Empirical Software Engineering (2022) 27: 154
https://doi.org/10.1007/s10664-022-10179-6

Do I really need all this work to find vulnerabilities?

An empirical case study comparing vulnerability detection techniques on a Java application

Sarah Elder[1] · Nusrat Zahan[1] · Rui Shu[1] · Monica Metro[1] · Valeri Kozarev[1] · Tim Menzies[1] · Laurie Williams[1]

Accepted: 20 May 2022 / Published online: 6 August 2022

Abstract

Context: Applying vulnerability detection techniques is one of many tasks using the limited resources of a software project.

Objective: The goal of this research is to assist managers and other decision-makers in making informed choices about the use of software vulnerability detection techniques through an empirical study of the efficiency and effectiveness of four techniques on a Java-based web application.

Method: We apply four different categories of vulnerability detection techniques – systematic manual penetration testing (SMPT), exploratory manual penetration testing (EMPT), dynamic application security testing (DAST), and static application security testing (SAST) – to an open-source medical records system.

Results: We found the most vulnerabilities using SAST. However, EMPT found more severe vulnerabilities. With each technique, we found unique vulnerabilities not found using the other techniques. The efficiency of manual techniques (EMPT, SMPT) was comparable to or better than the efficiency of automated techniques (DAST, SAST) in terms of Vulnerabilities per Hour (VpH).

Conclusions: The vulnerability detection technique practitioners should select may vary based on the goals and available resources of the project. If the goal of an organization is to find "all" vulnerabilities in a project, they need to use as many techniques as their resources allow.

Communicated by: Mehrdad Sabetzadeh

✉ Sarah Elder
seelder@ncsu.edu

Laurie Williams
laurie_williams@ncsu.edu

[1] North Carolina State University (NCSU) Department of Computer Science College of Engineering
890 Oval Drive, Engineering Building II Raleigh, NC 27695, USA

Keywords Vulnerability Management · Web Application Security · Penetration Testing · Vulnerability Scanners

1 Introduction

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. (Alomar et al. 2020), 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...*" (Executive Order 14028 2021). 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 techniques through an empirical study of the efficiency and effectiveness of four techniques on a Java-based web application. We perform a theoretical replication[1] of work done by Austin et al. (Austin and Williams 2011; Austin et al. 2013). Since the original Austin et al. work in 2011, the vulnerability detection landscape has changed - from the applications being tested to the vulnerabilities found (Open Web Application Security Project (OWASP) Foundation 2021a; 2017; 2013b; 2013a). For example, the number of vulnerabilities in the United States National Vulnerability Database assigned to Cross-Site Scripting (XSS) has increased faster than the prevalence of other vulnerability types such as Code Injection (NVD 2021a). Our methodology and findings may also be useful to future evaluations of new vulnerability detection techniques.

We examined the 4 vulnerability detection techniques from Austin et al. (Austin and Williams 2011; Austin et al. 2013).

- **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) (Smith and Williams 2011; Austin and Williams 2011; Smith and Williams 2012; Austin et al. 2013)
- **Exploratory Manual Penetration Testing (EMPT)**: the analyst "spontaneously designs and executes tests" (ISO/IEC/IEEE 2013), searching for vulnerabilities.
- **Dynamic Application Security Testing (DAST)**: automatic tools generate and run tests based on security principles, without access to source code(Scanlon 2018).
- **Static Application Security Testing (SAST)**: automatic tools scan source code for patterns that indicate vulnerabilities (Cruzes et al. 2017; Hafiz and Fang 2016; Scandariato et al. 2013).

These four techniques can be applied during and after software implementation. Applying vulnerability detection during and after software implementation is more common in many industry settings (Cruzes et al. 2017) compared to techniques which focus on earlier

[1] A 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 (Lung et al. 2008)

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 Sonarqube[3] 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 technique 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 code[4]. 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 (US Dept of Veterans Affairs 2021; Epic Systems Corporation 2020) we know of no other study applying multiple different vulnerability detection techniques to a system this large. The only previous work comparing as many different types of vulnerability detection techniques that we are aware of is the original study by Austin et al. (Austin and Williams 2011; Austin et al. 2013). The systems in Austin et al.'s 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.(Elder et al. 2021).

We answer the following research questions:

- RQ1:What is the effectiveness, in terms of number and type of vulnerabilities, for each technique?
- RQ2: How does the reported efficiency in terms of vulnerabilities per hour differ across techniques?

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

- RQ3: What other factors should we consider when comparing techniques?

Our research makes the following contributions:

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

We are releasing our vulnerability dataset once the vulnerabilities are safely mitigated at https://github.com/RealsearchGroup/vulnerability-detection-20. We are working with

[2] https://owasp.org/www-project-zap/

[3] https://www.sonarqube.org/

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

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 paper is structured as follows. In Section 4, we provide explanations for key concepts used throughout this paper. In Section 2 we provide a brief overview of the previous work. We discuss other related work in Section 3. In Section 5 we describe the vulnerability detection techniques used in this paper. In Section 6 we discuss the SUT used in our Case Study, OpenMRS. In Section 7 we discuss the sources of data for the Case Study. In Sections 8, 9, and 10 we outline our research methodology for RQ1, RQ2, and RQ3. We discuss the equipment used in Section 11. We report our results in Section 12. We discuss our findings in Section 14. We discuss limitations of our study in Section 13. We discuss the findings in Section 14 and conclude with Section 15.

2 Previous Work by Austin et al.

Our study is a theoretical replication[5] of previous work done by Austin et al. (Austin and Williams 2011; Austin et al. 2013). The goals of the previous work are "*to improve vulnerability detection by comparing the effectiveness of vulnerability discovery techniques and to provide specific recommendations to improve vulnerability discovery with these techniques*"(Austin and Williams 2011). In their first study (Austin and Williams 2011) 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 (Austin et al. 2013) 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 technique. 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 technique, as well as the rate of vulnerabilities per hour for each technique. 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 12.1.5, and for efficiency in Section 12.2.2.

3 Related Work

Many studies have focused on a single category of techniques, such as comparisons of DAST tools or comparisons of SAST tools(Amankwah et al. 2020; Bau et al. 2012). We highlight three notable examples. In 2010, Doupé et al. (Doupé et al. 2010) 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. (Klees et al. 2018) 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

[5] A 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 (Lung et al. 2008)

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) (Delaitre et al. 2018; Okun et al. 2013; 2011; Okun et al. 2009). 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 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 techniques, but do not compare across techniques and do not examine manual techniques.

Another common comparison between vulnerability detection techniques is between static techniques that primarily analyze source code, and dynamic techniques that run tests against an active software system. Scandariato et al. (Scandariato et al. 2013) 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 (Antunes and Vieira 2009) found that different tools within the same technique 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 techniques separately and noting differences between manual and automated techniques.

Many surveys and comparisons focus on a single type of vulnerability. For example, Chaim et al. perform a survey of Buffer Overflow Detection techniques. They note that existing vulnerability detection techniques are impractical or produce too many false positives, but that emerging hybrid techniques are "promising". Liu et al. (Liu et al. 2019) survey automated, state-of-the-art techniques 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 techniques, but do not perform an empirical analysis. Fonseca et al. perform an empirical comparison of the effectiveness of different DAST tools (Fonseca et al. 2007) for finding XSS vulnerabilities. Practitioners may prioritize some types of vulnerabilities over others and these studies assist practitioners in understanding how vulnerability detection techniques 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 techniques 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 (Antunes and Vieira 2010) 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 techniques (Delaitre et al. 2018). However, the use of vulnerability detection techniques in benchmark studies may differ from how security vulnerability detection techniques would be applied in practice. For example, the three web infrastructure performance benchmark systems used by Antunes and Viera(Antunes and Vieira 2010) 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.

4 Key Concepts

In this section, we discuss key concepts necessary to understand our work.

Vulnerability: We use the definition of vulnerability from the U.S. National Vulnerability Database.[6] 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.*' (NVD 2021d).

Common Weakness Enumeration (CWE): Per the CWE website, "*CWE is a community-developed list of software and hardware weakness types.*"(MITRE 2021b). 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 techniques.

OWASP Top Ten: The OWASP Top Ten is a regularly updated list of "the most critical security risks to web applications."(Open Web Application Security Project (OWASP) Foundation 2021b). The OWASP Top Ten categories and ranking are developed by security experts based on the incidence and severity of vulnerabilities associated with different CWEs. A mapping (MITRE 2021c) 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 Miscofiguration, 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*" (van der Stock et al. 2019). 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 2019,[7] 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*" (van der Stock et al. 2019).

[6] https://nvd.nist.gov/vuln
[7] https://github.com/OWASP/ASVS/tree/v4.0.1

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 (Mozilla 2021) and sent to the application. Each HTTP message requests that some action be applied to a particular resource in the application (Fielding and Reschke 2014). 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 Fig. 1 is a message sent to the OpenMRS application to log into the application. In Fig. 1, the browser is requesting that the information in the message be `POST`ed to the URI http://127.0.0.1:8080/openmrs/login.htm. The remainder of the HTTP message contains a *representation* (Fielding and Reschke 2014) of the requested resource. The application server software then responds to the request with an HTTP message.

As an example of how HTTP messages are passed between the browser and the application server is shown in Fig. 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 Fig. 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 Fig. 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 Fig. 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 Fig. 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. The technical details of HTTP messages are most applicable for DAST tools, which we cover further in Section 5.2.2.

5 Vulnerability Detection Techniques

We begin this section by explaining common classifications of the *analysis types* used by vulnerability detection techniques. We then describe the specific *vulnerability detection*

```
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=owczzkvwskvc8n4580psmvbb
Host: 127.0.0.1:8800

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

Fig. 1 Example HTTP message

	Request		Response	
	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

Fig. 2 Example HTTP Sequence

techniques from our case study. We distinguish between the *analysis type* and the *vulnerability detection technique* since confusion may arise due to common names for vulnerability detection techniques 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.

5.1 Analysis Types

In this section, we explain four common classifications of analysis types: *automated* vs *manual* analysis, *systematic* vs *exploratory* analysis, *dynamic* vs *static* analysis, and finally *source code* analysis.

Automated vs Manual analysis: Some techniques are based on *automated* analysis performed by a tool. Manual effort may be required to use vulnerability detection tools. However, for the purpose of this study we reserve the phrase *manual analysis* to describe techniques where no automated tool is needed.

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 (ISO/IEC/IEEE 2013) 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 (ISO/IEC/IEEE 2013). Static analysis is performed on static artifacts such as source code or binaries, where the software is not actively running (ISO/IEC/IEEE 2013).

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 (McGraw 2006; Austin et al. 2013), 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.

5.2 Case Study Techniques

In this section, we provide greater detail on the four categories of vulnerability detection techniques we examine in our case study.

5.2.1 Manual Techniques

Both *manual* techniques examined in this study are *dynamic* techniques 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 techniques, particularly systematic techniques, are applied is shown in Fig. 3. This figure is based on the process for dynamic techniques presented in ISO/IEC/IEEE 29119-1 (ISO/IEC/IEEE 2013).

Systematic Manual Penetration Testing (SMPT) Specifically, SMPT is a form of scripted testing defined by ISO 29119-1 (ISO/IEC/IEEE 2013) 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 in the *Test Design & Implementation* stage, as shown in Fig. 3. The tests are then executed. In the final stage, the test results are documented and reported.

Figure 4 shows an example SMPT test case from our case study. As can be seen in Fig. 4, 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 (Smith and Williams 2012). As we will discuss further in the methodology under Section 8.1.2, 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 Fig. 4, 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".

Exploratory Manual Penetration Testing (EMPT) Exploratory Manual Penetration Testing is a *manual*, unscripted, *exploratory*, *dynamic* technique that *does not require access to*

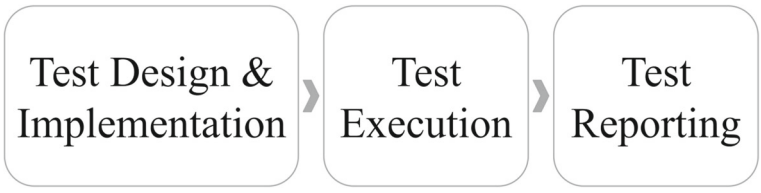

Fig. 3 Applying Manual Test Techniques (based on ISO/IEC/IEEE 29119-1)

```
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:
```

Fig. 4 Example SMPT Test Case

source code. Previous studies of functional exploratory testing have suggested that knowledge and experience may play a significant role in exploratory testing (Itkonen et al. 2013; Pfahl et al. 2014).

The process for EMPT is similar to the process shown in Fig. 3. As found by Votipka et al. (Votipka et al. 2018), 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" (Votipka et al. 2018). 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.

5.2.2 Automated Techniques

We examine two categories of *automated* vulnerability detection techniques, Dynamic Application Security Testing (DAST) and Static Application Security Testing (SAST). Figure 5 provides an overview of how automated, i.e. automatic tool-based, techniques 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.

Dynamic Application Security Testing (DAST) DAST uses *automated* tools to perform *dynamic* analysis. We only include techniques that *do not have access to source code* in the DAST category. DAST is sometimes referred to as Automated Penetration Testing (Antunes and Vieira 2010; Austin and Williams 2011; Austin et al. 2013), Black-Box Web Vulnerability Scanning (Doupé et al. 2010), Fuzzing (Klees et al. 2018), or Dynamic Analysis (Cruzes et al. 2017). 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,

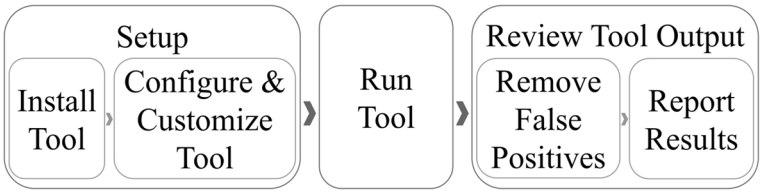

Fig. 5 Applying Tool-Based Techniques

further abbreviated as ZAP in tables),[8] is a free, open-source, dynamic analysis tool which describes itself as "the world's most widely used web app scanner"(Open Web Application Security Project (OWASP) Foundation 2021c). 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 Fig. 2. Background on HTTP is provided in Section 4. 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)[9] 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 (OWASP ZAP Dev Team 2021b; Doupé et al. 2010; Scandariato et al. 2013), 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 Fig. 1 as a potential message from a sample input, Fig. 6 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 A. 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.

[8] https://www.zaproxy.org/

[9] e.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=owczzkvwskvc8n4580psmvbb
Host: 127.0.0.1:8800

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

Fig. 6 Message from a Malformed Input (Test Case) Produced by a DAST Tool

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 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 techniques such as SMPT (Ackerman 2019). One of the motivations of our study is to understand whether this promise of "more" inputs executed "faster" by automated techniques produces equivalent or better results.

Static Application Security Testing (SAST) We use the term SAST to refer to techniques 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 (McGraw 2006; Johnson et al. 2013; Scandariato et al. 2013; Smith et al. 2015; Cruzes et al. 2017). In practice, SAST tools are less likely to be applied by security analysts (Cruzes et al. 2017; Hafiz and Fang 2016; Votipka et al. 2018) and more likely to be applied by the developers themselves (Cruzes et al. 2017). 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, while SAST-2 and SAST-3 were used to answer RQ2.

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 (McGraw 2006). Since the original work by Austin et al (Austin and Williams 2011; Austin et al. 2013), SAST tools have evolved to include additional features. For example, Sonarqube uses symbolic execution, as well as more traditional techniques such as parsing the code using regular expressions, to identify vulnerabilities (Mallet 2016). Similarly, both SAST tools used in this study employ taint analysis (Campbell 2020), although it is not clear whether taint analysis is available in the free / open-source version of Sonarqube used in this study.

6 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"(OpenMRS 2020).

6.1 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 (Delaitre et al. 2018), 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(OpenMRS 2020). 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(US Dept of Veterans Affairs 2021) and Epic(Epic Systems Corporation 2020), 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 (StackOverflow 2021; Github 2021; Cass 2021; Cass et al. 2021). Furthermore, as of July 2021(OpenMRSAtlas 2021), 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 (Austin and Williams 2011; Austin et al. 2013) 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(Tøndel et al. 2019; Purkayastha et al. 2020). 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 (Radio New Zealand (RNZ) 2021; U.S. Cybersecurity and Infrastructure Security Agency (CISA) 2021; Condon and Miller 2021; Bannister 2021).

6.2 Technical Description

OpenMRS contains 3,985,596 lines of code as measured by CLOC v1.74[10] 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 the basic reference application for OpenMRS Version 2.9. The source code for each module is available on github.[11] We compiled and ran OpenMRS using Maven and Jetty as described in the Developer's Manual(OpenMRS 2020).

6.3 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 techniques. When we reached out to OpenMRS with our results, our understanding was that SAST and DAST tools were not being used at the organizational

[10] https://github.com/AlDanial/cloc
[11] https://github.com/openmrs

level. Since then, OpenMRS's security posture has continued to mature, including more vulnerability detection efforts.

7 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.

7.1 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 techniques as part of data collection for RQ1. 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.

7.2 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. Student EMPT and SMPT data was used as part of data collection for RQ1. Researchers then analyzed students' reported efficiency scores to answer RQ2.

7.2.1 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. 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 Fig. 7. Fifty-five (55) of the 63 students whose data was used in this study provided valid survey responses. The x-axis of Fig. 7 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

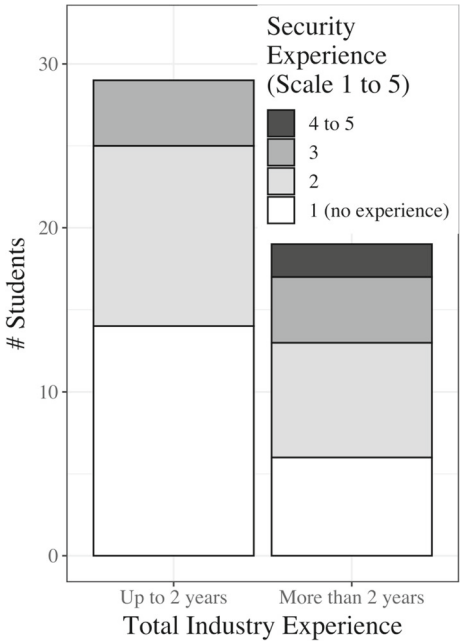

Fig. 7 Industry Security Experience of Students

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.

7.2.2 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 C. 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 (Effectiveness). The test cases were re-run and supplemented with additional test cases by researchers, as we will discuss in Section 8.1.2. 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 RQ2 (Efficiency) and RQ3 (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 (Effectiveness). Student performance and experience with EMPT informed their responses to the Comparison Assignment, which were used as part of Data Collection for RQ2 (Efficiency) and RQ3 (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 RQ2 (Efficiency) and RQ3 (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 RQ2 (Efficiency) and RQ3 (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 techniques in a free-response format. The numeric responses in the table were used to answer RQ2. Two researchers applied qualitative analysis to the students' free-response answers for RQ3.

7.3 Overview of Data Sources Per Research Question (RQ)

Table 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 1, student data is indicated with an S. Researcher data is indicated with an R. As can be seen in the table, RQ1 relied primarily on researcher efforts, although student data was used with SMPT and EMPT. Student data was used more extensively in RQ2, and was the source of the documents used in qualitative analysis for RQ3. The detailed methodology for each Research Question will be further explained in Sections 8, 9, and 10.

8 Methodology for RQ1 - Effectiveness

Our first research question is: *What is the effectiveness, in terms of number and type of vulnerabilities, for each technique?* To answer RQ1, we need a comparable set of vulnerabilities

Table 1 Data Sources R = Researchers; S = Students

		Technique			
	SMPT	EMPT	DAST	SAST	
RQ1	Applying Technique	S[a] & R	S[b]	R	R
	Review Failures	R	R	R	R
RQ2	Recorded Efficiency	S	S	S[c]	S[c]
	Data Cleaning	R	R	R	R
RQ3	Document Source	S	S	S	S
	Qualitative Coding	R	R	R	R

[a]Student SMPT results for RQ1 were reviewed and replicated by researchers

[b]Student EMPT results for RQ1 were reviewed by researchers

[c]For empirical comparison of human performance in RQ2, we use data from Students for all techniques

found by each technique. Ensuring the vulnerability counts were comparable required an extensive Data Collection process described in Section 8.1 which is split into two phases. In the first phase, *Applying the Technique* we applied each vulnerability detection technique described in Section 8.1.2 to our SUT. The initial outputs of each technique, which we will refer to as the list 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 technique in the second phase of Data Collection described in Section 8.1.3. Once the data has been collected, we analyze the results as described in Section 8.2.

8.1 Data Collection

Figure 8 provides an overview of the Data Collection process. As seen previously in Table 1, we subdivide our Data Collection process for RQ1 into two phases - *Applying the Technique* and *Reviewing the List of Failures* output by each technique. In the first phase, we collect a list of true positive failures. The first phase varies widely by technique. 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 techniques. We begin this section with a set of guidelines used across techniques. We then go into the details of how data collection was performed for each phase for each technique.

8.1.1 General Guidelines

This section provides key guidelines for the Data Collection process which we will refer to for the remainder of Section 8.1. True / False Positive Classification guidelines are used when applying automated techniques (DAST and SAST). As described in Section 8.1.3, the list of failures from each technique was assessed to ensure that the vulnerability count,

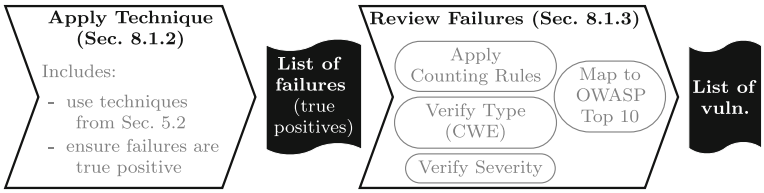

Fig. 8 Data Collection for RQ1

type, and severity are consistently evaluated using the guidelines provided in Sections 8.1.1, 8.1.1, and 8.1.1

True/False Positive Classification Guidelines A true positive alert or vulnerability is one that meets the definition of a vulnerability from Section 4. We follow a conservative policy towards true and false positive classification based on the principle of Defense-in-Depth (Joint Task Force Transformation Initiative 2013). 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 8.1.1.

Counting Guidelines The CVE program, which is the source for vulnerabilities in the NVD (NVD 2021c), also provides a set of guidelines(MITRE 2016) 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 technique on the CVE Counting Rules(MITRE 2016).[12] 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 technique. 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.

[12] The 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

Vulnerability Type Guidelines The vulnerability types assigned to each vulnerability are based on two systems - CWE and OWASP Top Ten - which are described in Section 4. How each vulnerability was initially assigned a CWE varied by technique. 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 5.2.2. When the malformed input is executed against the SUT, it triggers an error message revealing sensitive information (CWE-209.[13]) 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 (XSS[14]) 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 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 (MITRE 2021b) to find an appropriate CWE. The CWE mapping to the OWASP Top Ten (MITRE 2021c) as well as more general guidelines such as the list of "Weaknesses for Simplified Mapping of Published Vulnerabilities"(MITRE 2022) and relationships between CWEs provided by the CWE system (MITRE 2021a) 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(MITRE 2021c). 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 Section 4, 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.

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 8.1.2, 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".

[13] https://cwe.mitre.org/data/definitions/209.html
[14] https://cwe.mitre.org/data/definitions/79.html

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 technique. In our experience, if a tool or technique 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 12.1.3.

8.1.2 Applying the Technique

The process of applying of each technique is slightly different, as described in Section 5. We discuss details of how each technique was applied for our Case Study for SMPT in Section 8.1.2, for EMPT in Section 8.1.2, for DAST in Section 8.1.2, and for SAST in Section 8.1.2.

SMPT To apply SMPT as part of RQ1, 131 test cases were manually written and executed by a combination of *students* (S) and *researchers* (R), as shown in Table 1. The test cases were based on the OWASP Application Security Verification Standard (ASVS) (van der Stock et al. 2019). As described in Section 4, ASVS has three levels of controls. However, "*Level 1 is the only level that is completely penetration testable using humans*" (van der Stock et al. 2019). In addition to the ASVS Level 1 controls, students and researchers used knowledge of OpenMRS and documentation available on the OpenMRS wiki[15] 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 (van der Stock et al. 2019) 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 7.2.2. 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 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.

EMPT As shown in Table 1, for RQ1, EMPT was applied by *students* (S), according to the assignment outlined in Section 7.2.2. Data from 62 students was used as part of data

[15] https://wiki.openmrs.org/

collection for EMPT in RQ1, 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 8.1.3. 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.

An important factor in exploratory testing is ensuring that individuals have sufficient knowledge and experience (Itkonen et al. 2013; Itkonen and Mäntylä 2014; Votipka et al. 2018) because EMPT is based on knowledge and experience. The students had limited security experience at the start of the course, as discussed in Section 7.2. However, our results suggest that the students had sufficient experience by the time of the EMPT assignment to be effective.

DAST As shown in Table 1, researchers (R) applied DAST for RQ1. As shown in Fig. 5, 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 techniques.

Setting Up and Running the DAST Tools: As discussed in Section 5.2.2, 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 techniques, 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 13. 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 10 of Appendix A.

Two researchers performed multiple trials of each tool in different configurations to determine 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 5.2.2. As shown previously in Fig. 2 and discussed in Section 4, 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 Fig. 4, 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 Fig. 4, 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 8.1.1. 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" (OWASP ZAP Dev Team 2021a). 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 (Lombard et al. 2002; Votipka et al. 2018) 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 12.1.1.

SAST As seen in Table 1, applying SAST began with researchers (R) running the SAST tools on the SUT using the default security rules. As shown in Fig. 5 and described in Section 5.2.2, 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 10 in Appendix A. 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 8.1.1. We computed Cohen's Kappa(Cohen 1960) 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"(SonarSource 2019).

8.1.3 Reviewing the List of Failures

Each technique produces different outputs, which we refer to as "failures". For example, systematic, dynamic techniques 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 technique even though it is no more effective than another technique. To resolve these potential counting differences, we take the list of failures from each technique and apply the Counting Rules described in Section 8.1.1 to determine the number of vulnerabilities found by each technique. 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 technique output. As shown in Table 1, researchers (R) reviewed the list of failures for all techniques.

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 8.1.1. 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 8.1.1. 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 8.1.1.

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 8. Researchers used the counting rules specified in Section 8.1.1 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 8.1.1, 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 8.1.1.

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.

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 8.1.1. 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 Fig. 6 would contain two distinct XSS vulnerabilities. Discussions of duplication and deduplication 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 8.1.1. The severity measures provided by the DAST tools were then used to distinguish less severe vulnerabilities from more severe vulnerabilities as described in Section 8.1.1.

SAST Researchers (R) determined the number of distinct vulnerabilities indicated by the SAST alerts using the counting rules from Section 8.1.1. The researchers reviewed the vulnerability CWE assignments provided by the SAST tools to ensure their accuracy following the guidelines from Section 8.1.1. Additionally, severity measures provided by the SAST tools were used to distinguish less severe vulnerabilities from more severe vulnerabilities as described in Section 8.1.1.

8.2 Data Analysis

Once we had a comparable set of vulnerabilities, we calculated the number of vulnerabilities found by each technique 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 (Morrison et al. 2018). We used vulnerability counts and types to answer RQ1.

9 Methodology for RQ2 - Efficiency

For RQ2, we address the question *How does the reported efficiency in terms of vulnerabilities per hour differ across techniques?*. 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 (Kirk 2013). 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.

9.1 Data Collection

We collected efficiency information recorded by students (S), which we discuss in Section 9.1.1. Researchers (R) then performed data cleaning, as we discuss in Section 9.1.2 before the data could be analyzed (Fig. 9).

9.1.1 Recorded Efficiency

To quantify efficiency, we started with information provided by the students (S) as shown in Table 1. As discussed in Section 7.2, 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 7.2.2 which appear verbatim in Appendix C. 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. H ence we use the average VpH across all participating members of each team for EMPT. For RQ2 we exclude data from students whose team members did not participate in the study as as discussed in Section 7.2.

9.1.2 Data Cleaning

Once we have collected efficiency data, as shown in Table 1 the researchers (R) performed data cleaning as needed. We formally identified outliers for each technique using the median absolute deviation and median (MADN)(Wilcox and Keselman 2003; Kitchenham et al. 2017), applying MADN to the VpH scores for each technique. We removed outliers where the MADN was higher than 2.24, the threshold recommended in the literature(Wilcox and Keselman 2003; Kitchenham et al. 2015). 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 7.2.2 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 technique.

Fig. 9 Data Collection for RQ2

9.2 Data Analysis

With outliers removed, we retained 12 efficiency scores for each technique. We performed a statistical comparison to determine if the average efficiency across the groups for each technique was higher or lower than the average efficiency for other techniques. We first applied the Shapiro-Wilk test (Razali et al. 2011) to the data for each technique to assess normality. Based on the output of the Shapiro-Wilk test, we used Bartlett's test for homogeneity of variance (Bartlett 1937). Our case study data was normal, but the variance differed across techniques. Based on the results of the normality and homogeneity of variance tests, we chose to apply the Games-Howell test (Games and Howell 1976; Kirk 2013) to perform pairwise comparison across the different vulnerability detection techniques and determine which techniques were different. The Games-Howell test adjusts the p-value for multiple comparisons.

10 Methodology for RQ3 - Other Factors

Once the students had experience with the four vulnerability detection techniques, the students (S) were asked to reflect on their experiences with each technique and to compare the techniques as part of the "comparison assignment" described in Section 7.2.2. Students were instructed to discuss tradeoffs between the techniques, and "Based upon your experience with these techniques, compare their ability to efficiently and effectively detect a wide range of types of exploitable vulnerabilities" as shown in Appendix C. The student responses to the comparison assignment were the source documents for RQ3 as shown in Table 1. The comparison assignment was answered at the team-level, and so the data used for RQ3 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 techniques. 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"(Corbin and Strauss 2008). 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 and RQ2, and may be informative for future work.

11 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 Lab[16] (VCL) at our university, North Carolina State University

[16] https://vcl.apache.org

Table 2 Results Summary

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

[d]The total vulnerability count includes both "more severe" and "less severe" vulnerabilities as described in Section 8.1.1

[e]One category within the OWASP Top Ten is outside the scope of this study. Maximum possible coverage is 9.

[f]The difference in efficiency between SMPT and DAST is not statistically significant.

(NCSU).[17] 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 RQ2 and RQ3, leveraged the VCL images. Additional resources were needed when answering RQ1 to improve system coverage, including larger VCL instances, Virtualbox VMs based on the VCL images, and a large desktop machine with 24 CPUs, 32G RAM, and 500G disk space. Additional information on the systems is in Appendix D.

12 Results

In this section, we describe our results. Table 2 provides a high-level summary of the numeric results for RQ1 - *What is the effectiveness, in terms of number and type of vulnerabilities, for each technique?* and RQ2 - *How does the reported efficiency in terms of vulnerabilities per hour differ across techniques?*. Detailed results for RQ1 and RQ2 are provided in Sections 12.1 and 12.2 respectively. We provide our qualitative results for RQ3 - *What other factors should we consider when comparing techniques?* in Section 12.3.

12.1 RQ1 - Technique 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 technique?*. First, we go over information specific to automated, i.e. tool-based, techniques: 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 technique, and the types of vulnerabilities identified by each technique based on the CWE and OWASP Top Ten. We also include the severity of the vulnerabilities found.

12.1.1 True and False Positive Tool Alerts

We examined two tool-based techniques in this study, SAST and DAST, for which we could calculate the precision of the tools. As noted in Section 8.1.2, for tool-based techniques, two researchers classified the alerts produced by the tool as true or false positive. We calculated

[17]https://vcl.ncsu.edu

their inter-rater reliability and present the results in Section 12.1.1. 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 12.1.1 and shown in Table 3.

Reviewer Agreement We calculated the inter-rater reliability of the reviewers for SAST and DAST using Cohen's Kappa (Cohen 1960). 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 (Feinstein and Cicchetti 1990; Cicchetti and Feinstein 1990; Feng 2013). 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 (Lombard et al. 2002; Votipka et al. 2018). Therefore the remaining alerts were divided between the researchers to review as described in Section 8.1.2. 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.

True / False Positives and Precision Table 3 shows the Total Alerts (Tot. Alrt.), False Positives (FP), and Precision (Prec.) for automated, i.e. tool-based, techniques. Table 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. (Austin and Williams 2011; Austin et al. 2013) which we will compare with our results in Section 12.1.1.

On the left, subtable 3.a provides the Tot. Alrt., FP, and Prec. for DAST. On the right, subtable 3.b provides the Tot. Alrt., FP, and Prec. for SAST 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 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 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 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 8.1.1. The Tool-Assigned CWE, shown in the first column of Table 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*[18]. 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 8.1.3. 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 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 5.2.2, and false positives tended to be associated with specific rules. As we can see in Table 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 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

[18] https://cwe.mitre.org/data/definitions/89.html

Table 3 Total Alerts (Tot. Alrt.), False Positives (FP), and Precision (Prec) for the Current Study Compared with Austin et al.

	(a) DAST					(b) SAST					
	Current Study			Austin et al.		Current Study			Austin et al.		
	Total (M)	ZAP (M)	DA-2 (M)	Total (E)	Total (T)	Total (M)	Sonar (M)	SA-2 (M)	Total (E)	Total (T)	Total (P)
Tot. Alrt.	3414	550	2862	735	37	962	698	264	5036	2315	1789
FP	2612	28	2597	25	15	20	4	16	3715	2265	1644
Prec.	0.23	0.95	0.09	0.97	0.59	0.98	0.99	0.94	0.26	0.02	0.08

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

Table 4 DAST Alerts

	OWASP ZAP					DAST-2				
Tool-Assigned CWE	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	14
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					

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.

Tool False Positives and Precision Comparison with Austin et al. Table 3 also shows the tool precision reported by Austin et al. (Austin and Williams 2011; Austin et al. 2013) for comparison with the current study. As described in Section 2, 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 technique. 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, 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) (Delaitre et al. 2018) where the precision and recall of SAST tools for Java was far higher than the precision and recall reported in previous work (Austin and Williams 2011; Austin et al. 2013).

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. (Austin and Williams 2011; Austin et al. 2013) also found that entire categories of alerts could be labeled true or false positive, similar to the results shown in Table 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 (12.1.1) 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 techniques may be found in Section 12.1.5. Comparison with Austin et al. on efficiency measures may be found in Section 12.2.2.

12.1.2 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 technique and tool in Table 5. The main results for each technique are shaded gray, white-shaded columns indicate the results for each of the DAST and SAST tools.

In the first row of Table 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 5 for DAST and SAST "True Positive Failures" is the sum of all true positive alerts for the technique.

Table 5 Vulnerability Counts

	SMPT	EMPT	DAST (Total)	ZAP	DA-2	SAST (Total)	Sonar	SA-2
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 Tech./Tool	11	157	13	8	4	822	588	225

The second row of Table 5 shows the total number of vulnerabilities indicated by the failures. The vulnerability counts are determined by applying our counting rules described in Section 8.1.1. 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 technique, *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 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 14.

The fourth row of of Table 5, labeled "Vuln. Unique to Tech./Tool", shows the number of vulnerabilities that were only found by each technique or tool. It may be helpful to consider the "Vuln. Unique to Technique/Tool" as *the number of vulnerabilities we would have missed if we had not used the technique 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 technique. One (1) vulnerability was found by all techniques, 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.* [19]Ten (10) vulnerabilities were found by both of the SAST tools, but by no other technique. 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 techniques. Each technique and tool found vulnerabilities that were not found using other techniques and tools.

12.1.3 Vulnerability Severity

As discussed in Section 8.1.1, we reviewed vulnerabilities where the same tool or technique 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.

[19]https://cwe.mitre.org/data/definitions/7.html

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*.[20] 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 `@RequestParameter` 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 mechanisms.[21] 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*[22] 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 Scripting*.[23] 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*.[24] 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.

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".

[20] https://cwe.mitre.org/data/definitions/52.html
[21] https://docs.spring.io/spring-security/site/docs/5.0.x/reference/html/csrf.html
[22] https://cwe.mitre.org/data/definitions/352.html
[23] https://cwe.mitre.org/data/definitions/79.html
[24] https://cwe.mitre.org/data/definitions/404.html

12.1.4 Vulnerability Type (OWASP Top Ten)

Table 6 shows the distribution of the vulnerabilities found by each technique 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 (MITRE 2021c), as discussed in Section 8.1.1. The vulnerability counts for the specific CWE types within each Top Ten category are available in Appendix E.

The leftmost column of Table 6 indicates the OWASP Top Ten category. Columns two through five indicate the vulnerabilities that were found for each technique. Column six of Table 6 shows the total vulnerabilities found within each Top Ten category across all techniques. 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.

This study is an evaluation of techniques 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 6 of the tools examined found vulnerabilities in the OWASP Top Ten Category for Vulnerable and Outdated Components (A06), further suggesting that different techniques and categories of techniques are useful for finding different types of vulnerabilities.

Table 6 More Severe Vulnerability Count based on OWASP Top Ten (2021) (Total count, including both more and less severe vulnerabilities)

OWASP Top Ten (2021) Category	SMPT	EMPT	DAST	SAST	Total Found
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[i] (15)	19[i] (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 Technique	32 (37)	165 (185)	17 (23)	142 (823)	329 (1033)

[g]One more severe vulnerability found using SMPT mapped to both A01 and A04 through two different CWEs.

[h]One more severe vulnerability found using SMPT mapped to both A08 and A10 through two different CWEs.

[i]14 more severe vulnerabilities found using SAST were associated with two CWEs, one of which mapped to A05 while the other CWE was not mapped to the OWASP Top Ten. We only include these vulnerabilities under A05 since they are not "No Mapping"

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 techniques.

EMPT was one of the most effective techniques 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 technique for finding vulnerabilities associated with Security Misconfiguration (A05). Only one vulnerability found by other techniques was found by SAST in the entire dataset. Hence SAST should not be seen as something that can substitute for other techniques, or be substituted for by other techniques. 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 E, 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.

12.1.5 Effectiveness Comparison with Austin et. al.

A comparison with the previous study by Austin et al. (Austin and Williams 2011; Austin et al. 2013) of vulnerability counts for each vulnerability type is shown in Table 7. The first column of Table 7 indicates the technique (Tech.), the second column of Table 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 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(Austin and Williams 2011; Austin et al. 2013) include less severe vulnerabilities; and the vulnerability counts from the current study in Table 7 also include those that are less severe.

Our effectiveness with SMPT in the current study was similar to Austin et al.(Austin and Williams 2011; Austin et al. 2013). Austin et al. found more vulnerabilities in Open-

Table 7 Vulnerability type comparison with Austin study

	OWASP Top Ten Descr.	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)	Not Mapped to Top Ten	TOTAL
	OWASP Top Ten #	A01	A02	A03	A04	A05	A06	A07	A08	A09	A10	NM	TOT
Tech.	Study	SUT											
SMPT	Curr. Study	M (Total)	2	1	5	8	5		13	1	3	1	1	37
	Austin et al.	Total	4	3	17	4			9		99			136
		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
	Austin et al.	Total		1	8	1			1				2	13
		E			8	1			1				2	12
		T												
		P		1										1
DAST	Curr. Study	M (Total)	1	2	11	2	6		1	1				23
	Austin et al.	Total	502		221		9							732
		E	485		221		4							710
		T	17				5							22
SAST	Curr. Study	M (Total)	261	4	58	36	15		2	11			436	822
	Austin et al.	Total	93		1190	124	1				22		86	1516
		E	36		1155	122	1						7	1321
		T	13		35	2								50
		P	44								22		79	145

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

EMR 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 techniques 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.(Austin et al. 2013; Austin and Williams 2011) for EMPT is more complicated for methodological reasons. For DAST and SAST the analysis for RQ1 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.(Austin et al. 2013; Austin and Williams 2011). The procedure for EMPT differed between studies to take full advantage of the data generated by students. As we note in Section 8.1, the 229 vulnerabilities in Table 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 (Finifter et al. 2013). However, the use of smaller, internal teams such the 6-person team used to apply EMPT to OpenEMR in Austin et al. (Austin et al. 2013; Austin and Williams 2011), or even individual hackers working alone on EMPT as was done for Tolven and PatientOS is also not uncommon (Alomar et al. 2020; Votipka et al. 2018). The high number of students who applied EMPT for RQ1 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. (Austin et al. 2013; Austin and Williams 2011). 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 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 12.2.2.

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. (Austin et al. 2013;

Austin and Williams 2011) 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 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 techniques.

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

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 technique had been excluded from the analysis, at least 4 and up to 588 vulnerabilities would have been missed.

12.2 RQ2 - Efficiency

In this section, we discuss the results for our question *How does the reported efficiency in terms of vulnerabilities per hour differ across techniques?* The data was collected from students, as described in Section 9.

12.2.1 Efficiency Results

Boxplots for each technique's efficiency in terms of Vulnerabilities per Hour (VpH) are shown in Figure 10. 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.

Table 8 shows the Games-Howell test results for comparing each pair of techniques. As can be seen in the table, EMPT is significantly more efficient than every other technique ($p < 0.05$ for all comparisons). SAST is the second-most-efficient technique ($p < 0.05$ 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 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.

12.2.2 Efficiency Comparison with Austin et al.

Table 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 9 indicates the median and average (avg) across the groups performing the task.

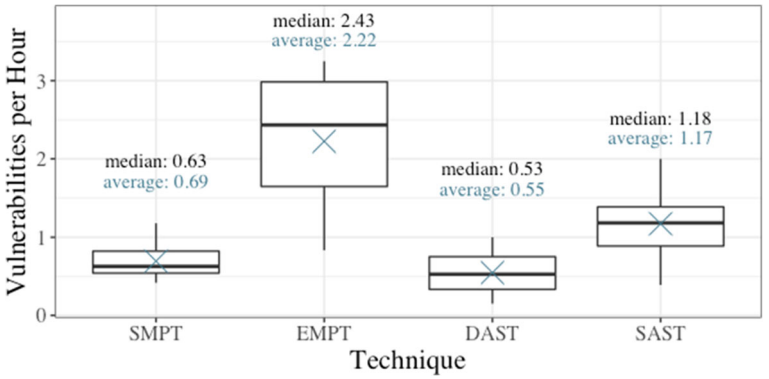

Fig. 10 Vulnerability Detection Technique Efficiency

The differences in efficiency are not only due to differences between student performance and researcher performance. When applying SAST for RQ1 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 8.1.1 compared with alerts or true positive failures. As shown previously in Table 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 technique.

RQ2 - How does the reported efficiency in terms of vulnerabilities per hour differ across techniques?

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.

12.3 RQ3 - 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 techniques?*. We discarded

Table 8 Games-Howell t-test of Efficiency Scores

Techniques	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 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

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 techniques rather than the further subdivision used in the rest of this paper (i.e., EMPT and SMPT are "manual" techniques, while SAST and DAST are "automated" techniques). We discuss our results for RQ3 in terms of automated and manual techniques 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.

12.3.1 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 techniques, discussions of effort for automated techniques were mixed.*

Every response mentioned human effort beyond VpH. Effort was perceived as a disadvantage for manual techniques (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 techniques, while 0 teams mentioned advantages relating to effort for either of the manual techniques. In contrast, for one or both automated techniques, 4 teams mentioned effort as a disadvantage, 3 teams mentioned effort as an advantage, and 2 teams mentioned both advantages and disadvantages.

12.3.2 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 techniques were seen as requiring more time. For automated techniques, some teams considered time an advantage while others considered time a disadvantage. Additionally, students conjectured that the efficiency of each technique may change if the techniques were applied over a longer timeframe.*

Similarly to effort, time was frequently seen as a disadvantage for manual techniques, 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 techniques, 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 technique. Responses for automated techniques 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 technique's effectiveness and efficiency would change if the techniques 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*".

12.3.3 Expertise

Summary: *Many types of expertise are needed to apply vulnerability detection techniques, particularly EMPT. Similar to findings from other works* (Itkonen et al. 2013; Itkonen and Mäntylä 2014; Votipka et al. 2018)*, 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 techniques, while 6 mentioned expertise when discussing EMPT, and one team mentioned expertise when discussing SMPT and manual techniques 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 technique 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 (Itkonen et al. 2013; Itkonen and Mäntylä 2014; Votipka et al. 2018). 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.

RQ3 - What other factors should we consider when comparing techniques?

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 techniques. Some teams considering effort or time an advantage of automated techniques, while others considered them a disadvantage. Expertise was associated with manual techniques, particularly EMPT, more than automated techniques.

13 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 (Cook and Campbell 1979; Feldt and Magazinius 2010; Wohlin et al. 2012). 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 (Cook and Campbell 1979; Wohlin et al. 2012; Ralph and Tempero 2018). In our study, the two primary outcome constructs we intended to observe were effectiveness (RQ1) and efficiency (RQ2). The specific proxy measures we use determine the outcome are the number and type of vulnerabilities (RQ1) and Vulnerabilities per Hour (RQ2). The cause construct (independent variable) is the vulnerability detection technique being used. Our treatments to represent this cause construct are the four categories of vulnerability detection techniques. These outcomes and treatments were previously used by Austin et al. (Austin and Williams 2011; Austin et al. 2013).

13.1 Conclusion Validity

Conclusion Validity is about whether conclusions are based on statistical evidence (Cook and Campbell 1979; Wohlin et al. 2012). While we have empirical results for RQ1, a single case study is insufficient to draw *statistically significant* conclusions for effectiveness. The measures used to evaluate effectiveness in RQ1 are based on the number of vulnerabilities found by applying each technique thoroughly and systematically. Measuring effectiveness with statistical significance would require the application of all four techniques to at least 10-20 additional applications (Kirk 2013). Applying all techniques to 10-20 similarly-sized SUT is impractical given the effort required to apply these techniques to a single application. To mitigate this threat to validity, we performed extensive review of the vulnerability counts, using the guidelines in Section 8.1.1, and at least two individuals were involved in the review process for each technique to verify the accuracy of the results. For efficiency (RQ2) the measure used, VpH, was evaluated by having more individuals apply the technique to a subset of the application, enabling us to evaluate the results with statistical significance.

13.2 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 (Cook and Campbell 1979; Wohlin et al. 2012; Ralph and Tempero 2018). Our measures for RQ1, the number and type of vulnerabilities, are commonly used measures of (in)security in academia and industry (Klees et al. 2018; Delaitre et al. 2018; Okun et al. 2013; 2011; 2010; Okun et al. 2009), including by the U.S. National Institute of Standards and Technology (NIST) Software Assurance Metrics and Tool Evaluation (SAMATE) program discussed in Section 3. 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 8.1.2, 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 8.1.1.

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 technique

on a subset of the application, and we could determine whether differences in efficiency were statistically significant. As discussed in Section 12.3.2, factors such as the length of time required to apply a technique may also be helpful in understanding efficiency. Applying SAST and DAST more comprehensively as part of RQ1 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 technique. Adding RQ3, which was not included in Austin et al.'s work (Austin and Williams 2011; Austin et al. 2013), allowed us to better understand how other factors such as time spent applying a technique were perceived by students, helping to mitigate this threat to validity.

13.3 Internal Validity

Internal Validity concerns whether the observed outcomes are due to the treatment applied, or whether other factors may have influenced the outcome (Cook and Campbell 1979; Feldt and Magazinius 2010). 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 techniques have also used a spider (Doupé et al. 2010; Scandariato et al. 2013).

Another threat to internal validity for this study is that the student data used for RQ2 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 techniques are used. Additionally, as shown by RQ3, students' reported numeric efficiency was not necessarily indicative of the time and effort the students perceived was required for each technique.

Another limitation with RQ2 is posed by equipment constraints for the graduate level class. Additional equipment was used to mitigate this risk for RQ1. To mitigate the risk that memory-related processing issues would negatively impact student efficiency (RQ2), 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 RQ3 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. 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 technique.

Finally, although both DAST tools examined in RQ1 and RQ2 were the same, we did not use the Sonarqube SAST tool for RQ2, 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 RQ2 we control for expertise by comparing efficiency scores from the same group of individuals.

13.4 External Validity

External Validity concerns the generalizability of our results (Cook and Campbell 1979; Feldt and Magazinius 2010; Wohlin et al. 2012). 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(Cowan et al. 2000; Nagarakatte et al. 2009). As discussed in Section 6.1, 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 (Delaitre et al. 2018), 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 14.3.2.

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(Delaitre et al. 2018; Okun et al. 2013; 2011) 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 technique 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 (Imtiaz et al. 2019). However, true positive vulnerabilities require more analysis since the alert must be resolved, while false positives can be ignored. In the SAST assignment[25], 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 experience[26], 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 8.1.2 was more extensive, reviewing the test cases for RQ1 was intended to ensure a more accurate test process and resulting vulnerability count. Since time was not considered in RQ1, the additional time spent on reviewing test cases for RQ1 would not impact the results.

[25] the full text of the assignment is available under Project Part 1 in Appendix C

[26] the first author has over 2 years of industry testing experience

14 Discussion

Section 14.1 and Section 14.2 should be considered together. In Section 14.1, 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 14.2 we discuss how the availability or limitation of *resources*, specifically Expertise, Time, and Equipment, may also impact which technique should be used. There may be tradeoffs between techniques when practitioners focus on one *objective* over another. A manager may want to avoid manual techniques in order to reduce the perceived effort for their team (Section 14.1.4). However, in our context automated techniques were less effective in terms of the coverage of different vulnerability types and the severity of vulnerabilities found (Section 14.1.3). There may also be tradeoffs between *objectives* and *resources* For example, as we note in Section 14.1, 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 14.2, practitioners may need to look to other techniques.

In Section 14.3 we go over findings that have additional implications relevant to research and other evaluations of vulnerability detection techniques. We would encourage any researcher comparing vulnerability detection techniques to also be aware of our findings in Sections 14.1 and 14.2.

14.1 Organizational Objectives

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

14.1.1 Specific Vulnerability Types (Effectiveness)

Practitioners may be more concerned about a certain type or class of vulnerabilities. For example, as seen in Table 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(Delaitre et al. 2018; Bau et al. 2012), which tool is most effective may vary across domains. Practitioners should look to vulnerability detection technique evaluations in their domain.

> If an organization is trying to target a specific type of vulnerability, they should focus their vulnerability detection efforts on techniques 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 6.

14.1.2 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 6. On the other hand, as seen in the comparison with the prior work by Austin et al. (Austin and Williams 2011) shown in Table 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 technique that effectively covers important types of vulnerabilities, a more systematic technique, such as SMPT, may be more effective.

14.1.3 Automation (Efficiency)

When considering automated tools, practitioners should note that automated techniques may not inherently be more efficient than manual techniques. An organization may choose to use automated tools for other reasons such as the need to integrate automated tools with continuous deployment pipelines (Rahman et al. 2015). Our results suggest that manual techniques are comparable to or better than automated techniques in terms of efficiency.

Our results suggest that manual techniques are comparable or better than automated techniques in terms of efficiency. If an organization is considering whether to use an automated technique over a manual one, they should not assume that the automated technique will be more efficient.

14.1.4 Percieved 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 (Davis 1989). 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 techniques. As we discuss in Section 12.3, *time* and *effort* was predominantly seen as negative for manual techniques but views of time and effort were mixed for automated techniques. Similarly, *expertise* was associated with manual techniques (SMPT and EMPT) more than automated techniques (DAST and SAST). The actual times recorded for manual techniques were, on average, no longer than the times recorded for automated techniques. Hence our findings suggest that time was *perceived* as a disadvantage of manual techniques even if they actually required no more time than automated techniques. Pfahl et al's (Pfahl et al. 2014) 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 (Smith et al. 2020). As noted by Gonçales et al.(Gonçales et al. 2021), 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 techniques.

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

14.2 Resources to Consider

The resources below represent factors that should be considered when selecting a vulnerability detection technique for a system such as OpenMRS. Two of the three resources were highlighted by student responses in RQ3, while the third resource provided a much more severe limitation on our experiment than anticipated.

14.2.1 Expertise

As noted by the students in RQ3 and supported by prior work (Itkonen et al. 2013; Itkonen and Mäntylä 2014), expertise plays a role in vulnerability detection, particularly EMPT. The effectiveness of the students as shown in Table 6; as well as their efficiency shown in Fig. 10 is promising. Students with an introductory knowledge of Security were efficient and effective with EMPT. Anecdotally based on our experience with RQ1 as well as in student responses in RQ3, experience may also impact the efficiency and effectiveness of automated techniques more than we expected.

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

14.2.2 Time

As noted by the students in RQ3, the amount of time an analyst has available to apply a technique may influence the efficiency and effectiveness of the technique. 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 12.3.2, some techniques such as DAST may perform better if practitioners have an extended timeframe in which to apply the technique. In contrast, as discussed in our comparison with Austin et al. in Section 12.1.5, 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 technique should be considered when selecting the technique. DAST, in particular, may benefit from a longer timeframe.

14.2.3 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 13.3, 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. Austin et al., as well as other studies of industry tools (Amankwah et al. 2020; Scandariato et al. 2013; Bau et al. 2012) do not mention equipment constraints. Where the equipment used in the experiment is mentioned (Amankwah et al. 2020), 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 techniques, OpenMRS is less than 4 million lines of code - smaller than many industry software systems (Desjardins 2017; Anderson 2020). 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.

14.3 Implications for Evaluating Vulnerability Detection Techniques

The resource concerns highlighted in Section 14.2 should be considered not only by practitioners but by researchers evaluating vulnerability detection techniques. Our results also have several implications specific to future evaluations of vulnerability detection techniques

14.3.1 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 techniques. 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 techniques. As can be seen in Table 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 (Klees et al. 2018) 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 (Dambra et al. 2020), 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 techniques. It should not be assumed that tools or techniques use the same counting rules.

14.3.2 SAST tools had fewer False Positives than expected.

High False Positive counts have historically been considered a drawback of SAST tools(Imtiaz et al. 2019; Johnson et al. 2013; Scandariato et al. 2013; Smith et al. 2015; Hafiz and Fang 2016). 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 (Delaitre et al. 2018). 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 techniques, and whether false positive counts have improved in other contexts.

15 Conclusions and Future Work

The motivation for this paper came from practitioner questions about which vulnerability detection techniques 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 techniques such as the more common use of symbolic execution and taint tracing in SAST tools(Mallet 2016; Campbell 2020) 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 technique. The main finding of Austin et al. still holds - each approach to vulnerability detection found vulnerabilities NOT found by the other techniques. If the goal of an organization is to find “all” vulnerabilities in their system, they need to use as many techniques as their resources allow.

We hope to leverage the lessons learned from this experience in future work. In an empirical comparison of vulnerability detection techniques on a large-scale application, we found that even simple measures, such as vulnerability count, are not entirely objective and require strict guidelines for the count to be consistent and replicable. More research is needed to understand how vulnerability detection techniques compare in terms of other measures, such as exploitability, as well as how to apply those measures in the context of large-scale web applications. Additionally, an emerging class of automated vulnerability detection techniques, sometimes referred to as “hybrid” techniques, combines static analysis with aspects of dynamic analysis (Chaim et al. 2018; Liu et al. 2019) and is considered “promising”(Chaim et al. 2018). While out of scope for the replication study, we look forward to expanding our comparison of vulnerability detection techniques to include these and other tools and techniques.

Appendix A: Automated Technique CWEs

Table 10 CWEs covered in the rules implemented by automated techniques

CWE ID	CWE Name	OWASP Top Ten	ZAP	DA-2	Sonar	SA-2
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
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
336	Same Seed in Pseudo-Random Number Generator (PRNG)	A02			X	X

Table 10 (continued)

CWE ID	CWE Name	OWASP Top Ten	ZAP	DA-2	Sonar	SA-2
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
610	Externally Controlled Reference to a Resource in Another Sphere	A03				X
643	XPath Injection	A03				X
917	Expression Language Injection	A03				X

Table 10 (continued)

CWE ID	CWE Name	OWASP Top Ten	ZAP	DA-2	Sonar	SA-2
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			
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

Table 10 (continued)

CWE ID	CWE Name	OWASP Top Ten	ZAP	DA-2	Sonar	SA-2
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
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		

Table 10 (continued)

CWE ID	CWE Name	OWASP Top Ten	ZAP	DA-2	Sonar	SA-2
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		
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

Table 10 (continued)

CWE ID	CWE Name	OWASP Top Ten	ZAP	DA-2	Sonar	SA-2
242	Use of Inherently Dangerous Function	NM				X
252	Unchecked Return Value	NM				X
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	
493	Critical Public Variable Without Final Modifier	NM			X	

Table 10 (continued)

CWE ID	CWE Name	OWASP Top Ten	ZAP	DA-2	Sonar	SA-2
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
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

Table 10 (continued)

CWE ID	CWE Name	OWASP Top Ten	ZAP	DA-2	Sonar	SA-2
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

Appendix B: 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 7.2

Appendix C: 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 [ ].

C.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.**

C.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).

C.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].

C.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].

Appendix D: Equipment Specifications

In this appendix we provide additional details of the equipment used in our case study. As noted in Section 11, a key resource used in this project was the school's Virtual Computing Lab[27] (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 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

[27] https://vcl.apache.org/

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 x2go[28]. 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 Eclipse[29]. The plugin allows 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. (Austin and Williams 2011; Austin et al. 2013), 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 E: All CWEs Table

Table 11 shows the CWE for high and medium severity vulnerabilities found. Table 12 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 11 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 Table5 may be lower than the sum of each column in Table 11.

[28] https://wiki.x2go.org/doku.php
[29] https://www.eclipse.org/ide/

Table 11 CWEs associated with *more severe* Vulnerabilities

CWE	Top Ten	SMPT	EMPT	DAST		SAST		Total
				ZAP	DA-2	Sonar	SA-2	
A01 Broken Access Control								
922 - Insecure Storage of Sensitive Information	A1	1						1
200 - Exposure of Sensitive Information to an Unauth. Actor	A1		2					2
601 - URL Redirection to Untrusted Site ('Open Redirect')	A1						9	9
285 - Improper Authorization	A1	1	13					13
22 - Path Traversal	A1						19	19
A02 Cryptographic Failures								
326 - Inadequate Encryption Strength	A1			1				1
319 - Cleartext Transmission of Sensitive Information	A2	1	1					1
327 - Use of a Broken or Risky Cryptographic Algorithm	A2					2		2
A03 Injection								
643 - XPath Injection	A3						1	1
89 - SQL Injection	A3						4	4
20 - Improper Input Validation	A3	3	19		2			21
79 - Cross-site Scripting	A3	2	100	3	7		19	124
A04 Insecure Design								
269 - Improper Privilege Management	A4		1					1

Table 11 (continued)

CWE	Top Ten	SMPT	EMPT	DAST		SAST		Total
				ZAP	DA-2	Sonar	SA-2	
313 - Cleartext Storage in a File or on Disk	A4						1	1
770 - Allocation of Resources Without Limits or Throttling	A4	1	1					1
419 - Unprotected Primary Channel	A4	2	1					2
807 - Reliance on Untrusted Inputs in a Security Decision	A4					3		3
73 - External Control of File Name or Path	A4						4	4
598 - Use of GET Request Method With Sensitive Query Strings	A4	2	5		1			6
A05 Security Misconfiguration								
548 - Exposure of Information Through Directory Listing	A5			1				1
614 - Sensitive Cookie in HTTPS Session Without 'Secure' Attribute	A5	1	1					1
16 - Configuration	A5	1	1	1				3
611 - Improper Restriction of XML External Entity Reference	A5					13	1	14
CWE	TT	SMPT	EMPT	ZAP	DA-2	Sonar	SA-2	Total
A06 Vulnerable and Outdated Components								
A07 Identification and Authentication Failures								
308 - Use of Single-factor Authentication	A7		1					1
384 - Session Fixation	A7	1	1					1
620 - Unverified Password Change	A7	1						1

Table 11 (continued)

CWE	Top Ten	SMPT	EMPT	DAST		SAST		Total
				ZAP	DA-2	Sonar	SA-2	
346 - Origin Validation Error	A7						2	2
613 - Insufficient Session Expiration	A5	1	1		1			2
521 - Weak Password Requirements	A7	10	7					10
A08 Software and Data Integrity Failures								
829 - Inclusion of Functionality from Untrusted Control Sphere	A8	1						1
502 - Deserialization of Untrusted Data	A8						10	10
A09 Security Logging and Monitoring Failures								
532 - Insertion of Sensitive Information into Log File	A9	1	1					1
778 - Insufficient Logging	A9	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	NA					13	1	14
404 - Improper Resource Shutdown or Release	NA						39	39

Table 12 Low Severity Vulnerability CWEs (continued)

CWE	Top Ten	SMPT	EMPT	DAST		SAST		Total
				ZAP	DA-2	Sonar	SA-2	
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)								

Table 12 (continued)

CWE	Top Ten	SMPT	EMPT	DAST		SAST		Total
				ZAP	DA-2	Sonar	SA-2	
Not Mapped to OWASP Top Ten								
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

Acknowledgements We appreciate the feedback provided by reviewers for this paper. We thank Jiaming Jiang for her support as teaching assistant for the security class. We are grateful to the I/T staff at the university for their assistance in ensuring that we had sufficient computing power running the course. We also thank the students in the software security class. Finally, we thank all the members of the Realsearch research group for their valuable feedback through this project.

This material is based upon work supported by the National Science Foundation under Grant No. 1909516. 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.

References

Ackerman E (2019) Upgrade to superhuman reflexes without feeling like a robot. IEEE Spectr. https://spectrum.ieee.org/enabling-superhuman-reflexes-without-feeling-like-a-robot

Alomar N, Wijesekera P, Qiu E, Egelman S (2020) "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), pp 319–339

Amankwah R, Chen J, Kudjo PK, Towey D (2020) An empirical comparison of commercial and open-source web vulnerability scanners. Softw - Pract Exp 50(9):1842–1857

Anderson T (2020) Linux in 2020: 27.8 million lines of code in the kernel, 1.3 million in systemd. The Register URL https://www.theregister.com/2020/01/06/linux_2020_kernel_systemd_code/. Accessed 21 Dec 2021

Antunes N, Vieira M (2009) 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. IEEE, pp 301–306

Antunes N, Vieira M (2010) Benchmarking vulnerability detection tools for web services. In: 2010 IEEE International Conference on Web Services,IEEE, pp. 203–210

Austin A, Holmgreen C, Williams L (2013) A comparison of the efficiency and effectiveness of vulnerability discovery techniques. Inf Softw Technol 55(7):1279–1288

Austin A, Williams L (2011) One technique is not enough: A comparison of vulnerability discovery techniques. In: 2011 International Symposium on Empirical Software Engineering and Measurement, IEEE, pp. 97–106

Bannister A (2021) Healthcare provider texas ent alerts 535,000 patients to data breach. The Daily Swig. [Online; Publication Date 20 Dec 2021; Accessed 21 Dec 2021]

Bartlett MS (1937) Properties of sufficiency and statistical tests. Proc R Soc A: Math Phys Eng Sci 160(901):268–282

Bau J, Wang F, Bursztein E, Mutchler P, Mitchell JC (2012) Vulnerability factors in new web applications: Audit tools, developer selection & languages. Tech. rep., Stanford, https://seclab.stanford.edu/websec/scannerPaper.pdf. Accessed 21 Dec, 2021

Campbell GA (2020) What is 'taint analysis' and why do i care?, https://blog.sonarsource.com/what-is-taint-analysis

Cass S (2021) Top programming languages 2021. IEEE Spectr, https://spectrum.ieee.org/top-programming-languages-2021. Accessed 21 Dec 2021

Cass S, Kulkarni P, Guizzo E (2021) Interactive: Top Programming Languages 2021. IEEE Spectrum, https://spectrum.ieee.org/top-programming-languages/. Accessed 20 Apr 2022

Chaim ML, Santos DS, Cruzes DS (2018) What do we know about buffer overflow detection?: A survey on techniques to detect a persistent vulnerability. International Journal of Systems and Software Security and Protection (IJSSSP) 9(3):1–33

Cicchetti DV, Feinstein AR (1990) High agreement but low kappa: Ii. resolving the paradoxes. J Clin Epidemiol 43(6):551–558

Cohen J (1960) A coefficient of agreement for nominal scales. Educ Psychol Meas 20(1):37–46

Condon C, Miller H (2021) Maryland health department says there's no evidence of data lost after cyberattack; website is back online. Baltimore Sun, https://www.baltimoresun.com/health/bs-hs-mdh-website-down-20211206-o2ky2sn5znb3pdwtnu2a7m5g6q-story.html. Accessed 21 Dec 2021

Cook TD, Campbell DT (1979) Quasi-experimentation: Design and analysis issues for field settings. Rand McNally College Publishing, Chicago

Corbin J, Strauss A (2008) Basics of qualitative research: Techniques and procedures for developing grounded theory, 3rd edn. SAGE Publications Inc., California

Cowan C, Wagle F, Pu C, Beattie S, Walpole J (2000) Buffer overflows: Attacks and defenses for the vulnerability of the decade. In: Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00, IEEE, vol. 2, pp. 119–129

Cruzes DS, Felderer M, Oyetoyan TD, Gander M, Pekaric I (2017) How is security testing done in agile teams? a cross-case analysis of four software teams. In: International Conference on Agile Software Development, Springer, Cham, pp. 201–216

Dambra S, Bilge L, Balzarotti D (2020) Sok: Cyber insurance–technical challenges and a system security roadmap. In: 2020 IEEE Symposium on Security and Privacy (SP), IEEE, pp. 1367–1383

Davis FD (1989) Perceived usefulness, perceived ease of use, and user acceptance of information technology. MIS Quarterly 13(3):319–340

Delaitre AM, Stivalet BC, Black PE, Okun V, Cohen TS, Ribeiro A (2018) Sate v report: Ten years of static analysis tool expositions. NIST SP 500-326, National Institute of Standards and Technology (NIST), https://doi.org/10.6028/NIST.SP.500-326. Accessed 20 Jul 2021

Desjardins J (2017) Here's how many millions of lines of code it takes to run different software. Business Insider, https://www.businessinsider.com/how-many-lines-of-code-it-takes-to-run-different-software-2017-2. Accessed 21 Dec 2021

Doupé A, Cova M, Vigna G (2010) 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, Springer, pp. 111–131

Elder SE, Zahan N, Kozarev V, Shu R, Menzies T, Williams L (2021) 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), IEEE, pp. 95–104

Epic Systems Corporation (2020) From healthcare to mapping the milky way: 5 things you didn't know about epic's tech, https://www.epic.com/epic/post/healthcare-mapping-milky-way-5-things-didnt-know-epics-tech. Accessed 07 Dec 2021

Executive Order 14028 (2021) Executive order on improving the nation's cybersecurity. Exec. Order No. 14028, 86 FR 26633, https://www.federalregister.gov/d/2021-10460

Feinstein AR, Cicchetti DV (1990) High agreement but low kappa: I. the problems of two paradoxes. J Clin Epidemiol 43(6):543–549

Feldt R, Magazinius A (2010) Validity threats in empirical software engineering research-an initial survey. In: Seke, pp. 374–379

Feng GC (2013) Factors affecting intercoder reliability: A monte carlo experiment. Qual Quant 47(5):2959–2982

Fielding RT, Reschke J (2014) Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC Editor https://doi.org/10.7231/RFC7231, https://rfc-editor.org/rfc/rfc7231.txt. Accessed 21 Dec 2021

Finifter M, Akhawe D, Wagner D (2013) An empirical study of vulnerability rewards programs. In: 22nd USENIX Security Symposium (USENIX Security 13), USENIX, Washington, D.C., pp. 273–288

Fonseca J, Vieira M, Madeira H (2007) Testing and comparing web vulnerability scanning tools for sql injection and xss attacks. In: 13th Pacific Rim international symposium on dependable computing (PRDC 2007), IEEE, pp. 365–372

Games PA, Howell JF (1976) Pairwise multiple comparison procedures with unequal n's and/or variances: a monte carlo study. J Educ Stat 1(2):113–125

Github (2021) The 2021 State of the Octoverse, https://octoverse.github.com/. Accessed 20 Apr 2022

Gonçales L, Farias K, da Silva BC (2021) Measuring the cognitive load of software developers: An extended systematic mapping study. Inf Softw Technol 106563

Hafiz M, Fang M (2016) Game of detections: how are security vulnerabilities discovered in the wild? Empir Softw Eng 21(5):1920–1959

Imtiaz N, Rahman A, Farhana E, Williams L (2019) Challenges with responding to static analysis tool alerts. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), IEEE, pp. 245–249

ISO/IEC/IEEE (2013) 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)

Itkonen J, Mäntylä MV (2014) Are test cases needed? replicated comparison between exploratory and test-case-based software testing. Empir Softw Eng 19(2):303–342

Itkonen J, Mäntylä MV, Lassenius C (2013) The role of the tester's knowledge in exploratory software testing. IEEE Trans Softw Eng 39(5):707–724

Johnson B, Song Y, Murphy-Hill E, Bowdidge R (2013) Why don't software developers use static analysis tools to find bugs? In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, pp. 672–681

Joint Task Force Transformation Initiative (2013) Security and privacy controls for federal information systems and organizations. NIST SP 800-53, National Institute of Standards and Technology (NIST), http://dx.doi.org/10.6028/NIST.SP.800-53r4. Accessed 20 Jul 2021

Kirk R (2013) Experimental design: Procedures for the behavioral sciences, 4th edn. Sage Publications, Thousand Oaks

Kitchenham B, Madeyski L, Budgen D, Keung J, Brereton P, Charters S, Gibbs S, Pohthong A (2017) Robust statistical methods for empirical software engineering. Empir Softw Eng 22(2):579–630

Kitchenham BA, Budgen D, Brereton P (2015) Evidence-based software engineering and systematic reviews, vol 4. CRC press, Boca Raton

Klees G, Ruef A, Cooper B, Wei S, Hicks M (2018) Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 2123–2138

Liu M, Zhang B, Chen W, Zhang X (2019) A survey of exploitation and detection methods of xss vulnerabilities. IEEE Access 7:182004–182016

Lombard M, Snyder-Duch J, Bracken CC (2002) Content analysis in mass communication: Assessment and reporting of intercoder reliability. Hum Commun Res 28(4):587–604

Lung J, Aranda J, Easterbrook S, Wilson G (2008) On the difficulty of replicating human subjects studies in software engineering. In: 2008 ACM/IEEE 30th International Conference on Software Engineering, pp. 191–200

Mallet F (2016) Sonaranalyzer for java: Tricky bugs are running scared. https://blog.sonarsource.com/sonaranalyzer-for-java-tricky-bugs-are-running-scared, Accessed 05 Dec 2021

McGraw G (2006) Software security: building security in. Addison-Wesley Professional, Boston

MITRE (2016) Common vulnerabilities and exposures (cve) numbering authority (cna) rules. https://cve.mitre.org/cve/cna/CNA_Rules_v1.1.pdf, Accessed 24 July 2021

MITRE (2021) Cve → cwe mapping guidance. In: (MITRE 2021b), https://cwe.mitre.org/documents/cwe_usage/guidance.html. Accessed 24 Jul 2021

MITRE (2021) Cwe common weakness enumeration (website), https://cwe.mitre.org/. Accessed 20 Jul 2021

MITRE (2021) Cwe view: Weaknesses in owasp top ten (2021). In: (MITRE 2021b), https://cwe.mitre.org/data/definitions/1344.html. Accessed 09 Dec 2021

MITRE (2022) Cwe 1003 - cwe view: Weaknesses for simplified mapping of published vulnerabilities

Morrison P, Moye D, Pandita R, Williams L (2018) Mapping the field of software life cycle security metrics. Inf Softw Technol 102:146–159

Mozilla (2021) Http messages, https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages. Accessed 21 Dec 2021

Nagarakatte S, Zhao J, Martin MiloMK, Zdancewic S (2009) Softbound: Highly compatible and complete spatial memory safety for c. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp 245–258

NVD (2021) Cwe over time. In: (NVD 2021b), https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cwe-over-time. Accessed 05 Dec 2021

NVD (2021) National vulnerability database (website). National Institute of Standards and Technology (NIST), https://nvd.nist.gov/. Accessed 01 Nov 2021

NVD (2021) Nvd - general faqs. National Institute of Standards and Technology (NIST). In: (NVD 2021b) https://nvd.nist.gov/general/FAQ-Sections/General-FAQs. Accessed 04 Apr 2022

NVD (2021) Vulnerabilities. In: (NVD 2021b), https://nvd.nist.gov/vuln. Accessed 01 Nov 2021

Okun V, Delaitre A, Black PE (2010) The second static analysis tool exposition (sate) 2009. NIST SP 500-287, National Institute of Standards and Technology (NIST), https://dx.doi.org/10.6028/NIST.SP.500-287. Accessed 20 Jul 2021

Okun V, Delaitre A, Black PE (2011) Report on the third static analysis tool exposition (sate 2010). NIST SP 500-283, National Institute of Standards and Technology (NIST), https://dx.doi.org/10.6028/NIST.SP.500-283. Accessed 20 Jul 2021

Okun V, Delaitre A, Black PE (2013) Report on the static analysis tool exposition (sate) iv. NIST SP 500-297, National Institute of Standards and Technology (NIST), https://dx.doi.org/10.6028/NIST.SP.500-297. Accessed 20 Jul 2021

Okun V, Gaucher R, Black PE (2009) Static analysis tool exposition (sate) 2008. NIST SP 500-279, National Institute of Standards and Technology (NIST), https://dx.doi.org/10.6028/NIST.SP.500-279. Accessed 20 Jul 2021

Open Web Application Security Project (OWASP) Foundation (2013) Owasp top ten - 2010, https://owasp.org/www-pdf-archive/OWASP_Top_10_-_2010.pdf. Accessed 05 Dec 2021

Open Web Application Security Project (OWASP) Foundation (2013) Owasp top ten - 2013, https://owasp.org/www-pdf-archive/OWASP_Top_10_-_2013.pdf. Accessed 05 Dec 2021
Open Web Application Security Project (OWASP) Foundation (2017) Owasp top ten - 2017, https://owasp.org/www-project-top-ten/2017/. Accessed 05 Dec 2021
Open Web Application Security Project (OWASP) Foundation (2021) Owasp top ten - 2021, https://owasp.org/Top10/. Accessed 05 Dec 2021
Open Web Application Security Project (OWASP) Foundation (2021) The owasp top ten application security risks project, https://owasp.org/www-project-top-ten/. Accessed 09 Dec 2021
Open Web Application Security Project (OWASP) Foundation (2021) Owasp zap, https://www.zaproxy.org/. Accessed: 21-Dec-2021
OpenMRS (2020) Openmrs developer manual, http://devmanual.openmrs.org/en/. Accessed 24 Jul 2021
OpenMRSAtlas (2021) Openmrs atlas, https://atlas.openmrs.org/. Accessed 24 Jul 2021
OWASP ZAP Dev Team (2021) Getting started - features - alerts. In: (Team OZD 2021), https://www.zaproxy.org/docs/desktop/start/features/alerts/. Accessed 06 Dec 2021
OWASP ZAP Dev Team (2021) Getting started - features - spider. In: (Team OZD 2021), https://www.zaproxy.org/docs/desktop/start/features/spider/. Accessed 20 Jul 2021
Pfahl D, Yin H, Mäntylä MV, Münch J (2014) 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, ACM, p. 5
Purkayastha S, Goyal S, Phillips T, Wu H, Haakenson B, Zou X (2020) Continuous security through integration testing in an electronic health records system. In: 2020 International Conference on Software Security and Assurance (ICSSA), IEEE, pp. 26–31
Radio New Zealand (RNZ) (2021) Health ministry announces $75m to plug cybersecurity gaps, https://www.rnz.co.nz/news/national/458331/health-ministry-announces-75m-to-plug-cybersecurity-gaps. Accessed 21 Dec 2021
Rahman AAU, Helms E, Williams L, Parnin C (2015) Synthesizing continuous deployment practices used in software development. In: 2015 Agile Conference, IEEE, pp. 1–10
Ralph P, Tempero E (2018) Construct validity in software engineering research and software metrics. In: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018, pp. 13–23
Razali NM, Wah YB et al (2011) Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. J Stat Modelling Anal 2(1):21–33
Scandariato R, Walden J, Joosen W (2013) Static analysis versus penetration testing: A controlled experiment. In: 2013 IEEE 24th international symposium on software reliability engineering (ISSRE), IEEE, pp. 451–460
Scanlon T (2018) 10 types of application security testing tools: When and how to use them. Blog, Software Engineering Institute, Carnegie Mellon University, https://insights.sei.cmu.edu/blog/10-types-of-application-security-testing-tools-when-and-how-to-use-them. Accessed 20 Jul 2021
Smith B, Williams L (2012) On the effective use of security test patterns. In: 2012 IEEE Sixth International Conference on Software Security and Reliability, IEEE, pp. 108–117
Smith B, Williams LA (2011) Systematizing security test planning using functional requirements phrases. Tech. Rep. TR-2011-5, North Carolina State University. Dept. of Computer Science
Smith J, Do LNQ, Murphy-Hill E (2020) 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), USENIX, pp. 221–238
Smith J, Johnson B, Murphy-Hill E, Chu B, Lipford HR (2015) Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, pp. 248–259
SonarSource (2019) Sonarqube documentation: Security-related rules, https://docs.sonarqube.org/8.2/user-guide/security-rules/. Accessed 06 Dec 2021
StackOverflow (2021) 2021 Developer Survey, https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies. Accessed: 07 Dec 2021
Team OZD (ed.) (2021) The owasp zed attack proxy (zap) desktop user guide
Tøndel IA, Jaatun MG, Cruzes DS, Williams L (2019) Collaborative security risk estimation in agile software development. Information & Computer Security 27(4)
U.S. Cybersecurity and Infrastructure Security Agency (CISA) (2021) Provide medical care is in critical condition: Analysis and stakeholder decision support to minimize further harm, https://www.cisa.gov/sites/default/files/publications/Insights_MedicalCare_FINAL-v2_0.pdf. Accessed 21 Dec 2021

US Dept of Veterans Affairs, Office of Information and Technology, Enterprise Program Management Office (2021) VA Monograph, https://www.va.gov/vdl/documents/Monograph/Monograph/VistA_Monograph_0421_REDACTED.pdf. Accessed 07 Dec 2021

van der Stock A, Cuthbert D, Manico J, Grossman JC, Burnett M (2019) Application security verification standard. Rev. 4.0.1, Open Web Application Security Project (OWASP), https://github.com/OWASP/ASVS/tree/v4.0.1/4.0. Accessed 20 Jul 2021

Votipka D, Stevens R, Redmiles E, Hu J, Mazurek M (2018) Hackers vs. testers: A comparison of software vulnerability discovery processes. In: 2018 IEEE Symposium on Security and Privacy (SP), IEEE, pp. 374–391

Wilcox RR, Keselman HJ (2003) Modern robust data analysis methods: measures of central tendency. Psychol Methods 8(3):254

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in software engineering. Springer Science & Business Media, New York