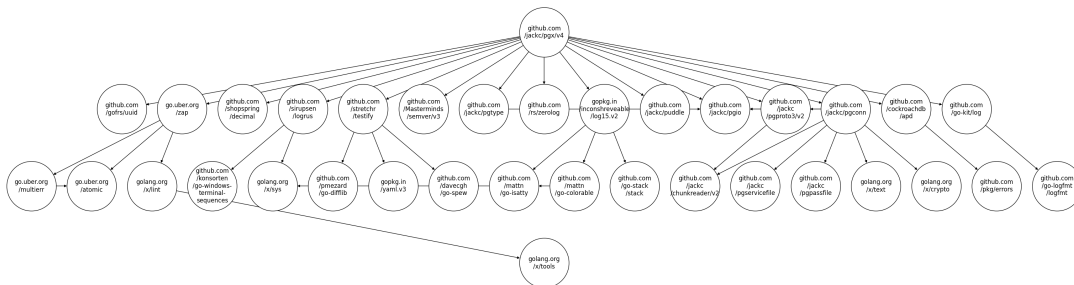
The dependencies that are included in the dependency tree but which are not explicitly referenced by the root node, e.g. are included because they are referenced by direct dependencies, are referred to as indirect dependencies. In the example above, the indirect dependencies are `multierr`, `atomic`, `lint`, `go-windows-terminal-sequences`, `sys`, `go-difflib`,

yaml.v3, go-spew, go-isatty, go-colorable, stack, chunkreader, pgservicefile, pgpassfile, text, crypto, errors, logfmt, and tools.

OpenSSF Scorecard includes a check that examines *whether Binary Artifacts, e.g. executables, are found in the source code* of an open source package.

The Binary Artifacts check currently only accounts for Binary Artifacts that occur in the root package, `pgx` in the example below. For example, if the root package did not contain binary artifacts, but one of its dependencies did, the root package would still score 10 out of 10. The Binary Artifacts score is calculated as $10 - \#Artifacts$ in the package.

For reference, the dependency tree of `pgx` is below. *In the case of `pgx`, neither `pgx` nor any of the dependencies have a binary artifact.* Hence the score is 10 out of 10, regardless of the aggregation method.



When determining whether to include or replace the package `pgx`, which of the following scores would you most prefer to know:

- 10 (out of 10):** Calculated by subtracting *only Binary artifacts in the root package* (the remainder of the dependency tree is not known)
- 10 (out of 10):** 10 (out of 10): Calculated by *subtracting Binary Artifacts from the entire dependency tree*
- Other** Score or Aggregation Approach (Please Specify) <free response text box available if selected>

Why would you prefer this score/approach?

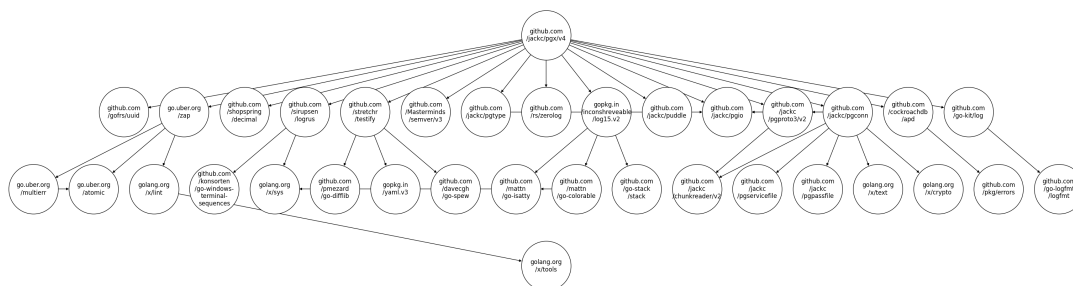
<free response text box>

OpenSSF Scorecard includes a metric, labeled “Branch Protection” that *examines the security of the project’s Branch Protection policies for their repository*.

Projects earn points based on policies, such as whether “Force Push” is disabled on public branches of the project, and whether a project requires at least 1 reviewer to approve a code change before the code change can be merged. [A full description of the Branch Protection metric can be found in the OpenSSF Scorecard Documentation](#). As with all other scores, a 10 is the “best” possible score and a 0 being the “worst” possible score.

Currently, the Branch Protection metric only examines the package itself, without accounting for policies that may be used in the package’s dependencies.

The dependency tree of the `pgx` is included below, for reference.



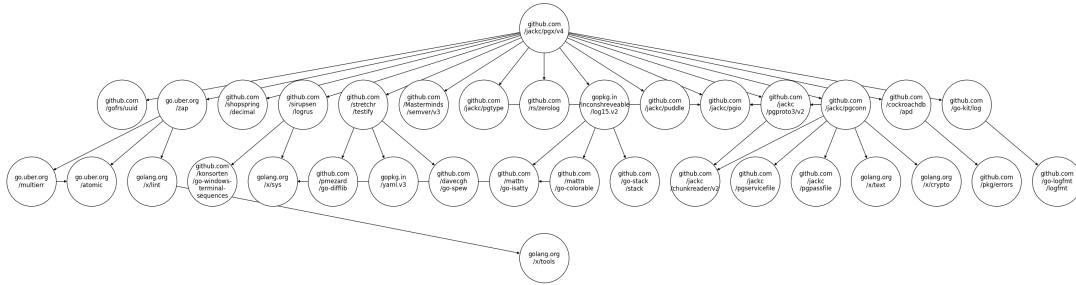
When determining whether to include or replace the package `pgx`, which of the following scores would you most prefer to know:

- () **7 (out of 10):** Following the current approach, *reflecting only the policies of the root package (`pgx`)*
- () **0 (out of 10):** The *minimum score of all packages in the dependency tree* of `pgx`
- () **Other** Score or Aggregation Approach (Please Specify) <free response text box available if selected>

Why would you prefer this score/approach?

<free response text box>

OpenSSF Scorecard includes a metric, labeled “Code Review”, that examines the level of code review required by the project. Points are deducted for any changes that are not code reviewed, with more points deducted for human changes than for bot-originated changes. [A full](#)



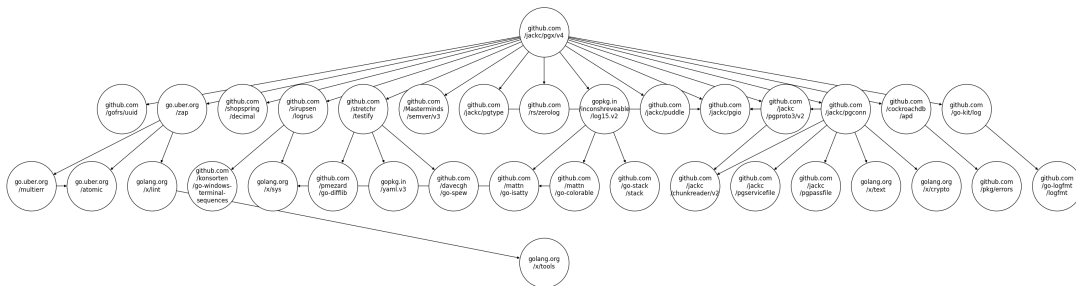
Why would you prefer this score/approach?

<free response text box>

OpenSSF Scorecard includes a metric, labeled Security Policy, that determines *whether the package has a security policy, including a process for reporting security vulnerabilities*, through the existence and contents of the project’s Security.MD file on Github. [A full description of the Security Policy metric can be found in the OpenSSF Scorecard Documentation.](#)

Currently, the Security Policy metric only examines the package itself, without accounting for the packages’ dependencies. As with other OpenSSF Scorecard scores, the value of the Security Policy ranges from 0 (lowest score, e.g. no security policy) to 10 (highest score, e.g. a detailed security policy)

For reference, the dependency tree of pgx is below.



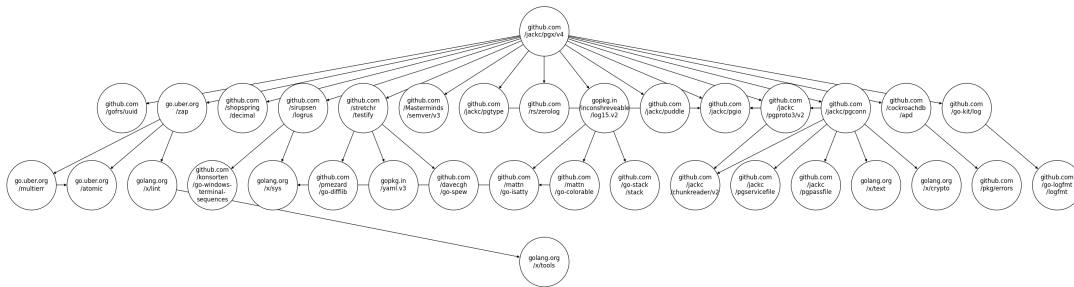
When determining whether to include or replace the package pgx, which of the following scores would you most prefer to know:

- () 0 (out of 10): Following the current approach, *the Security Policy metric of the root package*, i.e. whether pgx has a SECURITY.MD file documenting policies such as vulnerability

OpenSSF Scorecard includes a metric, labeled “SAST” that examines *whether the package uses a Static Analysis Security Testing (SAST) tool*, such as CodeQL or SonarCloud. [A full description of the SAST Metric can be found in the OpenSSF Scorecard Documentation.](#)

Currently, the SAST metric does not account for the use of such tools in any of a package’s dependencies.

The dependency tree of the pgx is included below, for reference.



When determining whether to include or replace the package pgx, which of the following scores would you most prefer to know:

- () **3 (out of 10):** Following the current approach, of scoring 10 if the root package uses a SAST tool frequently, *not considering any of its dependencies*
- () **0 (out of 10):** The *minimum score of all packages in the dependency tree* of pgx
- () **Other** Score or Aggregation Approach (Please Specify) <free response text box available if selected>

Why would you prefer this score/approach?

<free response text box>

Before you go! We have some short questions about you, which we hope to use in our analysis to account for how these factors may influence the responses you just provided. Thanks, again, for your participation!

B1: How many years have you worked in software development?

<free response text box>

How would you rate your level of cyber-security and/or software security expertise? (On a scale from 0 to 4)

0 (no expertise) 1 2 3 4 (High expertise)

How often have you *contributed to* open source software packages?

Often

Sometimes

Never

How often have you *used* open source software packages?

Often

Sometimes

Never

APPENDIX

F

SURVEY RESULTS FOR OTHER CHECKS

As discussed in Section 5.8.2, three checks which were originally included in the survey were later determined to be out of scope for the study. Below are the results collected for these three checks.

Table F.1: Survey Results for Checks with Limited Data

Metric	Aggregation Approach	Total	Prefer Original	Prefer Aggregate	Other (Explanation)
Dangerous Workflows	Subtracting 1 from the Root score each dependency with a Dangerous Workflow	6	0	4	2 Prefer Min. of Entire Tree
Pinned Dependencies	Avg. across Dependency Tree (not to exceed Root score)	6	5	1	0
SAST	Min. of Entire Tree	6	2	3	1 Check Irrelevant to Decision