Efficiency and Effectiveness of Web Application Vulnerability Detection Approaches: A Review

BING ZHANG, School of Information Science and Engineering, Yanshan University, China
JINGYUE LI, Norwegian University of Science and Technology, Norway
JIADONG REN and GUOYAN HUANG, School of Information Science and Engineering, Yanshan University, China

Most existing surveys and reviews on web application vulnerability detection (WAVD) approaches focus on comparing and summarizing the approaches' technical details. Although some studies have analyzed the efficiency and effectiveness of specific methods, there is a lack of a comprehensive and systematic analysis of the efficiency and effectiveness of various WAVD approaches. We conducted a systematic literature review (SLR) of WAVD approaches and analyzed their efficiency and effectiveness. We identified 105 primary studies out of 775 WAVD articles published between January 2008 and June 2019. Our study identified 10 categories of artifacts analyzed by the WAVD approaches and 8 categories of WAVD meta-approaches for analyzing the artifacts. Our study's results also summarized and compared the effectiveness and efficiency of different WAVD approaches on detecting specific categories of web application vulnerabilities and which web applications and test suites are used to evaluate the WAVD approaches. To our knowledge, this is the first SLR that focuses on summarizing the effectiveness and efficiencies of WAVD approaches. Our study results can help security engineers choose and compare WAVD tools and help researchers identify research gaps.

CCS Concepts: • **Security and privacy** → **Software and application security**; • **General Terms** → Web application security;

Additional Key Words and Phrases: WAVD approaches, efficiency and effectiveness, vulnerability test suites

ACM Reference format:
Bing Zhang, Jingyue Li, Jiadong Ren, and Guoyan Huang. 2021. Efficiency and Effectiveness of Web Application Vulnerability Detection Approaches: A Review. *ACM Comput. Surv.* 54, 9, Article 190 (October 2021), 35 pages.
https://doi.org/10.1145/3474553

This work is supported by the National Natural Science Foundation of China under Grant Nos. 61802332, 61807028, and 61772449, the Natural Science Foundation of Hebei Province P. R. China under Grant No. F2019203120, and the doctoral Foundation Program of Yanshan University under Grant No. BL18012.

Authors' addresses: B. Zhang, J. Ren, and G. Huang, School of Information Science and Engineering, Yanshan University, 438 W Hebei Avenue, Qin Huangdao, Hebei Province, China, 066004; emails: bingzhang@ysu.edu.cn, jdren@ysu.edu.cn, hgy@ysu.edu.cn; J. Li (corresponding author), Department of Computer Science, Faculty of Information Technology and Electrical Engineering, Norwegian University of Science and Technology, Gløshaugen, Trondheim, Norway, 7030; email: jingyue.li@ntnu.no.

0360-0300/2021/10-ART190 $15.00
https://doi.org/10.1145/3474553

1 INTRODUCTION

According to OWASP 2017 [1], vulnerabilities in web applications can be categorized into injection, sensitive data exposure, broken authentication, broken access control, XML external entities, security misconfiguration, and XSS. Web application vulnerabilities have also been classified into three high-level categories: **input validation (IPV)** vulnerability, **session management (SM)** vulnerability, and **application logic (AL)** vulnerability [3, 4, 37]. To detect vulnerabilities, many approaches, e.g., static code analysis [2, 11], taint analysis [12], white box [13], machine learning approaches [14], fuzz testing [10, 17], penetration testing [18], and dynamic monitoring [19, 24] have been proposed.

A few studies have surveyed or reviewed the existing WAVD approaches from different viewpoints. Approaches proposed in References [27–35, 41] summarized and compared different methods to identify and mitigate specific vulnerabilities, e.g., SQLi or XSS. References [4, 36–38, 40, 42, 104] classified and compared other WAVD or mitigation approaches focusing on the ideas of the approaches. Although the information in the existing survey or literature reviews provided an overview of the WAVD approaches from different perspectives, none of the studies were dedicated to summarizing empirical evaluations of the approaches' efficiency and effectiveness. To perform security analysis, an overview of the empirical evaluation of different WAVD approaches can facilitate security engineers in choosing optimal methods. Based on a systematic summary of the strengths and weaknesses of different approaches' effectiveness and efficiency, researchers can identify research gaps and improve the weaknesses in the approaches.

To summarize WAVD approaches and their empirical evaluation results, we performed a systematic literature review on articles published from January 2008 to June 2019. Our study attempted to answer three research questions:

- RQ1: What kinds of artifacts have been analyzed by the current WAVD approaches and how were the artifacts analyzed?
- RQ2: How well were the artifacts analyzed to detect web application vulnerabilities?
- RQ3: Which web applications and test suites have been used to provide empirical evaluation results?

We first identified 775 articles through keywords searching in prestigious scientific databases. We then filtered out articles that were duplicated, not accessible, or did not contain the detailed information we needed. After filtering, we identified 105 primary studies. We combined thematic analysis and simple statistical analysis to answer our research questions. The results of our study show the following:

- The artifacts analyzed by the WAVD approaches can be divided into 10 categories, such as models derived from source code, complexity or size properties extracted from source code, patterns or rules derived from source code, and behavior models or constraints derived from application execution. The artifacts can be further abstracted into five themes, namely, model, property, code element, **application entry point (AEP)**, constraint or pattern, and fingerprint.
- Based on analysis strategy performed on the artifacts, the WAVD approaches presented in the primary studies can be classified into eight categories of meta-approaches, such as matching fingerprints using elements extracted from models, matching fingerprints with elements extracted from code, verifying constraints or patterns using models, classifying using code properties extracted from code, and generating attacks from a model.
- Seventy-eight out of the 105 primary studies focused on detecting injection vulnerabilities. Only 19 approaches showed less than 10% false-positive rates and 10% false-negative rates

or similar results detecting injection vulnerabilities. Other WAVD approaches' effectiveness is either not evaluated adequately or cannot give satisfactory evaluation results.

- Researchers often use web applications and test suites to compare and evaluate their WAVD approaches. However, only 21 out of the 105 primary studies presented detailed information on the vulnerabilities in the web application evaluated. Most other studies reported their evaluation results without disclosing the real numbers and types of vulnerabilities in the web application or test suites that their evaluations use.

The rest of this article is organized as follows: Section 2 presents a brief overview of web application vulnerabilities. Section 3 lists existing literature reviews and surveys on this topic. Section 4 explains our research design and implementation. Section 5 presents our literature review results. Section 6 discusses the results, and Section 7 concludes and gives our future work.

2 WEB APPLICATION VULNERABILITIES

Web application vulnerabilities can be classified into three high-level categories (i.e., **Input validation (IPV)** vulnerability, **Session management (SM)** vulnerability, and **application logic (AL)** vulnerability) in References [4, 37] or more detailed categories, such as those in OWASP top 10 [1].

IPV vulnerability refers to the absence of sanitization or insufficient validation of input supplied by a user through the user interface, such as input fields, of a web application. Attackers can exploit vulnerabilities by injecting crafted malicious commands or strings that violate the syntactic structure of the **operating system (OS)** command or SQL or XML query. Many attacks [6, 7, 60, 81] exploit IPV vulnerabilities, such as SQL injection, XSS, XML injection, LDAP injection, OS command injection, **remote code execution (RCE)**, and **local or remote file inclusion (LFI/RFI)**.

SM vulnerability is related to defects in the generation and processing of session tokens (i.e., session ID), which is critical to maintaining the relationship between the identification of end-users of applications and mapping subsequent requests of applications (i.e., maintenance status). Once the session management vulnerability is exploited, an attacker can compromise the session of a valid user and perform illegal actions. Typical exploitations of session management vulnerabilities include session fixation [97], session sniffing [97], and **cross-site request forgery (CSRF)** [120].

Because of improper authentication and authorization of web users, AL vulnerability allows attackers to access confidential web pages and perform unauthorized operations in the application. Currently, the most popular application logic vulnerabilities include parameter manipulation [66], weak access control [15, 98], workflow bypass [62], and workflow violation [129].

When identifying security risks of a system, in addition to threat modeling approaches, analysts can apply WAVD approaches and tools to test the application or to detect weaknesses in the source and executable code and the configurations.

3 EXISTING SURVEYS AND LITERATURE REVIEWS OF WAVD APPROACHES

Several surveys and literature reviews, as shown in Table 1, have summarized existing WAVD approaches.

- **Studies focus on surveying and classifying WAVD methods for detecting a specific type of vulnerability.** Jyotiyana et al. [27] studied high-level strategies to defend against clickjacking attacks but did not focus on specific tools and approaches to identify clickjacking vulnerabilities in web applications. Hydara et al. [29] conducted a systematic literature review on XSS vulnerability detection approaches and classified the approaches to static analysis, dynamic analysis, hybridization, secure programming, and modeling.

Table 1. Summary of the Related Surveys or Literature Reviews

Research articles	Publication Year	VL Type covered			Classified WAVD	Surveyed Testbed	VLs Covered
		IPV	SM	AL			
Studies focusing on a specific type of vulnerability							
Jyotiyana et al. [27]	2018		√				Clickjacking
Hydara et al. [29]	2015	√			√		XSS
Gupta et al. [32]	2014	√			√		SQLi, XSS
Johari et al. [33]	2012	√					SQLi, XSS
Scholte et al. [41]	2012	√					IPV VLs
Calzavara et al. [34]	2017		√				SM VLs
Visaggio et al. [35]	2010		√			√	SM VLs
Studies focusing on multiple types of vulnerability							
Li et al. [4]	2014	√	√	√	√		Multiple VLs
Deepa et al. [37]	2016	√	√	√	√	√	Multiple VLs
Chang et al. [22]	2013				√		Multiple VLs
Gupta et al. [36]	2017	√	√	√			Top 10 VL in OWASP
Seng et al. [30]	2018	√	√	√	√	√	
Atashzar et al. [39]	2012	√	√	√		√	

IPV: Input validation, SM: Session Management, AL: Application Logic, VL: Vulnerability.

References [32, 33] surveyed techniques to detect IPV vulnerabilities, and Reference [32] classified the techniques into static analysis, dynamic analysis, and hybridization. Scholte et al. [41] performed an empirical analysis aiming at understanding how input validation flaws have evolved in the past decade without focusing on summarizing approaches to detect vulnerabilities that can be exploited by attacks. Calzavara et al. [34] surveyed the most common attacks against web sessions and corresponding mitigation approaches but covered very little about the tools and approaches to detect session-related vulnerabilities. Visaggio et al. [35] explored web application design flaws that could be exploited by SM attacks and the general strategy and approaches to defend against the attacks.

- **Studies focused on surveying and classifying WAVD methods for detecting multiple types of vulnerabilities.** The categorization in Reference [4] included static analysis, dynamic analysis, and hybrid analysis. Deepa et al. [37] focused on the detection and/or prevention of attacks targeting injection and logic vulnerabilities and focused on static and dynamic techniques. Chang et al. [22] outlined only two web-based malware detection methods, namely, virtual machine-based detection and signature-based detection. Gupta et al. [36] presented a comprehensive survey of emerging web application weaknesses and discussed mechanisms of avoidance, detection, and attack patterns for all critical web threats in OWASP 2013. However, Reference [36] focused on high-level principles to detect vulnerabilities without analyzing specific methods and tools. Seng et al. [30] surveyed web application security scanners and their qualities by summarizing the measurement metrics used to quantify scanner quality in various studies. Atashzar et al. [39] surveyed web application security aspects, including critical vulnerabilities, hacking tools, and approaches at a high level. References [30, 35, 37, 39] summarized web applications used as a testbed for evaluating WAVD approaches and tools.

Most of the existing surveys and interviews focus on summarizing the methods to detect vulnerabilities or methods to defend against attacks, and none of the surveys or reviews shown in Table 1 focused on summarizing and comparing the efficiency and effectiveness of the WAVD approaches.

Several studies, e.g., References [133, 134], showed that high false-positive rates (>10%) or high-performance overhead (≥5%) would discourage people from using specific WAVD approaches and tools. The motivation of this study was to investigate the efficiency and effectiveness of existing WAVD approaches to help practitioners choose the proper WAVD approach and tool and to help researchers identify research gaps.

4 RESEARCH DESIGN AND IMPLEMENTATION

In a mapping study [103], WAVD approaches were classified into static analysis, dynamic analysis, white box, black box, taint analysis, penetration testing, fuzz testing, concolic testing, symbolic execution, and model checking. References [4, 22, 29, 30, 32, 37] mostly classified existing WAVD approaches into three categories, namely, static, dynamic, and hybrid. To compare the efficiency and effectiveness of the WAVD approaches, the current classification may not provide enough granularity. For example, static analysis can focus on the client-side code or the server-side code. Static analysis can also target different kinds of vulnerabilities and different content and features of web applications. To more specifically compare the efficiency and effectiveness of WAVD approaches, we started by revisiting the existing WAVD method classification and made a finer-grained classification by investigating the detailed analysis processes (steps) of each approach, the artifacts (e.g., HTML tags, logic functions, variables in the source code, the HTTP request, GET/POST parameters, and sessions or cookies values), analyzed by the WAVD approaches, and the vulnerabilities the approaches focus on. Then, we compare the efficiency and effectiveness of each fine-grained category of WAVD approaches on different kinds of vulnerabilities. Finally, we identified the possible resources for evaluating the efficiency and effectiveness of the WAVD approaches. To make a complete and comprehensive summary and comparison of the efficiency and effectiveness of the WAVD approaches, we followed the systematic literature review guidelines in References [44–46]. We covered the peer-reviewed primary papers related to WAVD published between January 2008 and June 2019.

4.1 Searching and Filtering the Primary Studies

We started by searching the most popular scientific publication databases, namely, *IEEE Xplore, ACM Digital library, Elsevier Science Direct, SpringerLink, Web of Science,* using search strings and obtained 775 papers. The search strings are listed in Table 2. We filtered the search results manually by using the inclusion and exclusion criteria shown in Table 3 and obtained 72 papers. The results after filtering are shown in Table 4. After that, we manually performed a backward snowballing search by searching the reference list of the 72 papers and a forward snowballing search by reviewing all articles citing them using "Google scholar-search within citing articles." We performed a forward snowballing search, because the WAVD approaches may be updated or further evaluated by follow-up studies. After the snowballing search, 33 more articles were discovered. Finally, we included 105 studies as the primary papers for answering our research questions. A summary of the process of searching and filtering the papers is presented in Figure 1. The searching process and results of phase 1 to phase 3 in Figure 1 and the snowballing phase (i.e., the last phase of Figure 1) are logged in excel files and shared in the public link: https://figshare.com/s/c0d1a19db6d06a33d123 and the public doi: https://doi.org/10.6084/m9.figshare.13580048.v1.

4.2 Data Analysis and Synthesis

To answer RQ1, we followed the guidelines in References [47, 48] and applied thematic synthesis to classify artifacts the WAVD approaches focus on, and the WAVD approaches. We first extracted and coded the artifacts the WAVD approaches focus on. We then translated the artifacts into 10 categories. We created five high-order themes of the artifacts by exploring the relationship between

Table 2. Search String

Key words	Search string
web, website, web application web application vulnerability, web vulnerability, vulnerability,	(web **or** website **or** web application) **AND** (web application vulnerability **or** web vulnerability **or** vulnerability)

Table 3. Inclusion Criteria and Exclusion Criteria

Inclusion Criteria	Exclusion Criteria
1. paper related to WAVD; 2. papers published between 2008 January and 2019 June; 3. paper written in English; 4. paper come from peer-reviewed journals or Conferences.	1. Secondary studies; 2. Duplicated studies; 3. Paper with many versions (we choose only the latest one and excluded early version); 4. Papers we cannot find the full text or papers was withdrawn; 5. Papers are relevant to vulnerability detection but not about Web applications; 6. Papers we cannot extract the needed information to answer our research questions (e.g., analysis objects and analysis).

Table 4. The Analysis Results in Phase 3

	Type of Literature	Number	Percentage
1	Primary studies: Web application detection techniques (Inclusion Criteria 1, 2, 3, 4)	72	26.47%
2	Secondary studies et al. (Exclusion Criteria 1)	35	12.87%
3	Irrelevant, duplicated, or unavailable papers (Exclusion Criteria 2, 3, 4, 5)	95	34.93%
4	Paper without enough information we need (Exclusion Criteria 6)	70	25.73%

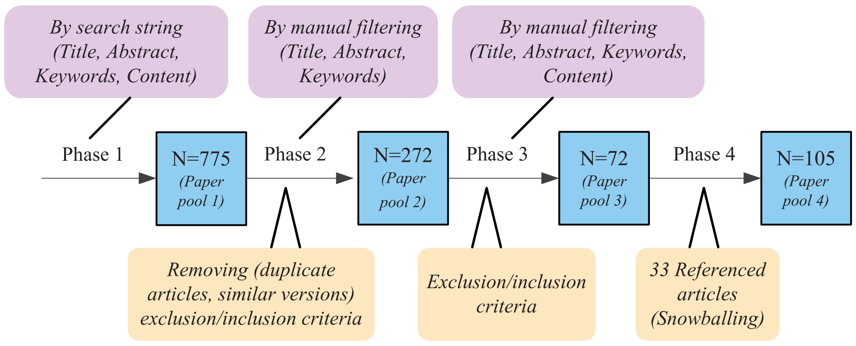

By search string (Title, Abstract, Keywords, Content)
By manual filtering (Title, Abstract, Keywords)
By manual filtering (Title, Abstract, Keywords, Content)
Phase 1
N=775 (Paper pool 1)
Phase 2
N=272 (Paper pool 2)
Phase 3
N=72 (Paper pool 3)
Phase 4
N=105 (Paper pool 4)
Removing (duplicate articles, similar versions) exclusion/inclusion criteria
Exclusion/inclusion criteria
33 Referenced articles (Snowballing)

Fig. 1. The selection process of primary papers.

the 10 artifact categories. To analyze the WAVD approaches, we first extracted and coded the data processing strategies of the WAVD approaches and identified four codes. By combing the high-order themes of the artifacts and the data processing strategy codes, we identified eight WAVD meta-approaches. We observed that the primary studies applied one meta-approach or combined several meta-approaches.

As RQ2 focuses on comparing the efficiency and effectiveness of the WAVD approaches, we first extracted efficiency and effectiveness metrics and results reported in the primary studies. Then, we summarized how well each category of WAVD meta-approaches detects particular categories of security vulnerabilities. Also, we analyzed studies that reported low false-positive results and low false-negative results to understand why those approaches provided good results.

To answer RQ3, we extracted the applications and test suits listed in the primary studies and manually tested whether the applications and test suites were accessible. We also summarized other information of the applications and test suites, such as the application code's programming language and vulnerabilities inserted.

5 RESULTS

5.1 Basic Information of the Primary Studies

More than half (64.76%) of the 105 primary papers are conference publications, and the others are journal publications. The numbers of papers published each year are shown in Figure 2 and illustrate that WAVD approach studies were a hot topic in the past 10 years.

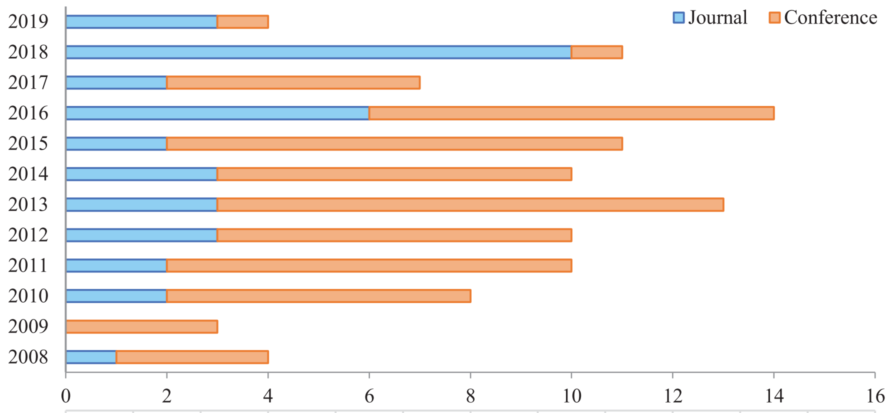
The programming languages analyzed by the WAVD approaches are summarized in Figure 3, which shows that PHP- and Java/JSP-based web applications were the main focuses of existing WAVD approaches. Only 1% of the study focused on web applications developed using Ruby on Rails. Another 1% of the study focused on the JavaScript code of the web application. 22% of primary studies shown in Figure 3 did not give information about the programming languages analyzed by the WAVD approaches.

5.2 Results of RQ1

To present the results of RQ1, we first describe the artifact categories and their high-order themes. After that, we explain the WAVD meta-approaches categories based on their detailed analysis process and artifacts analyzed.

5.2.1 The Classification of the Artifacts the WAVD Analyses. We find that different WAVD approaches analyzed different artifacts. Forty-seven out of the 105 studies focused on analyzing the source code or intermediate results derived from the source code. The artifacts they analyze can be classified as follows:

- **S1. Models derived from source code.** Some WAVD analyzed the control flow models (e.g., **Control Flow Graph (CFG), Call Graph (CG)**, and **Data Dependence Graph (DDG)**), data flow models (e.g., **Data flow (DF)** and **Information flow (IF)**), or syntax model (e.g., **Abstract syntax tree (AST)**) of the web application. For example, Shar et al. [76] built the CFG and DDG of the web application to identify potential XSS vulnerabilities.
- **S2. Complexity or size properties of the code.** Properties of the source code of web applications, e.g., cyclomatic complexity, lines of code (HTML, non-HTML such as JavaScript, CSS, and comments), number of functions, maximum nesting complexity, Halstead's volume, or total number of external calls, were used to train a prediction model to discover some vulnerabilities of the web application. For example, Catal et al. [14] proposed an approach for predicting vulnerability in web services by artificial neural networks trained on the attributes of software metric values received from a web form.
- **S3. Elements of source code of web application.** After extracting fingerprint from vulnerable code, the vulnerability scanners (e.g., Reference [8]) compare the fingerprint with code elements of the web application, such as file and function names. The tool SAWFIX [11] starts by obtaining the file names from the static portion of the inclusion to replace the



	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019
🟦 Journal	1	0	2	2	3	3	3	2	6	2	10	3
🟧 Conference	3	3	6	8	7	10	7	9	8	5	1	1

Fig. 2. Publications published during 2008–2019.

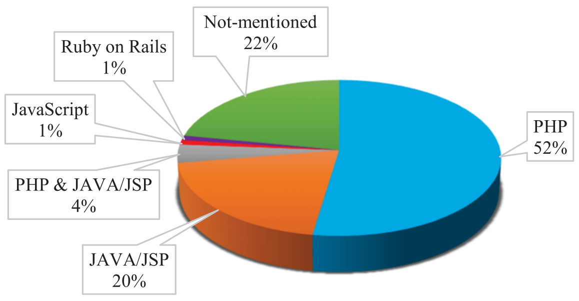


Fig. 3. The programming languages analyzed by the WAVD approaches.

dynamic process to detect session fixation vulnerabilities. Besides, the context information about configuration directives [50], context-sensitive alias, field names in a program [57], and different types of APIs [87] also belongs to code elements.

- **S4. Application entry points identified from code.** Some vulnerability scanners, such as Reference [16], crawl all web sites first to identify all possible application entry points (AEPs, such as inputs and SQL statements) and then send normal, malicious, or incorrect parameter values to the AEPs to detect vulnerabilities. Alkhalaf et al. [80] extracted client-side and server-side input validation functions, and Reference [153] collected SQL statements from database execution logs.
- **S5. Constraints or patterns derived from secure code.** Some WAVD approaches generate constraints or patterns from a secure web application or legitimate use of the web application and then identify vulnerabilities by checking whether the new or updated web applications or malicious inputs violate the constraints or patterns. For example, Trinh et al. [82] generated constraints from a secure JavaScript Program, and Jang et al. [83] formed the patterns of legitimate SQL queries first and then used the patterns to identify SQL injection attacks compared with the tools such as SQLIPA [135], CANDID [78], SQLProb [136], and SQLinjectionGen [137].

- **S6. Fingerprints derived from vulnerable code.** Some WAVD approaches, especially vulnerability scanners (e.g., Reference [85]), search file names, tokens, and function names, are known to be vulnerable. These known vulnerable pieces of code are called fingerprint data in Reference [85]. Le et al. [86] analyzed the token list of each file to identify potentially dangerous functions, and Shahriar et al. [87] applied information retrieval methods to search known method calls that are related to object injection vulnerabilities. For approaches (e.g., Reference [61]) using taint analysis, the sensitive sources, sinks, or sanitization points always come from analyzing known vulnerable code.

Fifty-five articles focused on analyzing the web application execution or intermediate data derived from running web applications. The artifacts they analyzed can be classified into four categories.

- **B1. Behavior models derived from application execution.** The web application behavior model, navigation graph, and navigation paths can be obtained by analyzing the execution trace of the web application or messages between the client and server. For example, Thilagam et al. [74] modeled parameters and session variables flowing between different web pages and the sequence of actions among web pages as an annotated finite state machine. Li et al. [148] analyzed logs of a web application and modeled the normal users' behavior using the **hidden Markov model (HMM)**. Li et al. [66] constructed a partial FSM over the expected input domain by collecting and analyzing the execution traces when users follow the web application's navigation paths.
- **B2. Elements of dynamically generated web page.** Some approaches focused on the script, iframe, CSS style, or image tags in the generated web pages to identify possible vulnerabilities inserted into the pages. For example, Gupta et al. [58] explored the HTTP response for extracting the script content and compared the content with possible malicious script functions to detect malicious XSS worms. Kavitha et al. [90] invented an approach to check the iframe of the code to detect the clickjacking vulnerability.
- **B3. Constraints or patterns derived from web application execution.** Some studies focused on analyzing the patterns generated when executing web applications. For example, Li et al. [62] identified invariants from the execution of the web application during its attack-free execution and then used invariants to detect vulnerabilities of the application at runtime.
- **B4. Constraints or patterns derived from HTTP traffic.** Some studies focused on analyzing the HTTP traffic (such as page input fields, form fields, login input, HTTP GET/POST parameters, HTTP cookies, HTTP user-agents, and referrer header values) between client and server. For example, References [16, 100] first extracted the **application entry point (AEP)** and then created a set of incorrect or malicious parameter values to attack these entry points and analyzed the results to detect vulnerabilities. Muthukumaran et al. [9] proposed a **user-data-policy (UDA)**-based approach to detect violations of access control by tracking HTTP traffic. Sunkari et al. [91] designed a regular expression engine that takes values of HTTP request/response parameters and built a set of application regular expression attribute validation rules to prevent input type validation vulnerabilities. Gupta et al. [59] extracted a set of axioms by monitoring the sequences of HTTP requests/responses and their corresponding session variables to detect workflow bypass and XSS vulnerabilities.

Three papers analyzed both static source code attributes and dynamic behavior attributes to detect vulnerabilities. Monga et al. [54] statically analyzed PHP bytecode and searched for dangerous code statements and monitored these statements by dynamic analysis. Scholte et al. [95] designed the **input parameter analysis system (IPAAS)**, which automatically extracted the parameters for a web application and learned types for each parameter. IPAAS then applied a combination

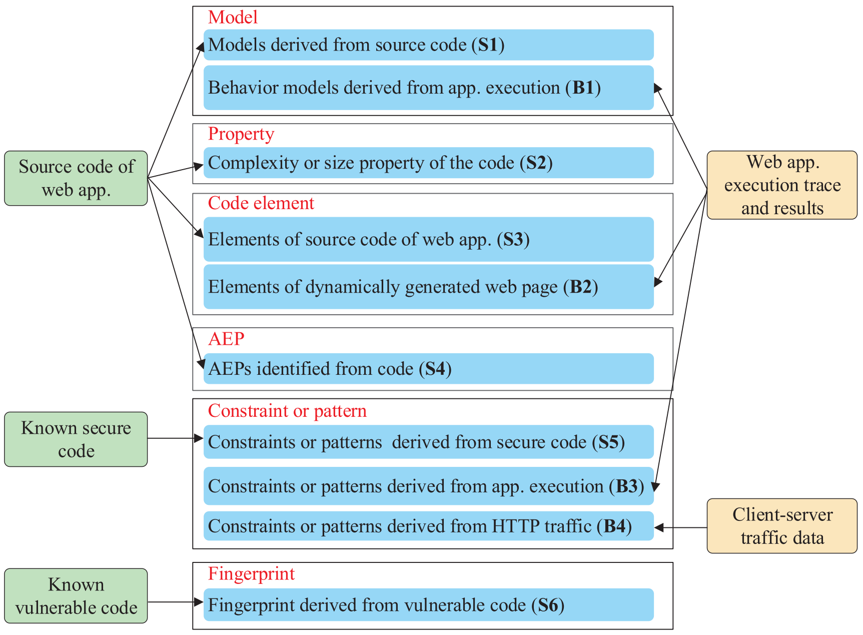

Model
Models derived from source code (S1)
Behavior models derived from app. execution (B1)
Property
Complexity or size property of the code (S2)
Code element
Elements of source code of web app. (S3)
Elements of dynamically generated web page (B2)
AEP
AEPs identified from code (S4)
Constraint or pattern
Constraints or patterns derived from secure code (S5)
Constraints or patterns derived from app. execution (B3)
Constraints or patterns derived from HTTP traffic (B4)
Fingerprint
Fingerprint derived from vulnerable code (S6)
Source code of web app.
Known secure code
Known vulnerable code
Web app. execution trace and results
Client-server traffic data

Fig. 4. The artifacts and their relationship.

of machine learning and static analysis to find parameters and application resources that were missed to prevent the exploitation of XSS and SQL injection vulnerabilities. Balzarotti et al. [122] combined static and dynamic analysis techniques to identify faulty sanitization procedures that an attacker can bypass.

The artifacts analyzed by the WAVD approaches and their relationship are shown in Figure 4. The artifacts can be summarized into five high-order themes: model, property, code element, AEP, constraint or pattern, and fingerprint.

5.2.2 The Classifications of WAVD Approaches. From the primary studies, we identified four analysis strategy codes: match, verify, classify, and generate attack. We then identified eight WAVD meta-approaches by combining the artifacts' high-order themes and the analysis strategy codes. The results are shown in Figure 5.

- **Match Fingerprint with Elements Extracted from a Model (MFM).** WAVD methods in this category [6, 7, 11, 12, 23, 52, 53, 55, 60, 61, 64, 69, 71, 76, 77, 80, 81, 86, 108, 142, 145, 147, 150] usually begin by deriving models, e.g., CFG, DDG, AST, browsing behavior models, navigation graphs, and navigation paths. The WAVD approaches then traverse the model to extract code elements to compare with known fingerprints. For example, Yan et al. [12] implemented a backward variable tracing algorithm to all trace variables along all paths in AST, CFG, CG for taint analysis, which determines whether sanitization functions have sanitized a variable before the value of the variable is used in sink functions. Dahse et al. [61] also built AST, CFG, and DF first and then performed backward-directed taint analysis. Shar et al. [76] built the CFG and DDG and then used taint-based analysis techniques and pattern matching to identify potential XSS vulnerabilities.
- **Match Fingerprint with elements extracted from Code (MFC).** This category of WAVD approaches [49, 50, 54, 58, 85, 87, 90, 96, 116, 122, 132, 136, 143, 149] usually matches and

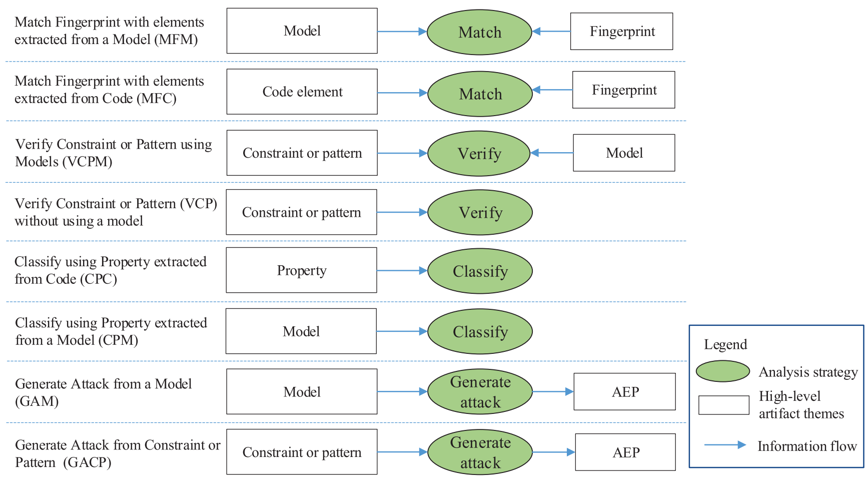

Match Fingerprint with elements extracted from a Model (MFM)
Model
Match
Fingerprint
Match Fingerprint with elements extracted from Code (MFC)
Code element
Match
Fingerprint
Verify Constraint or Pattern using Models (VCPM)
Constraint or pattern
Verify
Model
Verify Constraint or Pattern (VCP) without using a model
Constraint or pattern
Verify
Classify using Property extracted from Code (CPC)
Property
Classify
Classify using Property extracted from a Model (CPM)
Model
Classify
Generate Attack from a Model (GAM)
Model
Generate attack
AEP
Generate Attack from Constraint or Pattern (GACP)
Constraint or pattern
Generate attack
AEP
Legend
Analysis strategy
High-level artifact themes
Information flow

Fig. 5. The WAVD meta-approaches.

compares the source code or dynamic code of the web application with known legitimate or malicious code. For example, the idea of Reference [85] was to crawl a website to extract the site keys, such as file name, and then compare the keys with known vulnerable keys. Gupta et al. [49] first built a whitelist of legitimate scripts offline. The whitelist was used as a fingerprint to match scripts in the runtime **document object model (DOM)** tree to detect the injection of malicious scripts. Wu et al. [116] used structure matching to analyze server-side SQL commands rendered from user queries and to compare them with the general structure of benign SQL commands to detect malicious SQL queries.

- **Verify Constraint or Pattern Using Models (VCPM)**. The methods in this category [15, 53, 66, 68–70, 74, 78, 88, 95, 98, 140, 142, 148, 150, 153, 154] typically compare the expected behavior (i.e., constraints) with the information of possible behavior that can be extracted from models. For example, the tool in Reference [68] first asked developers to specify AEPs and role-based application states as a basis for automatically inferring the privileged pages. Then, the tool constructed a CFG that represents the possible HTML outputs and the sitemaps for different roles in web applications. After that, the tool infers privileged pages to access them directly to detect access-control vulnerabilities. Li et al. [148] analyzed the differences in the behavior patterns of attackers and normal users, which are expressed using the **hidden Markov model (HMM)**, to detect SQL injection attacks. Li et al. [66] first constructed a partial FSM by collecting and analyzing the execution traces of expected application execution. Then, they tested the application program in each state by constructing unexpected input vectors and evaluated the corresponding web response to detect logical vulnerabilities according to the difference between the expected FSM and the implemented FSM.
- **Verify Constraint or Pattern (VCP) without using models**. This category of WAVD methods [9, 23, 54, 55, 57, 59, 62, 67, 72, 75, 82, 83, 91, 94, 105, 109, 110, 115, 118, 119, 121, 126, 130, 151, 152] begins by identifying constraints, patterns, or policies related to a specific vulnerability from source code, execution trace, or user inputs and then checks or monitors violations of the constraints to detect vulnerabilities or attacks. Unlike VCPM, the VCP approaches do not generate and use models in the process of constraint and pattern verification. For example, Zheng et al. [57] analyzed and encoded PHP scripts into nonstring and

string constraints for detecting vulnerabilities related to remote code execution. Li et al. [62] checked invariants to detect vulnerabilities of an application at runtime. Halfond et al. [72] built constraints from trusted data and then used the constraints to detect SQLi attacks at runtime. Sunkari et al. [91] designed a regular expression engine to sanitize HTTP requests and response traffic to defend against SQLi and XSS attacks.

- **Classify Using Property Extracted from Code (CPC)**. Such approaches [14, 65, 111, 139, 144, 146] usually extract features of known vulnerable or nonvulnerable web applications as input for training classifiers and then use the classifiers to detect and predict vulnerabilities of other web applications. For example, Catal et al. [14] investigated and compared several classification algorithms to predict vulnerabilities in web services by using properties such as cyclomatic complexity, lines of code, number of functions, and maximum nesting complexity.
- **Classify Using Property Extracted from a Model (CPM)**. These methods [7, 56, 102, 107, 141, 152] build models first and then extract properties from models to perform classification. For example, Shar et al. [56] first extracted the input validation and sanitization properties based on CFG, PDG, and SDG and then applied supervised and semisupervised learning to build vulnerability predictors. Shar et al. [107] collected static and dynamic code properties, such as "*the number of nodes which invoke functions that return only numeric, mathematic, or dash characters,*" from CFG and DDG and used three different classifiers to build prediction models to predict SQLi and XSS vulnerabilities. Reference [141] classified the user queries by extracting features from known benign and malicious user queries.
- **Generate Attack from a Model (GAM)**. These methods [63, 64, 79, 84, 89, 92–94, 101, 106, 138, 155] usually use information in the models to generate or guide the generation of attacks to test the web application. For example, Avancini et al. [106] first built the CFG and then used symbolic execution to "*generate input values to make the execution take all the identified target branches*" to detect XSS vulnerabilities.
- **Generate Attack from Constraint or Pattern (GACP)**. These methods [15, 16, 18, 20, 21, 51, 74, 75, 98, 100, 110–114, 117, 120, 121, 125, 130, 131] generate attacks or penetration test cases from constraints or attack patterns to determine whether the web application violates the constraints or the attacks succeed. For example, Deepa et al. [75] invented a two-phase approach. The first phase is a training step, which crawled the web application and analyzed HTTP requests and responses to build the constraint sets. The second phase is the testing step, in which the system "*uses the learned set of constraints for generating the attack vectors*" to identify parameter tampering vulnerabilities and XQuery injection vulnerabilities. The scanner of Kumar et al. [114] first crawled the web application to identify all its pages and then runs a simulated attack on the pages. Although the attacks are not derived from abstract constraints, the attack payloads are created based on the historical SQLi attacks, which can be regarded as patterns. Awang et al. [18] proposed an automated framework to first generate test cases of a web application by using SQLi attack patterns and permutation algorithms and then analyze the test results to detect the SQLi vulnerability. References [15, 98] extracted access-control constraints from models built based on execution traces and then generated test cases to test constraint violations.

5.3 Results of RQ2

To answer RQ2, we collected ***time consumption/memory consumption (TC/MC)*** data presented in the evaluation and information about whether some steps of the method are ***automatic (Auto)*** to measure the efficiency of the WAVD approaches. For measuring effectiveness, in addition to popular metrics [156] listed in Table B.1 in Appendix B, such as ***false positive rate (FPR), false negative rate (FNR), true negative rate (TNR), true positive rate (TPR), precision***

(P), accuracy (Acc), Recall (R), F-measure/F1-score, code coverage (CC) *[124]*, ***detection rate (DR)*** *(i.e., number of attacks detected as attacks/number of attacks [131, 150])*, metrics such as *p-value* and *effect size* [20], ***area under the receiver operating characteristic curve (AUC)*** [14], and *fitness* [123, 131], are also presented in the evaluation of the WAVD approaches and are therefore summarized and compared.

Seventy-eight out of the 105 studies focus on injection vulnerabilities, probably because injection vulnerabilities are top listed in OWASP 2013 and OWASP 2017. The other primary studies focus on SM vulnerability, AL vulnerability, and vulnerabilities, e.g., side-channel [101, 102], which cannot be classified into IPV, SM, AL, or OWASP Top 10 categories. Fifteen out of the 105 primary studies focus on multiple types of vulnerabilities, and the others focus on only one type of vulnerability. As most primary studies focus on injection vulnerabilities, we summarized the efficiency and effectiveness of WAVD approaches into two subsections, namely, the injection vulnerability section and other vulnerability section.

5.3.1 The Efficiency and Effectiveness of Approaches Detecting Injection Vulnerabilities. The injection vulnerability detection approaches' effectiveness data are shown in Tables 5 and 6. More detailed data, including the artifacts the approaches analyze and the number of projects included in approach evaluations, are shown in Tables A.1 and A.2 in Appendix A.

For studies detecting injection vulnerabilities, if we consider those that reported fewer than 10% of both FPR and FNR, and the ones that show equivalent results using other metrics, e.g., precision (>90%), recall (>90%), and F-measure (>90%), we found 19 primary studies. These 19 studies are marked in bold in Tables 5 and 6. The summaries of these 19 approaches are as follows:

- **XSS.**
  - **MFM+VCP.** Reference [23] proposed context-sensitive taint analysis with pattern matching and reported 0% FRP and 0% FNR.
  - **MFC.** The Reference [143] approaches first identified all untrusted user inputs and sanitized them and then stored the sanitizer snapshot and used the snapshots to detect the injection of XSS worms achieved high precision and recall.
  - **MFC.** Reference [49] used a similar idea to the one in Reference [143] to detect DOM-based XSS vulnerability but analyzed elements in the DOM tree and achieved high precision and recall.
- **SQLi.**
  - **VCP.** Reference [72] applied positive taint analysis, which is based on identifying and tracking trusted data, and syntax-aware evaluation, which considers the context in which the trusted and untrusted data are used and reports 0% FPR and 0% FNR.
  - **MFC.** Reference [149] measured document similarity to identify SQLi and achieved high precision and recall.
  - **GACP.** Reference [84] generated SQLi test cases by combining various patterns, e.g., test statement patterns and control patterns, and injection points of the system, and achieved 0% FPR and 0% FNR.
  - **GACP+GAM.** Reference [92] first summarized existing SQLi payload patterns using **SQLIV Penetration Test Finite State Machine (SPT-FSM)** models and then generated SQLi test cases from these models.
  - **VCPM.** Reference [153] extracted patterns of benign and malicious SQL commands from historical data and then used the patterns to identify malicious SQL commands. The evaluations evaluations of Reference [153] reported low FPRs and low FNRs.
- **SQLi and XSS.** Many studies try to detect SQLi and XSS attacks or vulnerabilities using one approach.

Table 5. The Effectiveness of WAVD Methods on Input Validation Vulnerabilities (Part I)

ID	Vulnerability	Type	FPR	FNR	TPR	Precision	Other metrics	Ref.
1	XSS	MFM				33.3%–60%	Cac.	[12]
2	XSS	MFM	0.00%					[77]
3	XSS	MFM				26.20%		[76]
4	**XSS**	**MFM+VCP**	**0.00%**	**0.00%**				**[23]**
5	XSS	MFM+VCP						[55]
6	XSS	MFC		6.7%–17.9%	82.1%–93.3%	88.5%–97%	Cac.	[58]
7	**XSS**	**MFC**	**0.5%–0.667%**	**0.8%–1.8%**	**98.2%–99.1%**	**94.1%–95.8%**	**F-measure>96.1%. Cac.**	**[143]**
8	XSS	CPC				69.2%–92%	Recall: 69.5%–92.6%. F-measure: 68.2%–92.6%. Accuracy: 69.5%–92.6%.	[146]
9	XSS	VCP	0.00%					[59]
10	XSS	VCPM	10%–15%	0.00%				[154]
11	XSS	GACP					Reported Phi and Fitness graphs.	[131]
12	XSS	GACP+CPC						[111]
13	XSS	GAM						[63]
14	XSS	GAM						[106]
15	XSS	GAM+VCP						[94]
16	**DOM-based XSS**	**MFC**			**98.4%–99.2%**	**97.7%–99.2%**	**F-measure: >98%.**	**[49]**
17	**SQLi**	**VCP**	**0.00%**	**0.00%**				**[72]**
18	SQLi	MFM+VCPM						[69]
19	SQLi	MFC	20.50%					[116]
20	SQLi	MFC					Accuracy: 100%.	[132]
21	**SQLi**	**MFC**	**0.65%–1.17%**	**1.07%–3.36%**	**97.96%–98.93%**	**93.17%–97.09%**	**F1-score: 95.51%– 98.04%**	**[149]**
22	SQLi	MFC					Detection rate: 100%.	[96]
23	SQLi	MFC					Detection rate: 100%	[136]
24	SQLi	GACP	0.00%			100.00%	Cac.	[114]
25	SQLi	GACP					Identified 29 VLs.	[18]
26	SQLi	GACP					7 Scanner compared.	[112]
27	SQLi	GACP						[113]
28	SQLi	GACP					P-value <0.001 and effect size (d>1).	[20]
29	**SQLi**	**GACP**	**0.00%**	**0.00%**			**Detection rate: 100%.**	**[84]**
30	**SQLi**	**GACP+GAM**	**0.34%**				**Recall: 97.5%.**	**[92]**
31	SQLi	GAM					F-measure: 257.33.	[155]
32	SQLi	GAM	0.05%–0.08%		97.6%–100%			[93]
33	SQLi	VCPM	16.20%	5.75%				[148]
34	SQLi	VCP		0.00%				[115]
35	SQLi	VCP	0.00%					[83]
36	SQLi	VCPM	0.00%					[78]
37	**SQLi**	**VCPM**	**0%–0.6%**	**0.00%**	**100%**	**12.3%–100%**	**Cac.**	**[153]**
38	SQLi, XSS	MFM				33.3%–100%	Cac.	[86]
39	SQLi, XSS	MFM				SQLi: 35%; XSS: 37%		[81]
40	Second order SQLi, XSS	MFM		21.00%		79.00%		[60]
41	SQLi, XSS	MFM+VCPM				XSS: 60.5%–100%; SQLi: 62.5%–84%	Cac.	[53]
42	**SQLi, XSS**	**MFM+VCPM**	**0.00%**	**0.00%**				**[150]**
43	SQLi First order XSS Second order XSS	MFM+GAM	SQLi 0% First order XSS: 42% Second order XSS: 0%				Code coverage<50%.	[64]
44	SQLi, XSS	MFC						[122]
45	SQLi, XSS	MFC+VCP						[54]
46	SQLi, XSS	GACP	Low*	Low*				[100]
47	SQLi, XSS	GACP				SQLi: 87.5%; XSS: 100%		[117]
48	**SQLi, XSS**	**GAM**	**0.00%**	**0.00%**				**[79]**
49	**SQLi, XSS**	**GAM**	**4%–6.4%**	**0%–4.8%**				**[138]**
50	SQLi, XSS	CPC				22.1%–84.9%	Recall: 13.8%–85.1%. F1-score: 16.9%–84.8%.	[139]
51	SQLi, XSS	CPC	XSS: 6%–9%, SQLi: 11%–16%		XSS: 65%–78%; SQLi: 92%–93%	XSS: 78%–82%; SQLi: 90%–92%	Accuracy>=85%.	[65]
52	SQLi, XSS	CPM				Supervised predicator: 85%; Unsupervised predicator: 39%	Recall of supervised predicator: 90%. Recall of unsupervised predicator: 76%.	[107]
53	SQLi, XSS	VCP	Low*				SQLi accuracy: 89.99%–95.99%. XSS accuracy: 73.01%–83%.	[91]

Table 6. The Effectiveness of WAVD Methods on Input Validation Vulnerabilities (Part II)

ID	Vulnerability	Type	FPR	FNR	TPR	Precision	Other metrics	Ref.
54	SQLi, XSS	VCP					Detection rate: 80%.	[118]
55	**SQLi, XSS**	**VCP**	**0.00%**	**0.00%**				**[119]**
56	SQLi, XSS	VCPM	Low*		SQLi: 87%; XSS: 86%			[95]
57	File injection	MFM						[145]
58	Remote code execution	VCP	22.00%					[57]([126])
59	Remote code execution	VCP	0.00%					[126]
60	SPARQLi/SPARULi	VCP+GACP	0%–20%				Detection rate: 80%–100%.	[130]
61	**XQuery injection**	**VCP+GACP**		**8.00%**	**92.00%**	**95.80%**	**Code coverage: High*.**	**[75]**
62	**Inconsistency of client and server-side input validation**	**MFM**	**0.00%**	**0.00%**				**[80]**
63	Object injection	MFC						[87]
64	SQLi for web service	GACP	0.00%			74.05%		[125]
65	SQLi for web service	GACP+VCP	0.00%			100.00%	Cac.	[110]
66	SQLi, XSS, XMLi, XPathi, LDAPi,	MFM					Reduce security slices to 76%.	[6]
67	SQLi, XSS, CRLFi	MFM	0%–28%	0%–24%	72%–100%		Reported 73 unknown VLs.	[61]
68	**SQLi, XSS, XPathi, XMLi, LDAPi**	**MFM+VCPM**	**0.00%**	**2.00%**	**98.00%**	**100.00%**	**Cac.**	**[142]**
69	**SQLi, XSS, RFI, LFI, DT/PT, SCD, OSCI, PHPCI**	**MFM+CPM**	**Low***	**0.00%**		**92.50%**	**Accuracy: 92.1%.**	**[7]**
70	**SQLi, XSS, RFI, LFI, DT/PT, PHPCI**	**MFM+CPM**	**12.00%**	**0.00%**		**95.00%**	**Accuracy: 96%.**	**[52]**
71	Code injection, XSS	CPC			<80%		AUC: 0.616–0.765.	[14]
72	SQLi, XSS, Remote code injection, and File injection	CPM	CF: 0%–7%; RF: 1%–11%		CF: 42%–80%; RF: 22%–66%	CF: 59%–81%; RF: 26%–68%		[56]
73	**SQLi, XSS, Directory Traversal, RFI**	**CPM**	**0.001%–0.03%**		**99.95%–100%**		**F-measure: 99.5%–99.96%.**	**[141]**
74	Injection attacks	VCP						[82] ([127, 128])
75	SQLi, XSS, command injection, RFI,	VCP	0.88%				Detection rate: 82.57%.	[151]
76	**SQLi and stored injection attacks**	**VCPM**	**0.00%**	**0.00%**				**[140]**
77	SQLi, File injection, OSCI	GACP		Low*				[51]
78	SQLi, XSS, LFI/RFI	GACP					Accuracy: 20%–90%. Mean of accuracy: 54%.	[16]

LFI: Local File Inclusion; RFI: Remote File Inclusion; DTPT: Directory Traversal or Path Traversal; OSCI: OS Command Injection; PHPCI: PHP Command Injection; SCD: Source Code Disclosure.
FPR: False Positive Rate; FNR: False Negative Rate; TPR: True Positive Rate.
CF: Co-trained Random Forest; RF: Random Forest.
Cac. means that we calculated the value based on data in the primary study. The reason for calculation and detailed calculation methods are explained in Table B.2 in Appendix B.
Low* and High* mean that the primary study did not provide exact numbers but give qualitative results such as low or high.
In the reference column, the reference in the parentheses is a following up and evaluation study of the one in the bracket.

- **MFM+VCPM.** The approach in Reference [150] combined taint analysis with taint tracking, which is based on runtime monitoring, and reported 0% FPR and 0% FNR. However, the total number of true SQLi and XSS vulnerabilities identified in the evaluation of Reference [150] was small, which indicates that evaluation with more applications may be needed.
- **GAM.** Reference [79] applied a goal-directed model-checking system that automatically generates attacks and reports 0% FPR and 0% FNR, but the approach requires that "*the analyst specifies the vulnerability of interest, and care must be taken when developing the specification*—missing *a propagator may lead to false negatives while missing sanitizers is likely to lead to many false positives.*" Reference [138] generates test cases using logic programming and model-based testing. Although Reference [138] reports low FPRs and FNRs, the FPRs and FNRs may also be highly dependent on the quality of the models. In addition, the evaluation of Reference [138] is based on only one vulnerable application.

  - **VCP.** Reference [119] requires analysts to manually set rules and then uses the rules to match the application's response to the injected attack. If the response matches a specific rule, then the vulnerability will be reported. Although the evaluation in Reference [119] reported 0% FPR and 0% FNR of detection SQLi and XSS attacks, the effectiveness of the tool in Reference [119] may be highly dependent on the quality of the rules.
- **Other kinds of injection vulnerability.** In addition to SQLi and XSS, a few studies focus on other kinds of injection vulnerabilities. The WAVD approaches [75, 80] report complete evaluation results and high FNR and FPR values. Reference [75] focuses on detecting XQuery injection and Reference [80] focuses on detecting inconsistency of client and server-side input validation.
  - **XQuery injection.** Reference [75] used a **VCP plus GACP** approach that uses black-box fuzzing to detect XQuery injection vulnerabilities and achieved low FNR and high precision.
  - **Inconsistency of client and server-side input validation.** Reference [80] used an **MFM** approach that models both client-side and server-side input validation functions to **deterministic finite automata (DFA)** and compares the DFAs to identify inconsistencies between them. Evaluation of the approach in Reference [80] shows that the approach can find all inconsistencies without false positives.
- **Multiple injection vulnerabilities.** Some studies have attempted to identify several injection vulnerabilities using one approach or in one study.
  - **MFM+VCPM.** Reference [142] was performed by the same team as Reference [6] and focused on detecting XSS, SQLi, XML injection, XPath injection, and LDAP injection. In addition to security slicing, Reference [142] enforced constraints on the input string for different vulnerabilities and achieved 98% TPR and 100% precision.
  - **MFM+CPM.** References [7, 52] are from the same group of researchers and focus on detecting XSS, SQLi, local file inclusion, PHP command injection, and so on. Their idea was to use a machine learning approach to reduce the false positives of taint analysis. The evaluations in References [7, 52] showed that such a combination can achieve 92.1% accuracy [7] and 96% accuracy [52].
  - **CPM.** Reference [141] applied an adaptive learning algorithm to evolve the vulnerability prediction model based on unknown queries that are classified as benign and malicious by a **web application firewall (WAF)** and achieved an F-measure of 94.79% and 0.09% FPR. The vulnerabilities covered in Reference [141] include XSS, SQLi, remote file inclusion, and directory traversal.
  - **VCPM.** Reference [140] focused on various injection attack targeting at DBMSs and proposed a method for catching SQLi attacks and stored injection attacks, which was obtained from a training phase, by comparing queries with query models, and the method achieved 0% FPR and 0% FNR.

From a close look at the 19 approaches, we find the following:

- Using taint analysis solely, e.g., References [12, 64, 81], may result in a high false-positive rate, because many paths of web applications are unreachable from static analysis. When the unreachable path behavior conforms to the vulnerability behavior, it is misclassified as vulnerable, which results in high FPRs.
- Six out of the 19 studies applied MFM or combined MFM with other approaches, such as **MFM** [80], **MFM+VCP** [23], **MFM+VCPM** [142], **MFM+VCPM** [150], and **MFM+CPM** [7, 52].

- Combining the advantages of multiple techniques can reduce FPR and FNR. For example, to detect XSS and SQLi, a combination of goal-directed model-checking and attack generation in Reference [79] leads to 0% FPR and 0% FNR. **GACP+GAM** [92] and **VCP+GACP** [75] combined several meta-approaches and achieved low FPR and FNR.
- The specific mechanisms proposed may also reduce FPR and FNR. For example, Medeiros et al. [140] proposed a mechanism that is trained by forcing calls to all queries in an application to generate query models. The models are then updated after a new release of an application.

Besides analyzing the 19 studies, we had a close look at other injection vulnerability detection approaches, which report low FPR. We find the following:

- Complete attack rule libraries and signatures, full coverage of constraints, and full coverage of structure tracking can reduce FPR. For example, Singh et al. [114] attempted to make different libraries for different databases to expand the attack rule libraries to avoid FPR. Zheng et al. [126] indicated that a string-only analysis will likely miss pure integer or string-to-integer constraints (e.g., length of the string) and will, therefore, result in path constraints that are not precise enough and lead to false positives. Thus, they proposed a complete string constraint solver Z3-Str that can reduce FPR to 0%.

A high-level summary of time consumption, automation, and memory consumption of the injection vulnerability detection approaches is as follows. More detailed efficiency data can be found in Tables A.1 and A.2 in the Appendix.

- **Time and memory consumption**. **The time and memory consumption reported in the primary studies related to the studies' experimental settings are not comparable**. It is essential to report such data in evaluation to help users estimate the hardware resources required to use the approach and performance overhead. Only 50% (39 out of 78) of the primary studies present the time consumption of their methods or tools, and only eight articles [55, 60, 61, 63, 80, 132, 136, 145] analyzed and reported memory consumption.
- **Automation**. Not all WAVD methods for detecting injection vulnerabilities are fully automated. The **CPM** approach, e.g., References [107, 141], and the **CPC** approach, e.g., Reference [65], need manual tagging of the vulnerable code to build the prediction model. The **VCP** methods [55, 83] allow users to add rules manually to complement the automatically generated rules. In the **GACP** method in Reference [113], one step is web crawling. In addition to automatic web crawling, crawling can also be performed manually.

5.3.2 The Efficiency and Effectiveness of Approaches Detecting Other Vulnerabilities. The other vulnerability detection approaches' effectiveness data are shown in Table 7. More detailed such data are included in Table A.3 in Appendix A.

For studies detecting other vulnerabilities, we found eight approaches if we choose approaches by applying similar FPR and FNR criteria as the injection vulnerabilities. The studies are marked in bold in Table 7. These eight studies focused on detecting vulnerabilities include parameter tampering, access-control, workflow bypass, path traversal, horizontal privilege escalation, clickjacking, and phishing website. The summaries of these eight approaches are as follows:

- **Parameter tampering.**
  - **VCP.** "*Parameter tampering attack vectors sometimes arise because the developer simply fails to realize that the client code checks should be replicated on the server-side* [121]." Reference [109] used the constraints generated from the client-side to verify the server-side code. Bisht et al. [121] summarized the reason for high FPR and FRN of detecting parameter

tampering, which either pertained to the max-length constraints on form inputs that could not be exploited to any serious vulnerability or related to rewriting by the server without any observable difference in HTML output. Bisht et al. addressed these issues in Reference [109] and achieved high precision and recall.
  - **VCP or VCPM + GACP.** The approaches in References [74, 75] first learn the parameter constraint from analyzing the client-side code and legal HTTP requests and responses and then generate illegal requests based on the legal requests to attack the server-side application to determine whether the illegal requests are checked at the server-side. The approach in Reference [74] also detects access-control and workflow bypass vulnerabilities well.
- **Path traversal.**
  - **MFM**. Through fine-grained dataflow analysis of PHP built-in features, the approach in Reference [61] identified three path traversal vulnerabilities from five popular applications with 0% FPR. Although the authors reported 0% FNR in Reference [61], they used only **Common Vulnerabilities and Exposures (CVE)** entries to measure the number of path traversal vulnerabilities in the tested applications, which may not be very accurate. Although References [7, 52, 100] reported low FPR and FNR of several vulnerabilities, including path traversal, they did not report exact FPR and FNR values for detecting each vulnerability.
- **Horizontal privilege escalation**.
  - **VCP.** The approach in Reference [105] first analyzes the application code to identify access-control constraints and then checks if the constraints are consistent across the application. Any inconsistencies identified may indicate a potential access-control vulnerability. The approach in Reference [105] identifies several zero-day authorization vulnerabilities with no false positives.
- **Clickjacking.**
  - **MFM.** The approach in Reference [147] first builds a behavior model based on **Finite State Automaton (FSA)** from known clickjacking attacks and legitimate web pages. If the sequence of states of the user matches the attack signature, which is the information embedded in FSA, then an attack warning is raised.
  - **VCP.** Reference [152] presented a **VCP** idea, of using a client-side proxy to filter malicious input by analyzing the parameter value of request pages and JavaScript code of the response pages to detect clickjacking vulnerabilities. References [147, 152] seemed to perform well with FPR up to 7.8% and 0% FNR. However, user behavior (e.g., clicking) has a significant influence on FPR and FNR when detecting clickjacking vulnerabilities. All evaluations of the clickjacking approaches are carried out in a laboratory, which may have eliminated the influence of human factors and therefore showed effective results.
- **Phishing website.**
  - **CPC.** Reference [144] presented an approach that uses machine learning to classify and predict phishing websites. The evaluation classified 9,076 test websites. Although the authors in Reference [144] reported the high accuracy of their approach, they did not give a detailed list of the tested websites.

The evaluation of other WAVD approaches listed in Table 7 did not show fewer than 10% of both FPR and FNR. The approaches and their effectiveness are summarized as follows:

- **Path disclosure.** Reference [14] introduced a **CPC** approach and claimed that their machine learning approach could detect path disclosure vulnerability. However, no details about the number of path disclosure vulnerabilities identified were presented in Reference [14].

Table 7. The Effectiveness of WAVD Methods on Other Vulnerabilities

ID	Vulnerability	Type	FPR	FNR	TPR	Precision	Other metrics	Ref.
1	Parameter tampering	GACP	1.12%					[120]
2	**Parameter tampering**	**VCP**	**0.00%**	**0.00%**			**Identified 45 unknown VLs**	**[109]**
3	Parameter tampering	VCP+GACP					FP: 43.	[121] ([109])
4	**Parameter tampering**	**VCP+GACP**	**0.00%**	**0.00%**	**100.00%**	**100.00%**	**Code coverage: High*; Cac.**	**[75]**
5	**Parameter tampering Access-control Workflow bypass**	**VCPM+GACP**	**0%–5%**	**0%–6.25%**	**97.9%**	**99.10%**	**Parameter tampering: 0% FP and FN. Access control: 5% FP and 6.25% FN. Workflow bypass: 0% FP and FN.**	**[74]**
6	**Path traversal**	**MFM**	**0.00%**	**0.00%**			**Built-in functions coverage: 89%. Report 73 unknown VLs.**	**[61]**
7	Path traversal	MFM	7.00%		93.00%		Identified 159 second-order VLs. No detailed path traversal data.	[60]
8	Path traversal	MFM+CPM	Low*	0.00%		92.50%	Accuracy: 92.1%. No detailed path traversal data.	[7]
9	Path traversal	MFM+CPM	12.00%	0.00%		95.00%	Does not identify any path traversal VL	[52]
10	Path disclosure, Authorization issue	CPC					AUC: 0.616–0.765.	[14]
11	Logic vulnerabilities	VCPM		FB: 0–33.3% PM: 0–100%	FB: 0–66.7% PM: 0–100%	FB: 0–100% PM: 0–88.9%	Cac.	[66]
12	Logic vulnerabilities	GAM			6.60%		6.6% real bugs; 93.4% harmless presentation.	[89]
13	Logic vulnerabilities	VCPM				86.00%		[88]
14	Unvalidated redirects and insecure direct object references	GACP	Low*	Low*			No exact FPR or FNR data.	[100]
15	Insecure Direct Object References.	MFM	High*			97.00%		[71]
16	Execution after redirect	MFM			59.90%			[108]
17	State violation	VCP	Low*					[59]
18	State violation	VCP	Low*					[62]
19	State violation	VCPM	Low*	0.00%				[70]
20	Missing authorization check	MFM+VCPM	Low*				Reported 47 unknown VLs.	[69]
21	Access-control errors	VCP					Identified 38 access-control VLs.	[67]
22	Authorization logic error	VCP	Low*	Low*				[9]
23	**Horizontal privilege escalation**	**VCP**	**0.00%**		**100.00%**		**Cac.**	**[105]**
24	Access control	VCPM	Low*				Coverage: 79.33%–100%.	[68]
25	Access control	VCPM+GACP	0.00%			100.00%	Code coverage: 88.16%–92.21%. Cac.	[15]
26	Access control	VCPM+GACP	Low*				Code coverage: 58.7%–81.87%.	[98]
27	Clickjacking	MFC	0%–5%				Accuracy: >80%.	[90]
28	**Clickjacking**	**MFM**	**0.28%–7%**	**0.00%**	**92.22%–98.78%**			**[147]**
29	**Clickjacking**	**VCP**	**0%–7.84%**	**0.00%**				**[152]**
30	Session fixation	MFM						[11]
31	CSRF	GACP					Accuracy: 20%–90%. Mean of accuracy: 54%.	[16]
32	CSRF	CPC				22.1%–84.9%	Recall: 13.8%–85.1%. F1-score: 16.9%–84.8%.	[139]
33	CSRF	CPC					AUC: 0.616–0.765.	[14]
34	Side channel	CPM				6.8%–96.3%		[102]
35	Side channel	GAM	0.00%					[101]
36	DoS	MFM+VCPM	High*					[69]
37	Second-order DoS	GACP	33.00%			42.1%–100%	Cac.	[21]
38	Configuration	GACP	Low*	Low*			No exact FPR or FNR data.	[100]
39	Configuration	MFC					Identified more VLs than three other scanners.	[50]
40	**Phishing website**	**CPC**	**4.4%–4.9%**	**0.55%–0.74%**	**99.26%–99.45%**	**87.56%–88.89%**	**Accuracy: 96.23%–96.58. Cac.**	**[144]**
41	Web shell	MFM	0.00%		81.24%			[86]
42	No information in the paper	MFC					Average VLs per site: 13.1.	[85]

FPR: False Positive Rate; FNR: False Negative Rate; TPR: True Positive Rate; FB: Forceful Browsing; PM: Parameter Manipulation; Cac. means that we calculated the value based on data in the primary study. The reason for calculation and detailed calculation methods are explained in Table B.2 in Appendix B. Low* and High* mean that the primary study did not provide exact numbers but give qualitative results such as low or high. In the reference column, the reference in the parentheses is a following up and evaluation study of the one in the bracket.

- **Logic vulnerability.** Logic vulnerability is difficult to identify, because it is specific to the intended functionality of the application [88].
  - **VCPM.** Reference [66] tested the application at each state by constructing unexpected input vectors and evaluating corresponding web responses to detect *parameter manipulation* and *forceful browsing* according to the discrepancies between the intended FSM and the implementation FSM, and finally identified unknown logic vulnerabilities but reported 7 false positives out of 18 true positives.
  - **GAM.** Reference [89] began by modeling the behavior of an application based on the execution trace of benign applications and then generated attacks based on known attack patterns. The approaches in Reference [89] identified unknown logic vulnerabilities but reported 6.6% TPR.
  - **VCPM.** Reference [88] identified invariants from the application and then used model checking to determine whether the invariants were violated. The approach presented in Reference [88] identified unknown logic vulnerabilities but reported 8 false positives out of 48 true positives.
- **Unvalidated redirect and insure direct object reference.**
  - **GACP.** Reference [100] generated attacks by modifying suspicious URLs to detect unvalidated redirect vulnerability and by modifying object value to detect insecure direct object reference. The tool in Reference [100] was compared with three scanners and did not cover these two vulnerabilities. Although the overall FPR and FNR of the tool in Reference [100] was higher than the three compared scanners, the exact FPR and FNR of detecting unvalidated redirect vulnerability was not reported.
  - **MFM.** Reference [71] used taint analysis to identify insecure direct object reference vulnerability, and the approach reported several false positives.
- **Execution after redirect (EAR).** EAR is a logic flaw when the indented execution of the server-side code is expected to halt after redirection, but the execution continues. Reference [108] used **MFM** approach and was the only study focusing on this vulnerability. The approach analyzed the control flow of the code to determine whether the execution halted after redirection. From 1,173 OSS projects, the tool in Reference [108] identified 3,944 instances of the EAR, in which 855 were vulnerable.
- **State violation.** State violation (also called workflow violation) occurs if there exists a path leading to restrictive function with insufficient or erroneous checking of session variables [62]. References [59, 62] presented **VCP**-type approaches that included two phases to detect state violations. The first phase learned input, input/output, or input/output sequence invariants [62] or axioms [59] from attack-free execution of the application. The second phase used the invariants or axioms to evaluate each web request and responses to detect any violations. References [62] and [59] used a similar testbed and reported low FPR without reporting FNR data. Reference [70] introduced a **VCPM** approach that extracted invariants and then used the invariants to identify malicious SQL commands. Evaluations in Reference [70] reported no false negative. The authors of Reference [70] claimed low FPRs without giving concrete numbers. The authors analyzed the false positives and concluded that false positives "*can be introduced by the incomplete exploration of user simulators, which is known as an inherent challenge for dynamic analysis techniques.*"
- **Other access control vulnerabilities.** Reference [69] described an **MFM** approach that uses taint analysis to identify missing authorization checks and report low FPRs. The **VCP**-type approach in Reference [67] "*starts with a high-level specification that indicates the conditional statement of a correct access-control check and automatically computes an interprocedural access-control template (ACT)*" and then used ACT to find faulty access-control

logic that misses the conditional statements. Several unknown vulnerabilities were detected by the approach in Reference [67] and were also auto fixed. However, Reference [67] did not report FPR or FNR data. The **VCP**-type algorithm in Reference [9] first monitors HTTP traffic and builds user-data-access policies (a developer can also update the policies) and then implements a proxy to observe the HTTP traffic to detect unauthorized data disclosure with low FPR. Reference [68] applied **VCPM** approach; it first performs static analysis to build sitemaps for different roles from CFG and then directly accesses privileged pages from unprivileged roles to detect missing or insufficient access checks. The approaches in References [15, 98] combine **VCPM** and **GACP**, which start with a dynamic analysis to identify database access operations that are allowed for each role and user and then generate test cases to verify whether a role or a user can perform a privilege escalation attack. Both tools in References [15] and [98] reported low FPRs, but did not report FNRs.

- **Session fixation.** Only one study [11] focused on detection session fixation vulnerability, and the approach was an **MFM**-type approach. It abstracts the session states and then checks the session's abstract value at the authentication instruction. The evaluation of the approach in Reference [11] only presents its speed without giving information about its FPR and FNR.
- **CSRF.** Reference [16] proposed a **GACP**-type approach, i.e., generating penetration test cases from known attacks to detect CSRF vulnerabilities, and References [14, 139] used the **CPC** approach to predict the CSRF vulnerabilities from code features of vulnerable historical code.
- **Side-channel.** The approach in Reference [102] was based on classifying using models, i.e., a **CPM** approach, and its precision varied from 6.8% to 96.3% in the evaluation. The **GAM**-type approach in Reference [101] began by detecting the side channels of a web application using models and then performing "*a rerun test to assess the amount of information disclosed.*" The evaluations in References [101, 102] are based on a small number of web applications, and no FNR data were reported.
- **DoS.** Reference [69] used taint analysis and symbolic execution, which is a combination of **MFM** and **VCPM**, to identify DoS vulnerabilities, and the FPR was high.
- **Second-order DoS.** Only Reference [21] studied second-order DoS using the **GACP** approach, which uses backwards symbolic execution algorithm for generating candidate attack vectors. However, the approach reports 33% FPR.
- **Configuration.** References [50, 100] invented scanners to identify configuration vulnerabilities and used **MFC** approach and **GACP** approach, respectively. However, the scanner in Reference [50] was only evaluated by comparison with other scanners, which were not designed for identifying configuration vulnerabilities, and no FNR data were reported. The exact FNR and FPR of the scanner in Reference [100] to detect configuration vulnerability were not reported either.
- **Web shell.** "*A web shell is a script that can be run on a webserver to enable remote administration of the infected server* [86]." An **MFM** method, which combines taint analysis and pattern matching, was proposed in Reference [86] to detect web shells.
- **Any vulnerability**. Reference [85] presented an **MFC** approach to compare the source code of a web application with the fingerprint feature of vulnerable code to detect vulnerabilities. However, the authors in Reference [85] did not explain what types of vulnerabilities their tool can detect, and their evaluations reported only the speed of the tool without giving FPR or FNR.

A high-level summary of time consumption, automation, and memory consumption of the above mentioned approaches is below. More detailed efficiency data can be found in Table A.3 in the Appendix.

Table 8. The URL of Tools

Tool name and Ref.	Vulnerabilities covered by the tool	URL
PhpMinerI [65, 107]	SQLi, XSS	*http://sharlwinkhin.com/phpminer.html
RIPS [61]	SQLi, XSS,CRLFi	http://sourceforge.net/projects/rips-scanner/
WAPTEC [109]	Parameter tampering	*http://sisl.rites.uic.edu/waptec
PAPAS [120]	Parameter tampering	*http://papas.iseclab
THAPS [53]	SQLi, XSS	*https://bitbucket.org/heinep/thaps/
Notamper [121]	Parameter tampering	*http://sisl.rites.uic.edu/notamper
PIUIVT [118]	SQLi, XSS	*https://sites.google.com/site/asergrp/projects/PIUIVT
Z3-Str [126]	Remote code execution	*http://www.cs.purdue.edu/homes/zheng16/str
JOACO [142]	SQLi, XSS, XPathi, XMLi, LDAPi	https://sites.google.com/site/joacosite/home
XSSDE [77]	XSS	*http://sharlwinkhin.com/security-auditing.html
Black-box tool [102]	Side channel	http://www.cs.virginia.edu/sca
Ardilla [64]	SQLi, XSS	http://groups.csail.mit.edu/pag/ardilla/
White-box tool [108]	Execution after redirect	https://github.com/adamdoupe/find_ear_rails

* The URL is no longer valid or accessible on 9th February 2021.

- **Time and memory consumption**. Twenty-one out of the 42 primary studies in Table 7 reported their time consumption, and only three studies [60, 61, 109] reported their memory consumption.
- **Automation**. In the **VCPM+GACP** approach implemented in Reference [74], manual traces can be combined with automatically generated traces to infer the control flow of the application. Similarly, manual intervention from the tester can also be applied in the **VCP+GACP** approach in Reference [121] to complement the inputs from the client to represent a complete picture of the valid logic of a system. **CPC** approaches [14, 139, 144] usually need manual tagging to build the prediction models.

5.3.3 Publicly Available Tools Derived from the Approaches. Although 61.9% of the 105 primary studies proposed and developed WAVD tools, only 13.3% of them provided URLs of their tools. The URLs of tools are listed in Table 8. However, most of the URLs are no longer valid or accessible.

5.4 Results of RQ3

To answer RQ3, we first analyzed the test suites and web applications used to evaluate the primary studies' WAVD approaches. We then summarized the vulnerabilities inserted or identified in the test suites and web applications.

Although 92 of the 105 primary studies presented detailed information on the test suites or web applications they used, 13 studies did not provide details. For example, in References [20, 84, 85, 90, 111, 115, 144, 147, 151, 152], the authors only stated that "we evaluated our approach on *n* web applications." In References [130, 145], the authors only mentioned the test suites, e.g., OWASP top 10 attack test suites and Stivalet test suites [157], without specifying which parts of the test suites are used in their evaluation. In Reference [132], we could not find information about the vulnerable web applications used in their evaluation. When we tried to obtain access to the test suites or web applications listed in the primary studies, we found many invalid links. This is probably because of the update of the servers that host the test suite. The test suites and web applications that were not accessible were excluded from our analysis.

As shown in Figure 4, most primary studies focus on using applications developed using Java/JSP or PHP to evaluate their WAVD approaches. Only a few studies focus on web applications developed using other programming languages. For example, Reference [108] evaluated its approach using 59,255 open-source web applications developed using Ruby on Rails, and Reference [82] used the JavaScript test suite. We listed the applications and test suites used at least three times in primary studies for PHP-based web applications in Table 9. The most frequently

Table 9. PHP-based Applications

FQ	Application	URL	Vulnerabilities covered by primary studies	Reference
13	**WordPress**	wordpress.org/	SQLi, XSS, Configuration, RFI, LFI, SCD, DT, PPCI, DoS, IDOR, UR.	[9, 21, 49, 50, 52–54, 86, 95] [83, 100, 153] **[143]**
11	**Scarf**	www.sourceforge.net/projects/scarf/files/	DoS, XSS, SQLi. LFI, RCE, Access-control, PM.	[21, 60, 66–68, 98, 143] **[62, 70, 74, 105]**
9	**OsCommerce**	www.oscommerce.com/	XSS, DoS, Workflow violation, SQLi, LFI, RCE, Var tampering, PT.	[21, 51, 59, 60, 89, 143] **[61, 62, 74]**
9	**PHPMyAdmin**	*www.phpmyadmin.net/	CI, SQLi, CSRF, XSS, PD, DD, Configuration, RCE, File inclusion.	[9, 50, 56, 57, 65, 107, 126] **[14, 139]**
8	**Wackopicko**	github.com/adamdoupe/WackoPicko.git	PM, SQLi, XSS, Access control, Workflow violation.	[18, 66, 91, 98, 143] **[62, 70, 74]**
7	**Drupal**	www.drupal.org/download	SQLi, XSS, CI, CSRF, PD, Configuration.	[9, 49, 50, 55] **[14, 139, 143]**
7	**PhpBB**	www.phpbb.com/	XSS, Configuration, SQLi, XPathi, OSCI, File include, PT.	[9, 50, 51, 58, 83, 143] **[61]**
7	**MyBB**	sourceforge.net/projects/mybb/	Configuration, XSS, SQLi, PM, PT, RFI, OSCI, File write, Access-control.	[50, 67, 68, 83, 95, 140] **[61]**
6	**Schoolmate**	sourceforge.net/projects/schoolmate/	XSS, SQLi, MFE, RCE, File inclusion	[56, 63–65, 140] **[107]**
6	**Moodle**	sourceforge.net/projects/moodle/	CI, CSRF, SQLi, XSS, PD, MFE, Configuration.	[50, 63, 83, 95] **[14, 139]**
6	**Yapig**	sourceforge.net/projects/yapig/files/yapig/yapig%200.95b/	SQLi, XSS, Access-control.	[55, 65, 67, 68, 106] **[107]**
5	Faqforge	sourceforge.net/projects/faqforge/	SQLi, XSS, RCE, MFE, File inclusion.	[56, 63–65, 107]
5	DVWA	www.dvwa.co.uk	File inclusion, XSS, SQLi, CSRF, LFI/ RFI, CJ, UR, XPathi, OSCI.	[16, 51, 91, 138, 155]
5	**MyBloggie**	sourceforge.net/projects/mybloggie/	XSS, SQLi. LFI, RCE, PM, DoS, Access control.	[60, 69, 109, 121] **[105]**
5	**BloggIT**	*www.sourceforge.net/	XSS, SQLI, PM, Workflow violation.	[59, 66, 98, 143] **[62]**
4	mutillidae	sourceforge.net/projects/mutillidae	SQLi, XSS, Configuration, UR, IDOR, Session fixation.	[11, 18, 91, 100]
4	Webchess	sourceforge.net/projects/webchess/	XSS, SQLi, MFE, RFI, OSCI, RCI.	[63–65, 140]
4	Events lister	www.exploit-db.com/exploits/17554/	PM, Forceful browsing attacks, Access-control.	[66–68, 98]
4	**Minibloggie**	www.exploit-db.com/exploits/6782/	Access-control, DoS, SQLi.	[67, 69, 98] **[105]**
4	**OpenIT**	sourceforge.net/projects/openit/	Access-control, PM.	[66, 121] **[70, 74]**
3	HotCRP	www.read.seas.harvard.edu/~kohler/hotcrp/	DoS, XSS, SQLi. LFI, RCE, Var tampering, PT.	[21, 60, 61]
3	Dnscript	dnscrypt.info/	Access-control, DoS, SQLi.	[67, 69, 105]
3	PunBB	github.com/punbb/punbb	SQLi, XSS, Workflow violation.	[59, 83, 95]
3	phpMyFAQ	www.phpmyfaq.de/	RCE, Session fixation.	[11, 57, 126]

* The URL is no longer valid or accessible on 9th February 2021.
FQ: Frequency of use; OSCI: Operating System Command Injection; CSRF: Cross-site Request Forgery; LFI/RFI: Local/Remote File Inclusion; CI: Code Injection; SCD: Source Code Disclosure; DT: Directory Traversal; PT: Path Traversal; PPCI: PHP Command Injection; DoS: Denial of Service; DD: Data Disclosure; PD: Path Disclosure; LDAP: Lightweight Directory Access Protocol; UR: Unvalidated Redirects; IDOR: Insecure Direct Object References; PM: Parameter Manipulation/tampering; MFE: Malicious File Execution; CJ: Clickjacking; RCE: Remote Code Execution.

used PHP-based web applications to evaluate WAVD approaches are WordPress, SCARF, and OsCommerce. We listed Java-based web applications and test suites used at least twice in primary studies in Table 10. Several Java-based web applications in the AMNESIA [73] and JOACO-Suite were the most popularly used ones.

From the primary studies, we found a few large-scale test suites, which have not been used frequently yet. However, we believe these test suites can contribute to evaluating vulnerability detection in the future.

- **Complete JOACO-Suite** includes 11 open-source security benchmark applications and Java web applications/services that were used in Reference [142], with known XMLi, LDAPi, SQLi, XPathi, and XSS vulnerabilities. As shown in Table 10, some of the open-source applications in this suite, such as WebGoat, Roller, and TPC-APP, have been used in the evaluation of WAVD approaches in primary studies. We believe that other applications, such as Bodgeit [163], openmrs-module-legacyui [164], and Regain [165], can also be valuable to evaluate WAVD approaches in the future.

Table 10. Java-based Applications and Test Suites

FQ	Application or test suite	URL	Vulnerabilities covered by primary studies	Reference
12	**Applications in AMNESIA suite**	viterbi-web.usc.edu/$\sim$halfond/testbed.html	SQLi, PM, XQuery injection, XSS.	[72, 75–78, 83, 94, 96, 116, 136, 149] [**74**]
4	**WebGoat in JOACO-suite**	www.owasp.org/index.php/Category: OWASP_WebGoat_Project	XSS, SQLi, XMLi, XPathi, LDAPi, IDOR.	[6, 71, 72] [**142**]
	TPC-App in JOACO-suite	www.tpc.org/	XSS, SQLi, XMLi, XPathi, LDAPi.	[6] [**110, 125, 142**]
	TPC-C in JOACO-suite			
	TPC-W in JOACO-suite			
3	Roller in JOACO-suite	roller.apache.org/	XSS, SQLi, XMLi, XPathi, LDAPi, IDOR.	[6, 71] [**142**]
	Pebble in JOACO-suite	pebble.sourceforge.net/		
	PersonalBlog	github.com/suyeq/personalblog	SQLI, XSS.	[76, 77, 79]
	Jgossip	sourceforge.net/projects/jgossipforum/	SQLi, XSS.	[76, 79, 80]
2	Jorganizer	sourceforge.net/projects/jorganizer/	SQLi, XSS.	[77, 79]
	Education	*www.jiaoyudaohang.com	SQLi, XSS.	[117] [**114**]
	Jsp Forum	sourceforge.net/projects/jsforum/	Access control, Logic vulnerability.	[88, 98]

* The URL is no longer valid or accessible on 9th February 2021.
FQ: Frequency of Use; IDOR: Insecure Direct Object References; PM: Parameter Manipulation/tampering.

- **The HTTP dataset CSIC 2010** [166] used in Reference [141] contains 36,000 benign requests and 25,000 malicious requests, which makes this dataset suitable for comparing malicious query detection systems. The dataset contains the normal traffic set for training and the test, and the malicious traffic set for the test.
- **Suite-9408 PHP source code** [167] used in Reference [146] has 5,600 nonvulnerable files and 3,808 vulnerable files. The test suite is more suitable than NVD [160], Bugzilla [161], and NIST [162] for evaluating XSS vulnerability detection approaches because NVD and Bugzilla provide only vulnerability information without providing source code, and NITS provides a dataset with only 80 PHP source code files.
- **StrangerJ-Suite** is a security benchmark extracted from five real-world PHP web applications: *MyEasyMarket, PBLguestbook, proManager, BloggIT,* and *aphpkb*. StrangerJ-Suite has been used to evaluate the effectiveness of the Stranger tool [158], which can automatically detect security vulnerabilities in PHP web applications. Moreover, the StrangerJ-Suite benchmark contains nine paths, which are all vulnerable to XSS.
- **Pisa-Suite** includes 12 constraints generated from sanitizers detected by PISA [159].
- **AppScan-Suite** includes eight constraints derived from the security warnings emitted by IBM Security AppScan [168], which is a commercial vulnerability scanner tool. The generated security warnings contain traces of program statements that reflect potentially vulnerable information flows when implementing the IBM Security AppScan on a set of popular websites.
- **Juliet Test Suite** [169] used in Reference [81] is a collection of test cases in the C/C++ and Java language for all types of security vulnerabilities, which consists of many small programs, and each program contains a class of artificial vulnerability. Each test case is usually focused on one specific type of vulnerability but still might be attached to other vulnerabilities at the same time.

To precisely calculate false positives and false negatives of WAVD approaches, we must know the number of vulnerabilities in the web applications or test suites. Among the 105 primary studies, only 21 studies presented the number of each type of vulnerabilities detected in WAVD approach evaluations. For example, Reference [119] listed the CVE ID of the identified vulnerabilities. In the application column of Table 9, the applications marked in bold are the frequently used PHP-based applications with known numbers of vulnerabilities. In the reference column of Table 9, the corresponding primary studies, which report the number of each type of vulnerabilities detected, are also marked in bold. Data in Table 9 show that a few frequently used PHP-based applications do not provide the number of vulnerabilities, which may make it challenging to calculate false negatives when evaluating WAVD approaches. In Table 10, we show the same data for Java-based applications and test suites. A complete list of the number and types of vulnerabilities reported in the 21 studies is shown in Table C.1 in Appendix C.

Without knowing the specific number of vulnerabilities in the applications or test suites, some primary studies, such as References [56, 66, 122], attempted to measure the effectiveness of the WAVD approaches through the following methods:

- **Injecting a known number of malicious attacks or codes or generating a known number of test cases to simulate attacks.** For example, Li et al. [66] designed a prototype system LogicScope, generated a known number of testing specifications by the TestSpec Generator module, detected vulnerabilities by the Output Evaluator module, and finally manually analyzed the reported vulnerabilities and classified them into true positives or false positives.
- **Compared with vulnerability detection performance with existing approaches**. A few studies, such as References [56, 122], used the vulnerabilities detected by existing approaches as ground truth to evaluate false positive and false negatives of their approach and attempted to show that the new approach is better than the existing ones. However, the challenge is that the existing approaches may not reflect the real ground truth, and the results of the new approach, even if it is superior to the existing approaches, can be misleading.

6 DISCUSSION

6.1 Comparison with Related Work

Several studies have reviewed the WAVD approaches. Our studies and results are different from existing studies from three aspects.

- **Our WAVD approach categorization is more comprehensive and finer-grained than any other approach classifications.** Many studies, such as References [4, 22, 32, 37], classified the WAVD approaches to static code analysis, dynamic code analysis, and hybrid. Other studies, such as Reference [29], added category secure programming and modeling. Our study covers WAVD approaches related to more categories of vulnerabilities than existing studies listed in Table 1 and includes approaches to detect Second-order DoS [21] and misconfiguration [99, 100] vulnerabilities. Our study covers more categories of WAVD methods and includes data mining and pattern matching techniques. In addition, our analysis of the WAVD approaches shows that static, dynamic, or hybrid analysis is usually the first step to building models, properties, constraints, and patterns. After this step, web content or features in the model can be used by other analyses, such as matching, classification, verifying compliance, and generating attacks. Our study also identified more artifacts that are analyzed by various WAVD approaches than those listed in Reference [43].
- **We focus on a detailed comparison of the effectiveness and efficiency of each WAVD approach.** Seng et al. [30] identified and summarized the measurement metrics of the quality of scanners and showed that the number of false positives, number of false negatives, number of true positives, and scanning time are the main metrics applied. However, the detailed data of these metrics are not analyzed. Most other WAVD approach survey approaches did not cover effectiveness and efficiency issues. In addition to analyzing the effectiveness and efficiencies of the WAVD approaches, we considered the WAVD approaches that have high precision and recall values in depth to summarize the experience and lesson learned from

those approaches. The results of these studies will provide valuable inputs to academia and industry practitioners who want to improve the quality of their approaches and tools.

- **Our analysis of the WAVD approach testbed provides a more comprehensive overview.** Many WAVD studies used homemade applications, open-source applications, or test data sets made deliberately for evaluating WAVD methods. Although References [30, 35, 37, 39] surveyed and listed some applications for evaluating WAVD approaches, none of the studies presented a comprehensive list of the test suites, the vulnerabilities the test suites cover, and the studies that used them, as we did. Our summary of the test suites can help WAVD researchers choose the most relevant and dependable applications or test suites for evaluating their methods and tools.

6.2 Implications of the Results for Practitioners

When security engineers follow, e.g., **risk management framework (RMF)** [5, 26], or threat modeling approach [25], e.g., misuse cases [8], they sometimes need to perform a general search to identify all possible vulnerabilities or a specific type of vulnerability. The engineers need to know which tools they can use and how well the tools can work. Our results of RQ1 and RQ2 summarized what artifacts the WAVD approaches use and how well a WAVD can detect specific vulnerabilities. Knowing what artifacts the WAVD approach used can help engineers filter out WAVD approaches that are not applicable. For example, without obtaining access to the source code or execution logs of web applications, WAVD approaches that rely on such artifacts can be excluded by engineers. Knowing the precision and recall values of WAVD approaches can help engineers prioritize and choose the proper tools to use. We have also listed the URL of the tools innovated by the primary studies in Table 8, although many URLs are no longer valid. People do not want to use specific WAVD tools with high-performance overhead [133, 134]. The time and consumption data we extracted from the primary studies can help engineers estimate the overhead and cost to use WAVD approaches. Our results of RQ3 identified test suites and web applications used to evaluate the primary studies and summarized the vulnerabilities these test suites and applications cover. Companies developing WAVD tools can use them to benchmark various tools. Companies providing security education and training services can take code snippets from vulnerable applications to develop training material. Results of RQ3 also show there are insufficient high-quality test suites and applications with available lists of vulnerabilities. Industry practitioners and researchers need to work collaboratively to provide more such test suites and applicants to benefit the software security community.

6.3 Implications of the Results for Researchers

Most existing WAVD approach surveys or interviews focus on summarizing the ideas of the approaches without targeting their effectiveness and efficiencies. Our results of RQ1 and RQ2 classified the WAVD approaches and compared the precision, recall, and efficiencies of the approaches. Our results of RQ3 summarized different evaluation resources that researchers used to evaluate their WAVD tools. Based on our study's results, we have identified several research gaps.

- We found the majority (78 out of 105) of our identified primary studies focus on injection vulnerabilities. Many other vulnerabilities, such as execution after redirect, logic vulnerabilities, state violation, side channel, DoS, and configuration, are underexplored. Some vulnerabilities listed in OWASP top 10, such as **XML External Entity (XXE)** and insecure deserialization, have not been covered by many primary studies. As *"A software security system is only as secure as its weakest component"* [170], more studies on detecting those underexplored vulnerabilities are needed.

- In the software engineering domain, there are quality criteria [171], such as the explanation of design, sampling, control group, data collection, data analysis, reflexivity, and findings, to evaluate primary studies. To apply such criteria to evaluate WAVD studies' quality, we shall modify the criteria to focus more on using rigorous data collection and analysis metrics. One of the significant quality issues of WAVD studies is that researchers use different metrics to report evaluation results, as shown in Tables 5, 6, and 7, and many studies did not report the results of the essential metrics. Only 57 out of the 105 studies presented FPR or gave us information to calculate FPR. Twenty-three studies showed only the numbers of vulnerabilities the methods can identify without giving any information about the false positives. Only 34 out of the 105 studies presented FNR or provided sufficient information to calculate FNR. Although a few other studies presented FNR-related information using detection rates or accuracy, 40% (43 out of 105) of the studies did not provide any information about FNR. Part of the reasons for many studies not reporting FNR might be that people did not know the exact numbers of the vulnerabilities in applications and test suites used for evaluating WAVD approaches, as shown in Tables 9 and 10. More artifacts studies are needed to develop applications and test suites with a known number of vulnerabilities as the benchmark.
- We found that only 50% of the studies report the approaches' time consumption, and only 11 of 105 studies reported the approaches' memory consumption. We will encourage researchers to measure and report the time and memory consumption of the methods to help practitioners estimate the approach's efficiency and the resources and costs to use it.
- Most primary studies we reviewed focused on PHP and Java-based applications. None of the studies focused on web applications developed using Python, and very few focused on JavaScript-based applications. According to market studies [172, 173], Python and JavaScript are becoming the most popularly used programming languages for web application development. More WAVD studies and test suites focus on applications developed using Python and JavaScript are needed.
- Our studies identified eight WAVD meta-approaches. A single meta-approach could give highly effective results in some cases, such as References [49, 143, 149]. In other cases, such as References [7, 23, 52, 75, 142, 150], a combination of multiple meta-approaches can jointly provide good results. Along with proposing new meta-approaches, researchers can also investigate combinations of meta-approaches to optimize WAVD tools.

6.4 Threats to Validity

One possible threat to validity is missing relevant articles. To address this validity issue, the paper searching and filtering were performed by the first author and validated by the second author. The first author and the second author read all primary papers individually and then cross-validated and consolidated the results to avoid possible data analysis errors. The possible risk to the external validity of the literature review is that the review focuses only on papers targeting detecting web security vulnerabilities. The conclusions of the literature review may not be generalized to studies focusing on other kinds of security vulnerabilities. Additionally, we limited the review to scientific studies. Industrial reports and white papers that present WAVD tools, e.g., vulnerability scanners, are not included in our review.

7 CONCLUSION AND FUTURE WORK

Due to the increased risk of web security breaches, many WAVD approaches and tools have been invented to detect web application vulnerabilities in the past 10 years, especially the vulnerabilities listed in OWASP. A few existing surveys and literature reviews summarized the approaches and classified them. However, to our knowledge, no study has systematically analyzed and compared the effectiveness and efficiency of WAVD approaches. This article provides a systematic literature review of WAVD approaches proposed in the past 10 years and their efficiencies and effectiveness. Unlike the classification of existing literature reviews, our study first identifies 10 categories of artifacts that the WAVD approaches to analyze and then identified eight WAVD meta-approaches based on how the artifacts are processed. After that, our study compares the precision, recall, and efficiency of each category of the approach to detect specific vulnerabilities. The effectiveness and efficiency analyses show that a few approaches can detect SQLi, XSS, and injection vulnerabilities with satisfactory effectiveness and efficiency. However, for many other vulnerabilities, such as side-channel, CSRF, and second-order DoS, there is a lack of an effective WAVD approach. To evaluate the trustworthiness of the evaluation results of WAVD approaches, we also analyzed the web applications and test suites that are popularly used in WAVD approach evaluations. We find a lack of benchmarking web applications and test suites with known vulnerabilities inserted to evaluate and compare WAVD approaches and tools. We have identified a few research questions that deserve further investigation. The research community needs to improve the effectiveness of WAVD approaches. More studies are needed to re-evaluate and compare existing WAVD approaches using test suites with known types and numbers of vulnerabilities.

A SUPPLEMENTARY MATERIALS

See the contents of Appendix A, Appendix B, and Appendix C in supplementary materials file.

ACKNOWLEDGMENTS

The authors are grateful to the valuable comments and suggestions of the reviewers.

REFERENCES

[1] OWASP TOP. 2010–2017. The Ten Most Critical Web Application Security Risks. Retrieved on 30 June, 2021 from https://owasp.org/www-project-top-ten/2017/Top_10.

[2] F. Yu and Y. Y. Tung. 2014. Patcher: An online service for detecting, viewing and patching web application vulnerabilities. In *Proceedings of the Hawaii International Conference on System Sciences.* 4878–4886.

[3] Vandana Dwivedi, H. Yadav, and A. Jain. 2014. Web application vulnerabilities: A survey. *Int. J. Comput. Applic.* 108, 1 (2014), 25–31.

[4] Xiaowei Li and Y. Xue. 2014. A survey on server-side approaches to securing web applications. *ACM Comput. Surv.* 46, 4 (2014), 29.

[5] G. McGraw. 2006. Software security: Building security. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*.

[6] Julian Thomé, L. K. Shar, D. Bianculli, and L. Briand. 2018. Security slicing for auditing common injection vulnerabilities. *J. Syst. Softw.* 137 (Mar. 2018), 766–783.

[7] Iberia Medeiros, N. Neves, and M. Correia. 2016. Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Trans. Reliab.* 65, 1 (Mar. 2016), 54–69.

[8] Inger A. Tøndel, J. Jensen, and L. Røstad. 2010. Combining misuse cases with attack trees and security activity models. In *Proceedings of the International Conference on Availability, Reliability and Security*. 438–445.

[9] D. Muthukumaran, D. O'Keeffe, C. Priebe, and D. Eyers. 2015. FlowWatcher: Defending against data disclosure vulnerabilities in web applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*.

[10] Taeseung Lee, G. Won, S. Cho, N. Park, and D. Won. 2012. Experimentation and validation of web application's vulnerability using security testing method. *Lecture Notes in Electrical Engineering, Computer Science and its Applications*, Springer Dordrecht, 203(2012), 723–731

[11] A. Amira, A. Ouadjaout, A. Derhab, and N. Badache. 2017. Sound and static analysis of session fixation vulnerabilities in PHP web applications. In *Proceedings of the 7th ACM on Conference on Data and Application Security and Privacy.* 139–141.

[12] Xuexiong X. Yan, H. T. Ma, and Q. X. Wang. 2017. A static backward taint data analysis method for detecting web application vulnerabilities. In *Proceedings of the IEEE 9th International Conference on Communication Software and Networks (ICCSN).* IEEE, 1138–1141.

[13] J. Miller and T. Huynh. 2010. Practical elimination of external interaction vulnerabilities in web applications. *J. Web Eng.* 9, 1 (2010), 1–24.

[14] Cagatay Catal, A. Akbulut, E. Ekenoglu, and M. Alemdaroglu. 2017. Development of a software vulnerability prediction web service based on artificial neural networks. U. Kang, (Ed.) *Springer International Publishing AG*, 59–67.

[15] Shuo Wen, Y. Xue, J. Xu, H. Yang, X. Li, W. Song, and G. Si. 2016. Toward exploiting access control vulnerabilities within MongoDB backend web applications. In *Proceedings of the 40th Computer Software and Applications Conference.* IEEE, 143–153.

[16] M. N. Khalid, M. Iqbal, M. T. Alam, V. Jain, H. Mirza, and K. Rasheed. 2017. Web unique method (WUM): An open source blackbox scanner for detecting web vulnerabilities. *Int. J. Adv. Comput. Sci. Applic.* 8, 12 (Dec. 2017), 411–417.

[17] C. Wang, L. Liu, and Q. Liu. 2014. Automatic fuzz testing of web service vulnerability. In *Proceedings of the International Conference on Information and Communications Technologies (ICT).*

[18] Nor F. Awang and A. A. Manaf. 2015. Automated security testing framework for detecting SQL injection vulnerability in web application. In *Proceedings of the International Conference on Global Security, Safety, and Sustainability*, Springer, Cham, 160–171.

[19] N. Antunes and M. Vieira. 2011. Enhancing penetration testing with attack signatures and interface monitoring for the detection of injection vulnerabilities in web services. In *Proceedings of the IEEE International Conference Services Computing (SCC'11)*, IEEE CS, 104–111.

[20] Angelo Ciampa, C. A. Visaggio, and M. D. Penta. 2010. A heuristic-based approach for detecting SQL-injection vulnerabilities in web applications. In *Proceedings of the ICSE Workshop on Software Engineering for Secure Systems.* 43–49.

[21] O. Olivo, I. Dillig, and C. Lin. 2015. Detecting and exploiting second order denial-of-service vulnerabilities in web applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 616–628.

[22] Jian Chang, K. K. Venkatasubramanian, A. G. West, and I. Lee. 2013. Analyzing and defending against web-based malware. *ACM Comput. Surv.* 45, 4 (Aug. 2013), 1–35.

[23] M. K. Gupta, M. C. Govil, G. Singh, and P. Sharma. 2015. XSSDM: Towards detection and mitigation of cross-site scripting vulnerabilities in web applications. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics (ICACCI).* 2010–2015.

[24] M. Debbabi, M. Girard, and L. Poulin. 2001. Dynamic monitoring of malicious activity in software systems. In *Proceedings of the Symposium on Requirements Engineering for Information Security*.

[25] E. A. Oladimeji, S. Supakkul, and L. Chung. 2006. Security threat modeling and analysis: A Goal-oriented approach. In *Proceedings of the 10th IASTED International Conference on Software Engineering and Applications*. 178–18.

[26] W. Linda. 2020. Software risk management. *Proceedings of the American Society for Quality ControlAnnual Quality Congress.* 32–39.

[27] Priya Jyotiyana and S. Maheshwari. 2018. Techniques to detect clickjacking vulnerability in web pages. In *Optical and Wireless Technologies*, Vol. 472, Springer Singapore, 615–624.

[28] M. I. Ahmed, M. M. Hassan, and T. Bhuyian. 2018. Local file disclosure vulnerability: A case study of public-sector web applications. *J. Phys. Conf. Ser.* 933 (2018), 12011.

[29] Isatou Hydara et al. 2015. Current state of research on cross-site scripting (XSS)—A systematic literature review. *Inf. Softw. Technol.* 58 (2015), 170–186.

[30] L. K. Seng, N. Ithnin, and S. Z. M. Said. 2018. The approaches to quantify web application security scanner quality, a review. *Int. J. Adv. Comput. Res.* 8, 38 (2018).

[31] Sandeep Kumar, R. Mahajan, N. Kumar, and S. K. Khatri. 2017. A study on web application security and detecting security vulnerabilities. In *Proceedings of the 6th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO).* IEEE, 451–455.

[32] Mukesh K. Gupta, M. C. Govil, and G. Singh. 2014. Static analysis approaches to detect SQL injection and cross site scripting vulnerabilities in web applications: A survey. In *Proceedings of the IEEE International Conference on Recent Advances and Innovations in Engineering (ICRAIE'14).*

[33] Rahul Johari and P. Sharma. 2012. A survey on web application vulnerabilities (SQLIA, XSS) exploitation and security engine for SQL injection. In *Proceedings of the International Conference on Communication Systems and Network Technologies.* IEEE, 453–458.

[34] Stefano Calzavara, R. Focardi, M. Squarcina, and M. Tempesta. 2017. Surviving the web: A Journey into websession security. *ACM Comput. Surv.* 50, 1 (Mar. 2017).

[35] C. Vlsaggio. 2010. Session management vulnerabilities in today's web. In *Proc. IEEE Secur. Privacy Mag.* 8, 5 (2010), 48–56.

[36] Shashank Gupta and B. B. Gupta. 2017. Detection, avoidance, and attack pattern mechanisms in modern web application vulnerabilities: Present and future challenges. *Int. J. Cloud Applic. Comput.* 7, 3 (2017), 1–43.

[37] G. Deepa and P. S. Thilagam. 2016. *Securing web applications from injection and logic vulnerabilities: Approaches and challenges.* Inf. Softw. Technol. 74 (June 2016), 160–180.

[38] V. Prokhorenko, K. K. R. Choo, and H. Ashman. 2016. Web application protection techniques: A taxonomy. *J. Netw. Comput. Applic.* 60 (Jan. 2016), 95–112.

[39] H. Atashzar, A. Torkaman, M. Bahrololum, and M. H. Tadayon. 2011. A survey on web application vulnerabilities and countermeasures. In *Proceedings of the 6th International Conference on Computer Sciences and Convergence Information Technology (ICCIT)*. 647–652.

[40] Ana L. Hernández-Saucedo and J. Mejía. 2015. Proposal of a hybrid process to manage vulnerabilities in web applications. In *Advances in Intelligent Systems and Computing, Trends and Applications in Software Engineering*. Vol. 405. Springer Cham, 59–69.

[41] Theodoor Scholte, D. Balzarotti, and E. Kirda. 2012. Have things changed now? An empirical study on input validation vulnerabilities in web applications. *Comput. Secur*. 31, 3 (2012), 344–356.

[42] T. Huynh and J. Miller. 2010. An empirical investigation into open source web applications' implementation vulnerabilities. *Empir. Softw. Eng*. 15, 5 (2010), 556–576.

[43] Xiaoguang Qi and B. D. Davison. 2009. Web page classification: Features and algorithms. *ACM Comput. Surv*. 41, 2 (2009), 1–31.

[44] David Budgen and P. Brereton. 2006. Performing systematic literature reviews in software engineering. In *Proceedings of the 28th International Conference on Software Engineering*. ACM. New York, NY, 1051–1052.

[45] Kai Petersen, R. Feldt, S. Mujtaba, and M. Mattsson. 2008. Systematic mapping studies in software engineering. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. 71–8.

[46] B. Kitchenham, S. D. Budgen, and P. Brereton. 2015. *Evidence-based Software Engineering and Systematic Reviews*. CRC Press.

[47] Xin Huang, H. Zhang, X. Zhou, et al. 2018. Synthesizing qualitative research in software engineering. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 1207–1218.

[48] D. S. Cruzes and T. Dyba. 2011. Recommended steps for thematic synthesis in software engineering. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*. IEEE, 275–284.

[49] Shashank Gupta, B. B. Gupta, and P. Chaudhary. 2018. Hunting for DOM-based XSS vulnerabilities in mobile cloud-based online social network. *Fut. Gener. Comput. Syst*. 79 (2018), 319–336.

[50] B. Eshete, A. Villafiorita, K. Weldemariam, and M. Zulkernine. 2013. Confeagle: Automated analysis of configuration vulnerabilities in web applications. In *Proceedings of the IEEE 7th International Conference on Software Security and Reliability (SERE)*. 188–197.

[51] R. Akrout, E. Alata, M. Kaaniche, and V. Nicomette. 2014. An automated black box approach for web vulnerability identification and attack scenario generation. *J. Braz. Comput. Sci.* 20, 1 (2014), 4.

[52] I. Medeiros, N. Neves, and M. Correia. 2016. DEKANT: A static analysis tool that learns to detect web application vulnerabilities. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 1–1.

[53] T. Jensen, H. Pedersen, M. C. Olesen, and R. R. Hansen. 2012. THAPS: Automated vulnerability scanning of PHP applications. In *Proceedings of the Nordic Conference on Secure IT Systems*. 31–46.

[54] Mattia Monga, R. Paleari, and E. Passerini. 2009. A hybrid analysis framework for detecting web application vulnerabilities. In *Proceedings of the ICSE Workshop on Software Engineering for Secure Systems*. IEEE, 25–32.

[55] Gary Wassermann and Z. Su. 2008. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 13th International Conference on Software Engineering*. 171–180.

[56] Lwin K. Shar, L. C. Briand, and H. Beng Kuan Tan. 2015. Web application vulnerability prediction using hybrid program analysis and machine learning. *IEEE Trans. Depend. Secure Comput*. 12, 6 (Dec. 2015), 688–707.

[57] Yunhui H. Zheng and X. Y. Zhang. 2013. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. 652–66.

[58] Shashank Gupta and B. B. Gupta. 2016. Enhanced XSS defensive framework for web applications deployed in the virtual machines of cloud computing environment. *Procedia Technol.* 24 (Jan. 2016), 1595–1602.

[59] Shashank Gupta and B. B. Gupta. 2015. PHP-sensor: A prototype method to discover workflow violation and XSS vulnerabilities in PHP web applications. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*. 1–8.

[60] J. Dahse and T. Holz. 2014. Static detection of second-order vulnerabilities in web applications. In *Proceedings of the 23rd USENIX Security Symposium.*

[61] Johannes Dahse. 2014. Simulation of built-in PHP features for precise static code analysis. In *Proceedings of the Network & Distributed System Security Symposium (NDSS).*

[62] X. Li and Y. Xue. 2011. BLOCK: A black-box approach for detection of state violation attacks towards web applications. In *Proceedings of the 27th Computer Security Applications Conference*. 247–256.

[63] Fang Yu, M. Alkhalaf, T. Bultan, and O. H. Ibarra. 2013. Automata-based symbolic string analysis for vulnerability detection. *Form. Meth. Syst. Des.* 44, 1 (2014), 44–70.

[64] Adam Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. 2009. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the IEEE 31st International Conference on Software Engineering*. 199–20.

[65] Lwin K. Shar and H. Beng Kuan Tan. 2013. Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns. *Inf. Softw. Technol.* 55, 10 (Oct. 2013), 1767–1780.

[66] Xiaowei Li and Y. Xue. 2013. LogicScope: Automatic discovery of logic vulnerabilities within web applications. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. 481–486.

[67] S. Sooel, K. S. Mckinley, and S. Vitaly. 2013. Fix me up: Repairing access-control bugs in web applications. In *Proceedings of the Network and Distributed System Security Symposium.*

[68] F. Q. Sun, L. Xu, and Z. D. Su. 2011. Static detection of access control vulnerabilities in web applications. In *Proceedings of the 20th USENIX Security Symposium.*

[69] S. S. V. Shmatikov. 2011. SAFERPHP: Finding semantic vulnerabilities in PHP applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, 1–13.

[70] Xiaowei Li, W. Yan, and Y. Xue. 2012. SENTINEL: securing database from logic flaws in web applications. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy.* 25–3.

[71] Anders Moller and M. Schwarz. 2012. Automated detection of client-state manipulation vulnerabilities. In *Proceedings of the 34th International Conference on Software Engineering (ICSE).* 749–759.

[72] W. G. J. Halfond, A. Orso, and P. Manolios. 2008. WASP: Protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Trans. Softw. Eng*. 34, 1 (2008), 65–81.

[73] William G. J. Halfond and A. Orso. 2005. Amnesia: Analysis and monitoring for neutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. 174–183.

[74] G. Deepa, P. S. Thilagam, A. Praseed, and A. R. Pais. 2018. DetLogic: A black-box approach for detecting logic vulnerabilities in web applications. *J. Netw. Comput. Applic.* 109 (May 2018), 89–109.

[75] G. Deepa, P. S. Thilagam, F. A. Khan, A. Praseed, A. R. Pais, and N. Palsetia. 2017. Black-box detection of XQuery injection and parameter tampering vulnerabilities in web applications. *Int. J. Inf. Secur*. 17, 1 (Feb. 2018), 105–120.

[76] Lwin K. Shar and H. Beng Kuan Tan. 2012. Automated removal of cross site scripting vulnerabilities in web applications. *Inf. Softw. Technol.* 54, 5 (May 2012), 467–478.

[77] L. K. Shar and H. B. K. Tan. 2012. Auditing the XSS defence features implemented in web application programs. *IEEE Softw*. 6, 4 (Aug. 2012).

[78] Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. 2010. Candid. *ACM Trans. Inf. Syst. Secur*. 13, 2 (Feb. 2010).

[79] M. Martin and M. S. Lam. 2008. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *Proceedings of the 17th USENIX Security Symposium.* USENIX Association, 31–43.

[80] M. Alkhalaf, S. R. Choudhary, M. Fazzini, T. Bultan, A. Orso, and C. Kruegel. 2012. Viewpoints: Differential string analysis for discovering client- and server-side input validation inconsistencies. In *Proceedings of the International Symposium on Software Testing and Analysis.* 56–66.

[81] Q. Binbin, L. Beihai, J. Sheng, and Y. Chutian. 2013. Design of automatic vulnerability detection system for web application program. In *Proceedings of the IEEE 4th International Conference on Software Engineering and Service Science.* 89–92.

[82] Minh-Thai T. Trinh, D.-H. H. Chu, and J. Jaffar. 2014. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 1232–1243.

[83] Young-Su S. Jang and J.-Y. Y. Choi. 2014. Detecting SQL injection attacks using query result size. *Comput. Secur*. 44 (2014), 104–118.

[84] L. Lei, X. Jing, L. Minglei, and Y. Jufeng. 2013. A dynamic SQL injection vulnerability test case generation model based on the multiple phases detection approach. In *Proceedings of IEEE 37th Computer Software and Applications Conference.* 256–261.

[85] H. He, L. L. Chen, and W. P. Guo. 2017. Research on web application vulnerability scanning system based on fingerprint feature. In *Proceedings of International Conference on Mechanical, Electronic, Control and Automation Engineering*. 150–155.

[86] Van-Giap G. Le, H.-T. T. Nguyen, D.-N. N. Lu, and N.-H. T. Nguyen. 2016. A solution for automatically malicious web shell and web application vulnerability detection. In *Proceedings of the International Conference on Computational Collective Intelligence.* Springer, Cham, 367–378.

[87] Hossain Shahriar and H. Haddad. 2016. Object injection vulnerability discovery based on latent semantic indexing. In *Proceedings of the 31st ACM Symposium on Applied Computing*. 801–807.

[88] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. 2010. Toward automated detection of logic vulnerabilities in web applications. In *Proceedings of the 19th USENIX Conference on Security*.

[89] Giancarlo Pellegrino and D. Balzarotti. 2014. Toward black-box detection of logic flaws in web applications. In *Network and Distributed System Security Symposium*. SanDiego, CA, USA.

[90] D. Kavitha, S. Chandrasekaran, and S. K. Rani. 2016. HDTCV: Hybrid detection technique for clickjacking vulnerability. In *Advances in Intelligent Systems and Computing, Artificial Intelligence and Evolutionary Computations in Engineering Systems,* Vol. 394. Springer New Delhi, 607–620.

[91] Venkatramulu Sunkari and C. V. Guru Rao. 2014. Preventing input type validation vulnerabilities using network based intrusion detection systems. In *Proceedings of the International Conference on Contemporary Computing and Informatics (IC3I)*. 702–706.

[92] L. Lei, X. Jing, G. Chenkai, K. Jiehui, X. Sihan, and Z. Biao. 2016. Exposing SQL injection vulnerability through penetration test based on finite state machine. In *Proceedings of the 2nd IEEE International Conference on Computer and Communications (ICCC)*. 1171–1175.

[93] Lei Liu et al. 2016. An effective penetration test approach based on feature matrix for exposing SQL injection vulnerability. In *Proceedings of the IEEE 40th Computer Software and Applications Conference (COMPSAC)*. 123–132.

[94] Michelle E. Ruse and S. Basu. 2013. Detecting cross-site scripting vulnerability using concolic testing. In *Proceedings of the 10th International Conference on Information Technology: New Generations*. IEEE, 633–638.

[95] T. Scholte, W. Robertson, D. Balzarotti, and E. Kirda. 2012. Preventing input validation vulnerabilities in web applications through automated type analysis. In *Proceedings of the IEEE 36th Computer Software and Applications Conference*. 233–243.

[96] Inyong Lee, S. Jeong, S. Yeo, and J. Moon. 2012. A novel method for SQL injection attack detection based on removing SQL query attribute values. *Math. Comput. Modell*. 55, 1 (2012), 58–68.

[97] Corrado A. Vlsaggio and L. C. Blasio. 2010. Session management vulnerabilities in today's web. *IEEE Secur. Privacy Mag*. 8, 5 (2010), 48–56.

[98] Xiaowei Li, X. Si, and Y. Xue. 2014. Automated black-box detection of access control vulnerabilities in web applications. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. 49–60.

[99] Cheng Huang, J. Y. Liu, Y. Fang, and Z. Zuo. 2016. A study on web security incidents in China by analyzing vulnerability disclosure platforms. *Comput. Secur*. 58 (May 2016), 47–62.

[100] Nisal M. Vithanage and N. Jeyamohan. 2016. Webguardia—An integrated penetration testing system to detect web application vulnerabilities. In *Proceedings of the IEEE International Conference on Wireless Communications, Signal Processing and Networking (Wispnet)*. 221–227.

[101] K. H. Zhang, Z. Li, R. Wang, X. F. Wang, and S. Chen. 2010. SideBuster: Automated detection and quantification of side-channel leaks in web application development. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*. 595–606.

[102] Peter Chapman and D. Evans. 2011. Automated black-box detection of side-channel vulnerabilities in web applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*. 263–274.

[103] Vahid Garousi et al. 2013. A systematic mapping study of web application testing. *Inf. Softw. Technol*. 55, 8 (Aug. 2013), 1374–1396.

[104] Manar H. Alalfi, J. R. Cordy, and T. R. Dean. 2009. Modelling methods for web application verification and testing: State of the art. *Softw. Test. Verif. Reliab*. 19, 4 (2009), 265–296.

[105] Maliheh Monshizadeh, P. Naldurg, and V. N. Venkatakrishnan. 2014. MACE: Detecting privilege escalation vulnerabilities in web applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 690–701.

[106] Andrea Avancini and M. Ceccato. 2013. Comparison and integration of genetic algorithms and dynamic symbolic execution for security testing of cross-site scripting vulnerabilities. *Inf. Softw. Technol*. 55, 12 (2013), 2209–2222.

[107] Lwin K. Shar, H. Beng Kuan Tan, and L. C. Briand. 2013. Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE, 642–651.

[108] Adam Doupé, B. Boe, C. Kruegel, and G. Vigna. 2011. Fear the EAR. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*. 251–261.

[109] Prithvi Bisht, T. Hinrichs, N. Skrupsky, and V. N. Venkatakrishnan. 2011. WAPTEC: Whitebox analysis of web applications for parameter tampering exploit construction. In *Proceedings of the 18th ACM Conference on Computer & Communications Security (CCS'11)*. 575–586.

[110] Nuno Antunes and M. Vieira. 2016. Designing vulnerability testing tools for web services: Approach, components, and tools. *Int. J. Inf. Secur*. 16, 4 (2016), 435–457.

[111] Xiaobing Guo, S. Jin, and Y. Zhang. 2015. XSS vulnerability detection using optimized attack vector repertory. In *Proceedings of the International Conference on Cyber-enabled Distributed Computing and Knowledge Discovery*. 29–36.

[112] Muhammmad S. Aliero, and I. Ghani. 2015. A component based SQL injection vulnerability detection tool. In *Proceedings of the 9th Malaysian Software Engineering Conference (MySEC)*. 224–22.

[113] Z. Djuric. 2013. A black-box testing tool for detecting SQL injection vulnerabilities. In *Proceedings of the 2nd International Conference on Informatics & Applications (ICIA)*. 216–221.

[114] A. K. Singh and S. Roy. 2012. A network based vulnerability scanner for detecting SQLI attacks in web applications. In *Proceedings of the 1st International Conference on Recent Advances in Information Technology (RAIT)*. 585–590.

[115] V. Shanmughaneethi, R. Y. Pravin, C. E. Shyni, and S. Swamynathan. 2011. *SQLIVD—AOP: Preventing SQL injection vulnerabilities using aspect oriented Programming through web services.* High-perform. Archit. Grid Comput. 169 (2011), 327–337.

[116] H. Y. Wu, G. Z. Gao, and C. Y. Miao. 2011. Test SQL injection vulnerabilities in web applications based on structure matching. In *Proceedings of the International Conference on Computer Science and Network Technology*. 935–938.

[117] L. Zhang, Q. Gu, S. Peng, X. Chen, H. Zhao, and D. Chen. 2010. D-WAV: A web application vulnerabilities detection tool using characteristics of web forms. In *Proceedings of the 5th International Conference on Software Engineering Advances*. 501–507.

[118] Nuo Li, T. Xie, M. Jin, and C. Liu. 2010. Perturbation-based user-input-validation testing of web applications. *J. Syst. Softw*. 83, 11 (2010), 2263–2274.

[119] Jan-Min M. Chen and C.-L. L. Wu. 2010. An automated vulnerability scanner for injection attack based on injection point. In *Proceedings of the International Computer Symposium (ICS'10)*. 113–118.

[120] M. Balduzzi, C. Gimenez, D. Balzarotti, and E. Kirda. 2011. Automated discovery of parameter pollution vulnerabilities in web applications. In *Proceedings of the NDSS Symposium.*

[121] Prithvi Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan. 2010. NoTamper: Automatic black-box detection of parameter tampering opportunities in web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*. 607–618.

[122] Davide Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. 2008. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*. 387–401.

[123] Abdalla W. Marashdih and Z. F. Zaaba. 2017. Detection and removing cross site scripting vulnerability in PHP web application. In *Proceedings of the International Conference on Promising Electronic Technologies (ICPET)*. IEEE, 26–31.

[124] W. E. Wong, V. Debroy, and B. Choi. 2010. A family of code coverage-based heuristics for effective fault localization. *J. Syst. Softw*. 83, 2 (2010), 188–208.

[125] Nuno Antunes and M. Vieira. 2011. Enhancing penetration testing with attack signatures and interface monitoring for the detection of injection vulnerabilities in web services. In *Proceedings of the IEEE International Conference on Services Computing*. 104–111.

[126] Yunhui Zheng, X. Zhang, and V. Ganesh. 2013. Z3-Str: A Z3-based string solver for web application analysis. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*. 114–124.

[127] Yunhui Zheng, V. Ganesh, S. Subramanian, et al. 2015. Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In *Proceedings of the International Conference on Computer-aided Verification* Springer, Cham, 235–254.

[128] Parosh A. Abdulla, M. F. Atig, Y.-F. F. Chen, et al. 2015. Norn: An SMT solver for string constraints. In *Proceedings of the International Conference on Computer-aided Verification*. Springer, Cham, 462–469.

[129] S. Gupta and B. B. Gupta. 2018. RAJIVE: Restricting the abuse of JavaScript injection vulnerabilities on cloud data centre by sensing the violation in expected workflow of web applications. *Int. J. Innov. Comput. Appl*. 9, (2018), 13–36.

[130] Hira Asghar, Z. Anwar, and K. Latif. 2016. A deliberately insecure RDF-based semantic web application framework for teaching sparql/sparul injection attacks and defense mechanisms. *Comput. Secur*. 58 (2016), 63–82.

[131] Moataz A. Ahmed and F. Ali. 2016. Multiple-path testing for cross site scripting using genetic algorithms. *J. Syst. Archit*. 64 (2016), 50–62.

[132] Nency Patel and N. Shekokar. 2015. Implementation of pattern matching algorithm to defend sqlia. *Procedia Comput. Sci.* 45 (2015), 453–459.

[133] Caitlin Sadowski et al. 2015. Tricorder: Building a program analysis ecosystem. In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*. 598–608.

[134] Brittany Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. 672–681.

[135] S. Ali, S. K. Shahzad, and H. Javed. 2009. SQLIPA: An authentication mechanism against SQL injection. *Eur. J. Sci. Res*. 38 (2009), 604–611.

[136] Anyi Liu, Y. Yuan, D. Wijesekera, and A. Stavrou. 2009. SQLProb. In *Proceedings of the ACM Symposium on Applied Computing*. 2054–2061.

[137] M. Junjin. 2009. An approach for SQL injection vulnerability detection. In *Proceedings of the 6th International Conference on Information Technology: New Generations*. 1411–1414.

[138] Philipp Zech, M. Felderer, and R. Breu. 2017. Knowledge-based security testing of web applications by logic programming. *Int. J. Softw. Tools Technol. Trans.* 21, 2 (2019), 221–246.

[139] Muhammad N. Khalid, H. Farooq, M. Iqbal, M. T. Alam, and K. Rasheed. 2019. Predicting web vulnerabilities in web applications based on machine learning. *Commun. Comput. Inf. Sci.* 932 (2019), 473–484.

[140] Iberia Medeiros, M. Beatriz, N. Neves, and M. Correia. 2019. SEPTIC: Detecting injection attacks and vulnerabilities inside the dbms. *IEEE Trans. Reliab.* 68, 3 (2019), 1168–1188.

[141] D. Ying, Z. Yuqing, M. Hua, W. Qianru, L. Qixu, W. Kai, and W. Wenjie. 2018. An adaptive system for detecting malicious queries in web attacks. *Sci. China Inf. Sci.* 61, 3 (2018).

[142] Julian Thome, L. K. Shar, D. Bianculli, and L. Briand. 2018. An integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving. *IEEE Trans. Softw, Eng*. 46, 2 (2018), 163–195.

[143] Shashank Gupta and B. B. Gupta. 2018. XSS-secure as a service for the platforms of online social network-based multimedia web applications in cloud. *Multimedia Tools Applic.* 77 (2018), 4829–4861.

[144] Vaibhav Patil, P. Thakkar, C. Shah, T. Bhat, and S. P. Godse. 2018. Detection and prevention of phishing websites using machine learning approach. In *Proceedings of the 4th International Conference on Computing Communication Control and Automation (ICCUBEA).* 1–5.

[145] Aditya Kurniawan, B. S. Abbas, A. Trisetyarso, and S. M. Isa. 2018. Static taint analysis traversal with object oriented component for web file injection vulnerability pattern detection. *Procedia Comput. Sci.* 135 (2018), 596–605.

[146] Mukesh K. Gupta, M. C. Govil, and G. Singh. 2018. Text-mining and pattern-matching based prediction models for detecting vulnerable files in web applications. *Journal of Web Engineering*. 17 1&2 (2018), 28–44.

[147] S. Anil, S. G. Manoj, L. Vijay, and C. Mauro. 2019. You click, I steal: Analyzing and detecting click hijacking attacks in web pages. *Int. J. Inf. Secur*. 18 (2019), 481–504.

[148] P. Li, L. Liu, J. Xu, H. Yang, L. Yuan, C. Guo, and X. Ji. 2017. Application of hidden Markov model in SQL injection detection. In *Proceedings of the IEEE 41st Computer Software and Applications Conference (COMPSAC)*. 578–583.

[149] Debabrata Kar, S. Panigrahi, and Sundararajan Srikanth. 2016. SQLiDDS: SQL injection detection using document similarity measure. *J. Comput. Secur*. 24, 4 (2016), 507–539.

[150] Giovanni Agosta, A. Barenghi, A. Parata, and G. Pelosi. 2012. Automated security analysis of dynamic web applications through symbolic code execution. In *Proceedings of the 9th International Conference on Information Technology: New Generations*. 189–194.

[151] Y. Zhong, H. Asakura, H. Takakura, and Y. Oshima. 2015. Detecting malicious inputs of web application parameters using character class sequences. In *Proceedings of the IEEE 39th Computer Software and Applications Conference.* 525–532.

[152] Hossain Shahriar, V. K. Devendran, and H. Haddad. 2013. Proclick. In *Proceedings of the 6th International Conference on Security of Information and Networks*. 144–151.

[153] M. Ceccato, C. D. Nguyen, D. Appelt, and L. C. Briand. 2016. SOFIA: An automated security oracle for black-box testing of SQL-injection vulnerabilities. In *Proceedings of the 6th International Conference on Security of Information and Networks*. 167–177.

[154] Shashank Gupta and B. B. Gupta. 2015. XSS-safe: A server-side approach to detect and mitigate cross-site scripting (XSS) attacks in JavaScript code. *Arab J. Sci. Eng*. 41, 3 (2015), 897–920.

[155] Z. Long. 2019. ART4SQLi: The art of SQL injection vulnerability discovery. *IEEE Trans. Reliab.* 68 (2019), 1470–1489.

[156] T. Hall, S. Beecham, D. Bowes, et al. 2012. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans. Softw. Eng*. 38, 6 (2011), 1276–1304.

[157] S. Bertrand and E. Fong. 2016. Large scale generation of complex and faulty PHP test cases. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation.* 409–415.

[158] Fang Yu, M. Alkhalaf, and T. Bultan. 2010. Stranger: An automata-based string analysis tool for PHP. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST).* 154–157.

[159] Takaaki Tateishi, M. Pistoia, and O. Tripp. 2013. Path- and index-sensitive string analysis based on monadic second-order logic. *ACM Trans. Softw. Eng. Methodol.* 22, 4 (2013), 1–33.

[160] NVD. 2019. The U.S. government repository of standards based vulnerability management data represented using the Security Content Automation Protocol (SCAP). Retrieved from https://nvd.nist.gov/.

[161] Bugzilla. 2019. A software to manage software development. Retrieved from https://www.bugzilla.org/.

[162] NIST. 2019. National Institute of Standards and Technology. Retrieved from http://www.nist.gov/.

[163] S. B. Psiinon. 2020. Bodgeit. The BodgeIt Store. Retrieved on 17 February, 2020 from https://github.com/psiinon/bodgeit.

[164] Kanakiya P. 2021. Openmrs-module-legacyui. OpenMRS Platform. Retrieved on 30 June, 2021 from https://github.com/openmrs/openmrs-module-legacyui/blob/master/omod/src/main/webapp/login.jsp.

[165] Regain. 2019. A search engine. Retrieved from http://regain.sourceforge.net/.

[166] HTTP dataset CSIC 2010. 2021. A testbed. Retrieved on 30 June, 2021 from https://www.kaggle.com/ispangler/csic-2010-web-application-attacks.
[167] B. Stivalet. Suite-9408 PHP source code. 2019. A test suite. Retrieved on 12 August, 2019 from https://github.com/stivalet/PHP-Vulnerability-test-suite.
[168] IBM Security AppScan. 2021. An enterprise scanner Retrieved from https://www.ibm.com/common/ssi/ShowDoc.wss?docURL=/common/ssi/rep_sm/9/897/ENUS5724-T59/index.html.
[169] NIST. 2019. Juliet Test Suite. Retrieved from https://samate.nist.gov/SRD/testsuite.php.
[170] M. Gegick and S. Barnum. 2013. Securing the Weakest Link Retrieved from https://us-cert.cisa.gov/bsi/articles/knowledge/principles/securing-the-weakest-link.
[171] Tore Dybå and T. Dingsøyr. 2008. Empirical studies of agile software development: A systematic review. *Information and Software Technology* 50, 9–10 (2008), 833–859.
[172] Software development blogs. 2020. Top 10 Programming Languages for Web Development in 2020. Retrieved on 12 August, 2020 from https://intersog.com/blog/top-10-programming-languages-for-web-development-in-2020/.
[173] Javinpaul. 2021. Top 5 programming languages for web development in 2021 Retrieved on 30 June, 2021 from https://medium.com/javarevisited/top-5-programming-languages-for-web-development-in-2021-f6fd4f564eb6.

Received March 2020; revised June 2021; accepted July 2021