Covariate Software Vulnerability Discovery Model to Support Cybersecurity Test & Evaluation (Practical Experience Report)

Julia Sorrentino, Priscila Silva, Gaspard Baye, Gokhan Kul, Lance Fiondella
University of Massachusetts Dartmouth
North Dartmouth, United States
{*jsorrentino, psilva4, bgaspard, gkul, lfiondella*}@umassd.edu

2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE) | 978-1-6654-5132-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/ISSRE55969.2022.00025

***Abstract*—Vulnerability discovery models (VDM) have been proposed as an application of software reliability growth models (SRGM) to software security related defects. VDM model the number of vulnerabilities discovered as a function of testing time, enabling quantitative measures of security. Despite their obvious utility, past VDM have been limited to parametric forms that do not consider the multiple activities software testers undertake in order to identify vulnerabilities. In contrast, covariate SRGM characterize the software defect discovery process in terms of one or more test activities. However, data sets documenting multiple security testing activities suitable for application of covariate models are not readily available in the open literature.**

To demonstrate the applicability of covariate SRGM to vulnerability discovery, this research identified a web application to target as well as multiple tools and techniques to test for vulnerabilities. The time dedicated to each test activity and the corresponding number of unique vulnerabilities discovered were documented and prepared in a format suitable for application of covariate SRGM. Analysis and prediction were then performed and compared with a flexible VDM without covariates, namely the Alhazmi-Malaiya Logistic Model (AML). Our results indicate that covariate VDM significantly outperformed the AML model on predictive and information theoretic measures of goodness of fit, suggesting that covariate VDM are a suitable and effective method to predict the impact of applying specific vulnerability discovery tools and techniques.

***Index Terms*—Software reliability, cybersecurity, penetration testing, vulnerability discovery, covariate model**

I. Introduction

Software is highly versatile, providing a wide variety of functionality in applications and software-enabled systems. However, the potentially negative impacts of software exploitation through latent vulnerabilities cast a dark shadow over the otherwise promising nature of software, especially those intended to monitor and control infrastructure, defend the homeland, and support national security. Significant effort has been dedicated to vulnerability taxonomies [1], [2] as well as tools [3], [4] and techniques [5]–[7]. Some attention has been allocated to models [8] that assess vulnerability discovery-focused testing. However, these models only consider the amount of time spent testing, preventing more detailed guidance on the relative effectiveness of alternative vulnerability testing tools and techniques. In the absence of models that link specific test activities and tools to vulnerability discovery, organizations will struggle to understand which combination of methods may be most effective for the system under test and how to conduct detailed test planning.

Relevant past research may be categorized into three primary categories, including (i) software vulnerability taxonomies, (ii) testing techniques and tools, and (iii) vulnerability discovery modeling. Taxonomies hierarchically organize various classes of common weaknesses and attacks, while vulnerability techniques and tools respectively define the abstract concepts and concrete implementations of algorithmic approaches to discover vulnerabilities for mitigation. Vulnerability discovery models quantify trends in the rate of vulnerabilities discovered during testing, estimate the number of vulnerabilities remaining, and metrics such as mean time between discovery, which may be regarded as the security analog of the traditional reliability, metric mean time to failure. In the absence of more detailed methods to complement processes such as the Risk Management Framework (RMF) [9], organizations developing or acquiring software will struggle to perform their diverse duties with the degree of diligence to which they aspire.

This paper presents a study, comparing a software vulnerability defect model incorporating covariates [10] with the AML model [8], which is one of the most flexible VDM without covariates. Since covariate data on the amount of time spent applying different tools and techniques in each interval was not readily available, a target application was identified and subjected to multiple vulnerability discovery tools and techniques. To support equitable comparison, grouped data expressions of the AML model as a nonhomogeneous Poisson process (NHPP) are derived and two methods to express the length of time intervals considered, including the total time required to perform the work in each interval and the sum of the times spent executing tools in each interval. Our results indicate that the software VDM incorporating covariates (i) more accurately tracks and predicts the number of vulnerabilities discovered in future intervals as a function of penetration testing activities performed and (ii) achieves better goodness

This research was supported by the Homeland Security Community of Best Practices (HS CoBP) through the U.S. Department of the Air Force through under award number SCR1158132. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the U.S. Department of Homeland Security or U.S. Department of the Air Force.

2332-6549/22/$31.00 ©2022 IEEE
DOI 10.1109/ISSRE55969.2022.00025

of fit, despite the fact that the information theoretic measures penalize the covariate models for their additional parameters. The AML model, in contrast, like other parametric VDMs that do not incorporate covariates, can only capture the primary trends embedded in their model forms. Thus, software VDMs incorporating covariates offer a more detailed methodology to assess the effectiveness of alternative tools and techniques to discover vulnerabilities. The software VDM incorporating covariates can, therefore, be used to allocate vulnerability testing effort and guide process improvement.

The remainder of the paper is organized as follows: Section II describes related research in the areas of vulnerability taxonomies and discovery models. Section III describes the process by which the target system was selected. Section IV provides an overview of the vulnerability categorization employed. Section V explains the methodology, including test environment and process by which the vulnerabilities discovered were mapped to covariate data. Section VI presents vulnerability discovery models with and without covariates. Section VII reviews quantitative goodness of fit measures employed. Section VIII present a comparative analysis of the alternative vulnerability discovery models, while Section IX summarizes the study and provides suggestions for future research.

II. Related Research

This section reviews related research in the areas of software vulnerability taxonomies and vulnerability discovery models. For a review of software vulnerability techniques, the reader is referred to [7].

A. Software vulnerability taxonomies

Efforts to identify, define, and catalog publicly disclosed cybersecurity vulnerabilities include the MITRE Common Vulnerabilities and Exposures (CVE) Program [11]. Prominent vulnerability lists include the MITRE Common Weakness Enumeration (CWE) [1] and the Open Web Application Security Project (OWASP) Top 10 [2]. The MITRE CWE is a community-developed list of software and hardware weakness types intended to provide a variety of services to security teams, including as a baseline for weakness identification, mitigation, and prevention efforts. The OWASP Top 10 represent major categories that can be further decomposed into CWEs, identifying a broader security control area for vulnerability types. The OWASP Top 10 covers approximately 200 CWE, and a single OWASP category can contain as many as three dozen CWEs. There are over 600 categories of CWEs, meaning that some risks are not covered by the OWASP Top 10, since CWEs include both software and hardware issues. The benefit of mapping discovered vulnerabilities to CWEs is that the MITRE Common Attack Pattern Enumeration and Classification (CAPEC) [5] relates CWEs to the most common exploitation methods, which can help developers better secure their software system by knowing what types of attacks the system is prone to. To enable government decision makers to make effective software assurance and software supply chain risk management decisions [7], the Institute for Defense Analyses combined several sources, including the Center for Assured Software Static Analysis Tool Study Methodology [12], National Vulnerability Database (NVD) [13], CWE/SANS Top 25 [14], the Program Protection Plan (PPP)/Defense Acquisition Guidebook [15], OWASP [16] and the Web Application Security Consortium [17] into a single categorization listing. Despite these efforts, it is always possible that additional unknown risks remain.

B. Vulnerability discovery modeling

Alhazmi and Malaiya [8] compared the Anderson Thermodynamic Model [18], [19], the Alhazmi-Malaiya Logistic Model [20], Rescola's Linear and Exponential Models [21], and the Logarithmic Poisson Model [22] on vulnerabilities in three major operating systems, including Windows 95, Red Hat Linux 6.2, and Windows XP, assessing goodness of fit with the chi-squared statistic, residual sum of squares, and Akaike information criterion. Alhazmi et al. [23] subsequently applied the Alhazmi-Malaiya Logistic Model to a larger number of Windows operating systems and versions of Red Hat Linux. Kim et al. [24] generalized the Alhazmi-Malaiya Logistic Model to a mixture of two successive versions of a software and applied this mixture model to the Apache HTTP Web server and Mysql DBMS, examining the relationship between software evolution and vulnerability discovery. Condon et al. [25] used the Laplace test [26] to guide NHPP model selection from the Goel-Okumoto [27], Yamada S-Shaped [28], Duane [29], and Khoshgoftaar K–stage Erlangian [30] models. Okamura et al. [31] proposed an optimal security patch release timing model under the assumption of a NHPP vulnerability discovery process, considering costs such as those associated with development and distribution of patches as well as the burden of unpatched vulnerabilities. Additional models applied include the Weibull [32] and folded [33] distributions as well as an effort-based Weibull model [34] and an effort-based model [35] considering numbers of vulnerabilities discovered by reporters and number of reporters per unit time. Nguyen and Massacci [36] presented a methodology to determine if a model can adequately characterize the vulnerability discovery process in terms of goodness of fit and predictive accuracy, applying several of the models previously noted to 30 major releases of four popular Internet browsers, including Explorer, Firefox, Chrome, and Safari. Okamura et al. [37] proposed a vulnerability lifecycle model possessing multiple stages from discovery to resolution, characterizing the time to distribution of a patch by the phase-type (PH) distribution. Kansal et al. [38] developed a vulnerability discovery model considering the operational effort and coverage rates, respectively defined as the proportion of manpower required to discover vulnerabilities and the proportion of software covered by the effort to discover vulnerabilities.

Unlike previous studies, this paper is the first to (i) create a data set suitable for application of covariate SRGM to the vulnerability discovery problem, (ii) carefully explain the

mapping process from penetration testing to a format suitable for model application in order to encourage additional studies, and (iii) illustrate the predictive performance of covariate SRGM on the vulnerability discovery problem.

III. Selection of target application

Prior to performing our experiments, it was necessary to identify a specific application to subject to penetration testing [39]. Three classes of applications were considered, including web, mobile, and desktop applications. The OWASP Top 10 defines unique top ten security risks for the three classes of applications considered, including web [2], mobile [40], and desktop [41] applications. A web application runs on a web server and is accessed by users through a web browser via a network connection, and can therefore be accessed by any user with a valid network connection on any device. In contrast, a mobile application is a program designed to run on a mobile device and can only be accessed via such devices, while a desktop application can only be run by a computer. US Government organizations such as the Department of Defense also distinguish between Major Automated Information Systems (MAIS) and Weapon Systems, many of which are software intensive. Similarly, NASA is especially interested in the reliability and security of their core Flight System (cFS) flight software [42]. However, the goal of this study was to demonstrate how to map the results of security testing in order to conduct quantitative assessments with covariate software vulnerability discovery models [43] in any domain, so that readers can understand how to perform a similar mapping in their particular domain of interest.

Among the classes of systems described above, a web application was chosen as the target for testing, due to their complexity, including the high degree of connectedness to internet components that provide functionality and their corresponding vulnerabilities [44]. Moreover, use of web applications is extremely widespread by people all over the world to obtain information, perform financial transactions, and communicate with one another. For example, an attacker seeking financial gain may target credit card information, once they have gained initial access to a network. This is especially attractive to attackers, since financial losses from the 58 largest web application security incidents reported in the last five years [45] exceeded $7.6 billion USD. Another factor that makes web applications a preferred target of attackers is the ease of use of tools that expose and automatically exploit vulnerabilities. Vulnerability analysis tools take advantage of application specific code, which is often developed under strict time constraints by programmers possessing little to no security training. In addition to these factors, web applications are also vulnerable to traditional desktop vulnerabilities that do not contain internet components. Therefore, the relative ease of access, availability of tools for exploitation, wide array of vulnerability types and potential benefit of demonstrating the proposed approach motivated our choice of conducting this study in the context of a web application.

Four open source web applications were investigated as possible targets, including VulnLab [46], Damn Vulnerable Web Application (DVWA) [47], Xtreme Vulnerable Web Application (XVWA) [48], and OWASP Juice Shop [49]. The purpose of each of these vulnerable web applications is to assist individuals possessing a range of backgrounds to learn skills and test tools in a legal environment. VulnLab is a web vulnerability lab project composed of labs divided into specific categories of vulnerabilities including SQL injection, cross site scripting, command injection, insecure direct object reference, cross site request forgery, XML external entity, insecure deserialization, file upload, file inclusion, and broken authentication. DVWA is a PHP/MySQL web application for individuals to practice discovering and exploiting some of the most common web vulnerabilities at varying difficulty levels. XVWA is another web application written in PHP/MySQL, which was poorly coded on purpose for the sake of learning. Similar to VulnLab, activities within DVWA and XVWA are categorized and separated by vulnerability type. OWASP Juice Shop emulates a modern day web application for an online Juice Shop marketplace with complex functionality, containing a wide variety of vulnerabilities. Ultimately, OWASP Juice Shop was chosen because it was determined to bear the greatest resemblance to a real world web application. In contrast, the alternative choices were relatively narrowly focused assignments that do not require any form of web exploration, explicitly pointing users to the vulnerabilities that could be exploited.

IV. Open Web Application Security Project (OWASP)

The OWASP [16] is a nonprofit foundation established in 2001 that works to improve the security of software. The OWASP Foundation is composed of 200 chapters worldwide with tens of thousands of members. This community leads open source software projects and holds educational and training conferences. OWASP projects, tools, and documents, are free and open to anyone interested in improving application security, or supporting organizations develop, acquire, operate, and maintain trusted applications.

OWASP is perhaps most well known for developing and maintaining security standards and processes for web applications as well as providing technical information about key security risks and countermeasures. One of the standards that the foundation developed is the OWASP Top 10 [2], a list of the top 10 most critical security risk categories related to web applications. There are a number of reasons that developers and web application security should classify vulnerabilities based on this document. The primary justification is that there is broad consensus among researchers [50] that the OWASP Top 10 constitute the main security risks to web applications. The standard is based on contributor data created by testing web applications for vulnerabilities and recording how many web applications contained specific vulnerabilities and is therefore community-driven. Additional reasons for using the

OWASP Top 10 list are that it can be used at each stage of the software development life cycle, justify security activities to management, and show progress over time toward industry standard security and compliance, which aligns well with the trend analysis enabled by software vulnerability discovery models. Thus, while the OWASP is not exhaustive, focusing on these Top 10 risks enables developers to mitigate the most common types of security risks found in web applications.

The following subsections provide a brief description of each of the OWASP Top 10 Web Application vulnerabilities.
A01: Broken Access Control Access control [51] regulates how the data and resources are accessed by users based on the permissions specified by administrators. Broken access control occurs when a user acts outside of these specified limits. When an attacker bypasses these control measures, they may be able to view sensitive data or perform tasks as a privileged user.
A02: Cryptographic Failures A web application accesses, sends and receives a variety of types of data, including information governed by privacy laws and requiring additional protection. Examples include passwords, credit card numbers, and personal and health information. This category of vulnerabilities [52] refers to situations when sensitive data is exposed as a direct result of improper implementation of cryptography.
A03: Injection An injection attack [53] occurs when an attacker provides malicious input to a web application and changes the operation of the application, forcing it to execute specific commands. There are several factors that may make an application vulnerable to attack such as not validating, filtering, or sanitizing user-supplied data, using dynamic queries, and several others. Common types of injection include: SQL [54], NoSQL [55], and operating system commands [56].
A04: Insecure Design Secure system and software development requires considering possible threat models [57], implementing secure design principles and patterns [58], and adopting security best practices [59]. In contrast, insecure implementation may be regarded as the source of the other categories in the Top 10, possessing different root causes and remedies.
A05: Security Misconfiguration Vulnerabilities may manifest as a direct result of security controls not being properly configured. There are many places where misconfiguration may occur in a web application [60] such as enabling unnecessary features, not changing default passwords, providing users with unnecessarily detailed error messages, and failure to update security features in a timely manner. Security misconfiguration vulnerabilities are much more likely to occur when developers do not implement a repeatable security configuration process.
A06: Vulnerable and Outdated Components Software usually includes libraries required to accomplish certain tasks. These libraries may be kept up-to-date with dependency managers [61] or manually. However, when one of these packages become vulnerable or left outdated [62], they become threats to the security of the software.
A07: Identification and Authentication Failures Authentication [63] is the process of verifying the identity of users prior to granting them access to the web application. Failure of these mechanisms could allow an attacker to gain access to another user's account [64], potentially giving them access to sensitive data. The most common weaknesses that lead to identification and authentication failures are related to how the web application handles passwords [65]. Hence, applications are more vulnerable, if an application allows an arbitrary number of password attempts, default or weak passwords, implements a weak credential recovery process, stores passwords in plaintext, or serves URLs that expose the session identifier.
A08: Software and Data Integrity Failures Integrity in cybersecurity [66] refers to the techniques and tools employed to safeguard against improper modification or destruction of information. Application integrity can be violated if plugins, libraries, or modules from untrusted sources are used or if a structure where data has been encoded or serialized into can be viewed and edited.
A09: Security Logging and Monitoring Failures In order for security professionals to be able to determine the threat level and consequences of an attacker's actions, they must be able to trace the movement of an attacker within the web application [67]. Therefore, web applications should log every action performed by all users in order to enable recovery and audits [68], while facilitating monitoring and tracing in case of malicious activities. A proper log should include: HTTP status codes, time stamps, usernames, API endpoints/page location, and IP addresses. Insufficient logging and monitoring of a system means that suspicious activity will go undetected in log files. Failures can occur if logs do not record events for auditing, generate warnings or errors containing sufficient information, actively monitor for suspicious activity, or receive alerts to trigger dynamic application security testing tools.
A10: Server-Side Request Forgery A server-side request forgery vulnerability [69] can occur when a web application retrieves a remote resource without first validating a user-supplied URL. This lack of validation allows an attacker to craft and send HTTP requests to a domain of their choosing, including domains protected by a firewall or VPN.

V. Methodology

This section provides an overview of the methodology, including the tools and techniques selected to identify vulnerabilities as well as the data collected.

Figure 1 summarizes the environment established to host and attack a web application. Figure 1 indicates that the testing environment consists of a vulnerable web application, OWASP Juice Shop, hosted on a Docker container running on port 3000 within a VirtualBox. All penetration testing activities were performed by the attacker from the host machine located within the same network running Kali Linux [70], which was chosen as the operating system because it combines a standard Debian base with a wide variety of information security tools. In this scenario, the attacker communicates with the web application via HTTP. Specifically, the web application can be accessed through the Mozilla Firefox browser via the

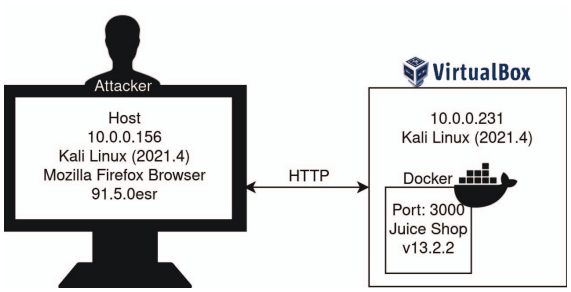

Fig. 1. Environmental Setup

IP address of the VirtualBox and the port that the docker container is running on.

Table I shows the tools used to identify each category of vulnerability defined by the OWASP Top 10 as well as the testing modes enabled, including A (automated) or M (manual) to indicate if the tool can find the vulnerability type automatically or if it requires manual user enumeration or review, respectively.

TABLE I
SELECTED TOOLS AND APPLICATION TO OWASP TOP 10

OWASP Top 10	Tools		OWASP Top 10	Tools	
	ZAP	*Burp Suite*		*ZAP*	*Burp Suite*
A01	A/M	A/M	A06	A/M	M
A02	A	M	A07	A/M	A/M
A03	A/M	M	A08	A	M
A04	A	M	A09	A/M	M
A05	A/M	M	A10	A	M

In addition to the use of OWASP Zed Attack Proxy (ZAP) and Burp Suite, it is also possible to directly inspect and test the code for any of the OWASP Top 10 with manual techniques directly within the web application. Thus, while the automated techniques are primarily intended to improve the speed and comprehensiveness with which search is performed, it is also desirable to quantify the relative effectiveness of alternative tools with respect to these vulnerability finding capabilities and their thoroughness. Hence, automation does not necessarily equate to a superior approach, if the automated techniques are less comprehensive and fail to find vulnerabilities in comparison to skilled manual inspection. Moreover, inexperienced users may become excessively dependent on automated tools, leading to overconfidence and reliance on automated techniques that are inadequate. For these reasons, covariate VDM are a potentially valuable complement to quantify the relative effectiveness of alternative techniques and tools.

A. Tools

The following subsections provide a brief description of the tools selected to find vulnerabilities, including OWASP ZAP [71], Burp Suite [72], and manual inspection. Tool workflows are also described to support reproducibility and clarify the data collection process.

1) OWASP ZAP: The OWASP ZAP is a free and open source vulnerability scanning tool developed and maintained by the OWASP community. It is an integrated penetration testing tool that can perform automated and manual scans to identify vulnerabilities in web applications. ZAP's main feature is automated web application scanning, which scans a web application passively and actively, builds a sitemap with spidering, and discovers vulnerabilities which trigger alerts. Spidering is an automated recursive process in which a web crawler identifies hyperlinks on a web page and continues to visit them as long as new resources are found resulting in the construction of a graph data structure mapping of the website.

ZAP also implements an intercepting proxy that can be used to manually perform tests on specific webpages. The advantage of this proxy is that the tester can select specific GET requests from the site map and open/resend them with the request editor without needing to change windows to forward individual requests, until they find the one they wish to modify, simplifying the vulnerability inspection process. The ZAP Intruder functionality enables unthrottled attacks that can tax the attacking machine and web server to their maximum capacity, assuming there are no safeguards implemented like a limit on the number of attempts. This enables brute force attacks which employ trial and error for a variety of end goals such as to crack passwords, login credentials, and encryption keys or browse for undiscovered directories.

ZAP version 2.11.1 is pre-installed on Kali Linux version 2021.4. On startup, ZAP asks the user to specify if they desire to use session persistence, which stores a user's session locally so that it can be accessed at a later date. Benefits of saving a session include not having to spend time re-running a scan. Instead, the session can be opened enabling the user to pick up where they left off and build upon their existing work. The primary options are to perform an automated scan or explore manually. To perform an automated scan, the user must supply the URL to attack and select between a traditional spider, an Ajax spider, or both to crawl the web pages. The traditional spider begins with a list of URLs to visit and then identifies and visits all the hyperlinks in additional pages. Given a browser type parameter, the Ajax spider performs a more in depth search of pages that rely on JavaScript, discovering additional pages that are accessed in the background or generated on the client side. In contrast, the manual explore option takes a URL parameter and launches a browser pre-configured to proxy through ZAP.

Vulnerabilities were identified primarily via the sitemap and alert interfaces. The structure of the website is shown as a list of GET requests and folders which contain additional requests from sub-domains. Selecting a specific request displays it in the work space where the user can view the response, and file contents, if applicable. Right-clicking displays additional functionality such as opening the specific request in the request editor or opening the URL in the proxy browser. The alert interface is a tab in the information window that displays any flags that were raised during the automated scan. Selecting a specific alert shows additional information such as the affected

URL, evidence, risk and confidence levels. Since these are automated features, these results require further investigation by the user to confirm the presence of an actual vulnerability. Manual review of all alerts and requests enabled vulnerabilities to be discovered and classified according to A01-A10 in the OWASP Top 10. Once the presence of specific vulnerabilities were confirmed, log files of the session were reviewed to identify the time taken by the scan to find the vulnerability.

2) Burp Suite: Burp Suite is not an open source tool, but provides a free Community Edition version. There are also Professional and Enterprise editions available for purchase, which come with additional features and tools within the application. The community edition may detect fewer vulnerabilities and require more effort. However, use of the Community Edition provided an additional vector of observations to apply the covariate software vulnerability discovery models, illustrating the suitability of the approach to objectively assess the relative effectiveness of alternative tools.

While Burp Suite Community offers fewer features relative to the Professional or Enterprise editions, it contains essential tools to perform manual security testing, including *Proxy*, *Repeater*, *Decoder*, *Sequencer*, *Comparer* and a demo version of *Intruder*.

Proxy is a central feature of Burp Suite, since it enables the interception and modification of HTTP requests/responses when interacting with the web application. This enables a user to gain insight into the behavior of the web application in order to infer potential vulnerabilities. From here, a request can be forwarded to one of the other tools which may require a HTTP request to perform additional inspection and manipulation of the requests.

Repeater allows a request to be modified and resent multiple times. This tool is useful for a variety of purposes such as changing parameter values for input-based vulnerabilities, issuing requests in a specific sequence to test for logic flaws, or modifying request headers to test for cryptographic flaws.

Decoder has the ability to decode intercepted information or encode a payload prior to sending it to the target. This function takes text or hex data and can decode or encode into plaintext, URL, HTML, Base64, ASCII hex, hex, octal, binary or gzip. This data can be used to decode recovered data or encode malicious payloads such as an SQL query string into a URL.

Sequencer is most commonly used when assessing the randomness of tokens or other important data that are intended to be unpredictable. This tool enables test of an application's session tokens or other important data items that are intended to be unpredictable, including anti-CSRF tokens or password reset tokens. If these values do not have an acceptable level of randomness they may introduce serious cryptographic or authentication failures as they can be easily forged by an attacker.

Comparer enables the comparison of two pieces of data at the word or byte level. This tool can be applied to find differences in a variety of fields such as failed login responses when looking for username enumeration conditions, comparing the site maps generated by different types of users or comparing responses of different injection conditions.

Intruder, which is severely rate-limited in the Community Edition, allows for an endpoint to be sprayed with requests, and can be used to perform a large range of tasks, but is most commonly used to enumerate identifiers, harvest useful data or fuzz for vulnerabilities. Depending on the position of the user configured payload parameters can be injected allowing for the brute force guessing of simple items such as directories, or credentials or more complex items such as blind SQL injection vulnerabilities.

Overall, the ability to capture and manipulate web requests prior to sending them to the target web server made Burp Suite an attractive tool to conduct manual web application testing.

Burp Suite version 11.0.14 is pre-installed on Kali Linux version 2021.4. Configuration files enable options related to the individual environment and user interface. The main functionality of Burp Suite is located on the Proxy tab, where the user is given the option to use the embedded Chromium browser or configure an external browser with a proxy such as FoxyProxy. Using Burp's embedded browser requires no additional configuration, whereas an external browser proxy requires additional steps to set up and configure with Burp Suite. As the user navigates the web application, the intercept button can be toggled on or off to intercept requests. The intercept may be turned on at any point, but the browser will not load properly until it receives a response. Intercepted requests can be forwarded to the destination server, dropped, or sent to tools within Burp Suite for additional inspection or manipulation. Burp Suite is a hands-on tool which requires user control of the actions to be performed. The user must manually toggle the intercept functionality and forward each individual request. Therefore, they must use their own judgment to determine if the request may have vulnerable elements that need further inspection. To document the time at which vulnerabilities were discovered, a manual timer was started at the time Burp Suite was executed and recorded when the vulnerability was uncovered.

3) Manual Inspection: The simplest form of penetration testing involves browsing the web application and looking for vulnerabilities. Not only is manual inspection a form of reconnaissance which allows the tester to gather information that may be helpful in the later stages of testing, it can also be used to uncover vulnerabilities without the use of additional vulnerability assessment tools. A tester can manually fuzz input fields to test for injection vulnerabilities, brute force directories to uncover sensitive information, and make educated guesses for a user's password or password reset security questions to gain unauthorized access to their accounts. In addition to being used independently, manual inspection can complement information collected with tools. A tester can use insecure cryptographic techniques to decode sensitive information or encode and forge information. The manual review of source code files can also uncover misconfiguration vulnerabilities. The time to vulnerability discovery was

collected by manually timing the reproduction of navigating to find a specific vulnerability within the web application.

B. Data Collection

Each testing session began with set up and configuration of one of the two penetration testing tools. Although setup varies by tool, a human tester was required for both ZAP and Burp Suite in order to review the results of automated tests, ensure an acceptable level of assurance, and remove false positives. The automated and manual functionality of ZAP produced a site map with items to be individually reviewed by the tester, whereas Burp Suite required the tester to manually explore the web application and review HTTP requests. Manual inspection complemented Burp Suite, since it was possible to interact with items within the application, view, and test before proxying specific functionality.

Table II summarizes the data collected during penetration testing activities, in a format suitable for input into a covariate vulnerability discovery tool. Each row corresponds to an interval ($\mathbf{t}$) in which approximately 5 hours were dedicated to investigating the web application, attempting to discover vulnerabilities, and confirming their presence. The vulnerability count ($\mathbf{k}$) indicates the number of vulnerabilities discovered in that interval. These vulnerabilities were also classified into categories according to the OWASP Top 10 in columns A01-A10. The last three columns document the time (in seconds) the three tools and test activities (covariates) were applied, including ZAP (Z), Burp Suite (B) and manual inspection (M).

To clarify the process of mapping test results to the format shown in Table II, we discuss the details of the vulnerabilities discovered in the first interval here. The techniques employed include an initial scan with the traditional spider in ZAP as well as proxying several of the main pages with Burp Suite to intercept and modify requested resources that the server sent to the client.

The automated scan performed by ZAP created a site map of Juice Shop, providing a view of requests and responses for every piece of the site discovered by the spider. Several of the responses directly exhibited vulnerabilities. For example, the response from GET http://10.0.0.231:3000/ftp/ notes that /ftp/ is a listing directory. Therefore, the attacker is able to gain access to a variety of files, including a folder containing quarantined malware. This was classified as *A01:Broken Access Control* and can be further classified as *CWE-548: Exposure of Information Through Directory Listing*, since it provides an attacker with the complete index of all resources located inside the directory. Next, the response from GET http://10.0.0.231:3000/ftp/acquisitions.md displays the contents of the markdown file. The document is explicitly described as confidential because it contains details related to the company's planned acquisitions over the next year. This vulnerability was classified as *A05: Security Misconfiguration* as well as *CWE-541: Inclusion of Sensitive Information in an Include File*.

Burp Suite required manual inspection of requests and responses. A temporary email was used to create an account and load the Juice Shop homepage with a variety of requests. Sending GET /api/Challenges/?name=Score%20Board to the Repeater enabled an attacker to modify the request to GET /api/Challenges and receive all the challenge information, including name, descriptions, difficulty level, and hints to find the security weaknesses within the web application, which was classified as *A01: Broken Access Control* as well as *CWE-922: Insecure Storage of Sensitive Information*, since this information should be read limited based on the challenges that have been completed or if the user requests a hint. Moreover, sending GET /rest/basket/1 to the Intruder allowed the identifier number to be targeted and changed to different integers. The responses of this brute force attack revealed that it was possible to access the contents of other user baskets. This vulnerability was classified as *A04: Insecure Design*, since the GET request contained the specific user basket identification number. This vulnerability was also classified as *CWE-598: Use of GET Request Method With Sensitive Query Strings* because an attacker could change the identification number included in the request and gain access to another user's information.

VI. Vulnerability Discovery Models with and without covariates

This section presents vulnerability detection models with and without covariates.

A. Covariate Software Vulnerability Detection Model based on the Discrete Cox Proportional Hazards Model

The discrete Cox proportional hazards NHPP SRGM [10] correlates m covariates to the number of events in each of n intervals. In the context of software vulnerability discovery, these covariates are the amounts of time dedicated to each of the distinct vulnerability testing activities performed with multiple tools or techniques. The matrix $\mathbf{x}_{n\times m}$ quantifies the amount of effort dedicated to each activity in each interval. For example, $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{im})$ denotes the amount of each activity $(1 \leq j \leq m)$ performed in the ith interval.

The mean value function (MVF) predicts the number of vulnerabilities discovered up to and including the n^{th} interval given covariates $\mathbf{x}$ according to

$$m(\mathbf{x}) = \omega \sum_{i=1}^{n} p_{i,\mathbf{x}_i} \tag{1}$$

where $\omega > 0$ denotes the number of vulnerabilities that would be discovered with infinite testing and

$$p_{i,\mathbf{x}_i} = \left(1 - (1 - h(i))^{g(\mathbf{x}_i;\boldsymbol{\beta})}\right) \prod_{k=1}^{i-1} (1 - h(k))^{g(\mathbf{x}_k;\boldsymbol{\beta})} \tag{2}$$

is the probability that a vulnerability is discovered in the ith interval, given that it was not discovered in the first $(i-1)$

TABLE II
DATA COLLECTED DURING PENETRATION TESTING ACTIVITIES

		OWASP Top 10										Tools		
t	**k**	***A01***	***A02***	***A03***	***A04***	***A05***	***A06***	***A07***	***A08***	***A09***	***A10***	**Z**	**B**	**M**
1	4	2	0	0	1	1	0	0	0	0	0	147	42	0
2	12	2	1	3	1	1	1	2	0	1	0	375	402	104
3	8	1	1	2	0	0	0	3	0	1	0	0	521	195
4	6	4	1	0	1	0	0	0	0	0	0	11	123	34
5	3	0	1	1	1	0	0	1	0	0	0	0	45	6
6	4	2	0	0	1	0	0	0	1	0	0	0	71	97
7	3	1	0	1	0	0	0	1	0	0	0	0	214	0
8	3	1	0	1	0	0	0	0	0	0	1	0	86	107

intervals, $h(\cdot)$ is the baseline hazard function, and $\boldsymbol{\beta}$ is the vector of m parameters contained within the Cox proportional hazards model

$$g(\mathbf{x}_i;\boldsymbol{\beta}) = \exp(\beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_m x_{im}) \quad (3)$$

B. Hazard functions

This section presents examples of hazard functions that can be incorporated into Equation (2). The following three were originally employed in the covariate software reliability model of [43].

1) Geometric (GM):

$$h(b) = b \quad (4)$$

where $b \in (0,1)$ is the probability of detecting a defect.

2) Negative binomial of order two (NB2):

$$h(i;b) = \frac{ib^2}{1+b(i-1)} \quad (5)$$

where $b \in (0,1)$ and 2 indicates the order.

3) Discrete Weibull of order two (DW2):

$$h(i;b) = 1 - b^{i^2-(i-1)^2} \quad (6)$$

Five additional hazard functions taken from the survey of [73] have also been applied, including the Type III discrete Weibull (DW3) [74], "S" distribution (S) [75], Truncated logistic (TL) [76], Increasing Failure Rate Salvia and Bollinger (IFRSB) [77], and IFR generalized Salvia and Bollinger (IFRGSB) [74].

To estimate the parameters of the DCPH model with covariates, the log-likelihood function [10] is

$$\begin{aligned} LL(\mathbf{x},\mathbf{k};\boldsymbol{\gamma},\boldsymbol{\beta},\omega) &= -\omega\sum_{i=1}^{n} p_{i,\mathbf{x}_i} + \sum_{i=1}^{n} y_i \ln(\omega) \quad (7)\\ &\quad + \sum_{i=1}^{n} y_i \ln(p_{i,\mathbf{x}_i}) - \sum_{i=1}^{n} \ln(y_i!) \end{aligned}$$

where $\mathbf{x} = \{Z, B, M\}$ are the covariates, $\mathbf{k}$ is the number of vulnerabilities discovered in each interval, $\boldsymbol{\gamma}$ is the vector of model parameters contained in the hazard function, and y_i is the number of vulnerabilities discovered in the ith interval. Substituting one of the hazard functions specified in Section VI-B into Equation (2) produces unique log-likelihood functions. Thus, given covariate data $\mathbf{x}$ and the vector of vulnerabilities discovered in each of the n intervals ($\mathbf{y}_n$), the model fitting step identifies the numerical values of the total number of vulnerabilities to be discovered (ω), vector of m covariate coefficients ($\boldsymbol{\beta}$), and hazard function parameters ($\boldsymbol{\gamma}$).

Letting $\boldsymbol{\theta} = \{\boldsymbol{\gamma}, \boldsymbol{\beta}, \omega\}$ denote the vector of all model parameters, the log-likelihood expression can be reduced from $|\boldsymbol{\theta}|$ to $|\boldsymbol{\theta}| - 1$ parameters by differentiating the log-likelihood function with respect to ω, equating the result to zero, solving for ω to produce

$$\hat{\omega} = \frac{\sum_{i=1}^{n} y_i}{\sum_{i=1}^{n} p_{i,\mathbf{x}_i}} \quad (8)$$

and substituting Equation (8) into the log-likelihood function to obtain a reduced log-likelihood (RLL) function.

The maximum likelihood estimates of the remaining $|\boldsymbol{\theta}| - 1$ parameters is determined by computing partial derivatives

$$\frac{\partial RLL}{\partial \boldsymbol{\beta}} = 0 \quad (9)$$

and

$$\frac{\partial RLL}{\partial \boldsymbol{\gamma}} = 0 \quad (10)$$

These steps can be applied to the alternative hazard functions in Section VI-B to obtain the corresponding maximum likelihood estimates of the models. Solving this system of equations and substituting the numerical values $\hat{\boldsymbol{\beta}}$ and $\hat{\boldsymbol{\gamma}}$ into Equation (8) produces the maximum likelihood estimate of $\hat{\omega}$.

C. Alhazmi-Malaiya Logistic (AML) model

This section derives expressions for the Alhazmi-Malaiya Logistic model [78] as a nonhomogeneous Poisson process.

The mean value function of the AML model is

$$m(t) = \frac{B}{Bce^{-ABt} + 1} \quad (11)$$

where B is interpreted as the number of vulnerabilities that would be discovered with indefinite testing, while A and c

are constants of proportionality characterizing the vulnerability discovery rate. The corresponding failure intensity function is

$$\lambda(t) = \frac{B^3 cAe^{-ABt}}{(Bce^{-ABt}+1)^2} \tag{12}$$

To estimate the parameters of the AML model given vulnerability discovery data, the grouped data form of the likelihood function is

$$\begin{aligned} LL(\mathbf{t},\mathbf{k};\boldsymbol{\theta}) &= \sum_{i=1}^{n} k_i \log(m(t_i) - m(t_{i-1}) \\ &\quad - \sum_{i=1}^{n} \log(k_i!) - \sum_{i=1}^{n} (m(t_i) - m(t_{i-1})) \end{aligned} \tag{13}$$

where $< \mathbf{t},\mathbf{k} >=< (t_1,k_1),(t_2,k_2),\ldots,(t_n,k_n) >$, t_i the time at which the i^{th} interval ended, k_i is the number of vulnerabilities discovered in interval i, and $\boldsymbol{\theta} = \{A, B, c\}$ the vector of model parameters.

The maximum likelihood estimates the AML model parameters are determined by solving the system of equations

$$\frac{\partial LL}{\partial \boldsymbol{\theta}} = \mathbf{0} \tag{14}$$

In addition to solving the system of equations defined by Equation (14) with algorithms such as Newton's method [79], alternative techniques include Bayesian methods [80], evolutionary algorithms [81], machine learning [82], and the expectation maximization (EM) [83] algorithm.

VII. Model Assessment

This section describes quantitative goodness of fit measures to assess how well a model performs on a given data set. In practice, it is rare that a single model performs best on all measures. Therefore, model selection often requires a subjective choice based on the preferences of a decision-maker. Regardless of these preferences, a primary consideration is the tradeoff between model complexity and predictive accuracy.

Sum of squares error (SSE) is calculated by fitting a model with n observations with maximum likelihood estimation and then computing the sum of squares difference between the observations and model predictions.

$$SSE = \sum_{i=1}^{n} (N(i) - \hat{m}(i))^2 \tag{15}$$

where $N(t_i)$ is the number of vulnerabilities discovered in the first i intervals and $\hat{m}(i)$ is the fitted model's estimate of the number of vulnerabilities discovered.

Predictive sum of squares error (PSSE)) fits a model with the first $n-\ell$ observations and then computes the sum of squares of the prediction residuals for the remaining ℓ observations not used to fit the model.

$$PSSE = \sum_{i=(n-\ell+1)}^{n} (N(i) - \hat{m}(i))^2 \tag{16}$$

Akaike information criterion (AIC) quantifies the tradeoff between a model's complexity and characterization of the observed data.

$$AIC = 2\nu - 2LL(\hat{\boldsymbol{\theta}}) \tag{17}$$

where ν penalizes models with more parameters.

Bayesian information criterion (BIC) is similar to the AIC. However, the penalty term includes the sample size (n).

$$BIC = \nu \log(n) - 2LL(\hat{\boldsymbol{\theta}}) \tag{18}$$

VIII. Analysis

This section performs a comparative analysis of software vulnerability discovery models with and without covariates, including their goodness of fit and predictive accuracy on the data created as part of this study (Table II).

A. Goodness of fit model assessment

For the vulnerability model with covariates, all three covariates specified in Table II were employed by substituting the values into Equation (7). A hazard function was then substituted into Equation (2) and the likelihood maximized. For the AML, which cannot include covariates, each time interval (t_i) was set to five, since this was the amount of time (in hours) required to setup and execute the vulnerability discovery tools. To avoid unfairly disadvantaging the AML model, intervals of length $t_i = Z_i + B_i + M_i$ were also considered, as this was the most straightforward method of incorporating information on the test activities into the AML model. However, this approach makes the simplifying assumption that each activity contributed equally to vulnerability discovery.

Table III summarizes the goodness of fit of the AML model and the covariate vulnerability discovery model with alternative hazard rate functions described is Section VI-B. For each measure, lower values are more favorable and the preferred model is indicated in bold.

TABLE III
Goodness of fit of vulnerability discovery models with and without covariates

Model	*SSE*	*PSSE*	*AIC*	*BIC*
AML	34.2907	5.6025	47.0145	47.2528
AML ($Z_i + B_i + M_i$)	21.5592	1.7770	41.8812	42.1195
GM	6.6994	2.7028	35.5228	35.9200
NB2	0.9573	0.1007	**34.9565**	**35.3537**
DW2	13.8840	2.8550	40.8420	41.2390
DW3	1.2286	0.3291	36.9901	37.4667
S	**0.8891**	**0.0857**	36.9387	37.4153
TL	6.4349	2.5106	37.5281	38.0047
IFRSB	95.1872	8.9720	65.7081	66.1053
IFRGSB	6.4503	2.5154	37.5306	38.0072

The variant of the covariate model incorporating the S distribution as the hazard function achieved the lowest SSE and PSSE, while the NB2 hazard function was a close second on both of

these measures and also performed best with respect to AIC and BIC. The covariate vulnerability discovery model with IFRSB hazard function and AML model without covariates ranked lowest on all measures, The AML model incorporating covariates by setting $t_i = Z_i + B_i + M_i$ performed slightly better, but still ranked third worst on the SSE, AIC, and BIC measures, suggesting that the vulnerability discovery model with covariates performed substantially better, despite the fact that the AIC and BIC measures penalized the inclusion of additional parameters for the covariates associated with vulnerability discovery activities. It should also be noted that covariates Z, B, and M were not treated as model parameters in the calculation of the AIC and BIC for the AML model, making the results in favor of the covariate VDM even more compelling for these measures, since a difference of two is consider to be statistically significant.

B. Assessment of model fit and vulnerability intensity

Figure 2 shows the empirical vulnerability count in each interval as reported in Table II as well as the overall best fitting software vulnerability discovery model incorporating covariates with second order negative binomial hazard function and the AML model with intervals characterized by $t_i = Z_i + B_i + M_i$. The dashed vertical line at interval $t = 7$ indicates that this and intervals to the left were used for model fitting and the last interval (12.5% of the intervals) was used to compute predictive accuracy measures.

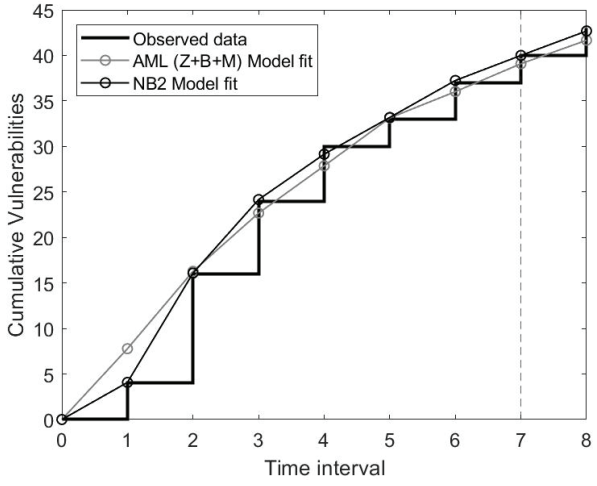

Fig. 2. Empirical vulnerability discovery process and best fitting models with and without covariates

Here, the number of vulnerabilities predicted in each interval were obtained from Equation (1) and Equation (11) for covariate and AML models respectively.

To further illustrate the enhanced fit and predictions attained by the software VDM incorporating covariates over the AML model, Figure 3 shows the number of vulnerabilities discovered in each interval as well as the vulnerability discovery intensity of the fitted models. Figure 3 clearly indicates that

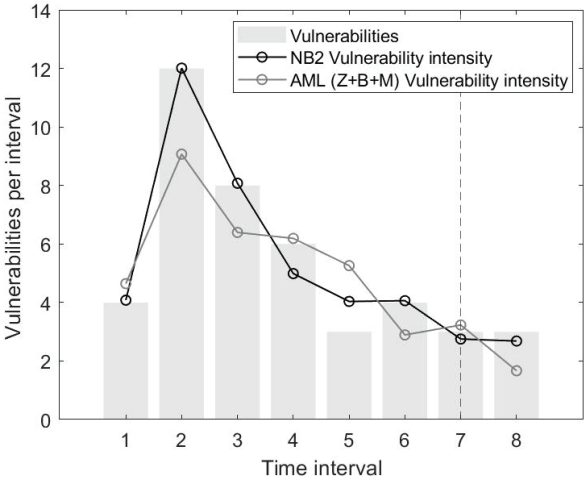

Fig. 3. Vulnerabilities discovered in each interval of the data set

the software VDM with covariates not only tracks the intervals used to fit models much better, but is also capable of predicting the future intervals much more accurately, whereas models without covariates such as the AML are only capable of capturing trends implicit in their parametric forms.

IX. Conclusions and Future Research

This paper presented a comparative study of a software vulnerability discovery model incorporating covariates with the Alhazmi-Malaiya Logistic model, one of the most flexible VDM without covariates. Our results indicated that the software VDM incorporating covariates (i) more accurately tracked and predicted the number of vulnerabilities discovered in future intervals as a function of penetration testing activities performed and (ii) achieved significantly better goodness of fit, despite the fact that the information theoretic measures penalized the covariate models for their additional parameters. The AML model, on the other hand, could only characterize primary trends embedded in its model form. Thus, software VDMs incorporating covariates offer a more detailed methodology to assess the effectiveness of alternative tools and techniques to discover vulnerabilities as well as guide the allocation of test activities and process improvement efforts.

Future research will combine the techniques presented here with other reliability engineering techniques to provide more comprehensive, accurate, and usable methods that support systematic test and evaluation of software.

Acknowledgment

This research was supported by the National Science Foundation under Grant Number 1749635 and the Homeland Security Community of Best Practices (HS CoBP) through the U.S. Department of the Air Force under award number

SCR1158132. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, U.S. Department of Homeland Security or U.S. Department of the Air Force.

REFERENCES

[1] "Common weakness enumeration," https://cwe.mitre.org/, accessed: 2021-05-22.

[2] "OWASP top 10," https://owasp.org/www-project-top-ten/, accessed: 2022-05-22.

[3] M. Vieira, N. Antunes, and H. Madeira, "Using web security scanners to detect vulnerabilities in web services," in *Proc. of International Conference on Dependable Systems & Networks*. IEEE/IFIP, 2009, pp. 566–571.

[4] N. Antunes and M. Vieira, "Assessing and comparing vulnerability detection tools for web services: Benchmarking approach and examples," *IEEE Transactions on Services Computing*, vol. 8, no. 2, pp. 269–283, 2014.

[5] "Common attack pattern enumeration and classification," https://capec.mitre.org/, accessed: 2021-05-22.

[6] S. S. Alqahtani, E. E. Eghan, and J. Rilling, "Sv-af—a security vulnerability analysis framework," in *Proc. of International Symposium on Software Reliability Engineering*. IEEE, 2016, pp. 219–229.

[7] E. Fong, D. A. Wheeler, and A. E. Henninger, "State-of-the-Art Resources (SOAR) for software vulnerability detection, test, and evaluation," Institute for Defense Analyses Alexandria, Tech. Rep., 2016.

[8] O. H. Alhazmi and Y. K. Malaiya, "Modeling the vulnerability discovery process," in *Proc. of International Symposium on Software Reliability Engineering*, 2005, pp. 10–pp.

[9] "NIST risk management framework," https://csrc.nist.gov/projects/risk-management/, accessed: 2022-05-22.

[10] V. Nagaraju, C. Jayasinghe, and L. Fiondella, "Optimal test activity allocation for covariate software reliability and security models," *Journal of Systems and Software*, vol. 168, p. 110643, 2020.

[11] "Common vulnerabilities and exposure," https://cve.mitre.org/, accessed: 2021-05-22.

[12] "CAS static analysis tool study - methodology," National Security Agency Center for Assured Software, Tech. Rep., 2012.

[13] "National vulnerability database," https://nvd.nist.gov/, accessed: 2021-05-22.

[14] "CWE/SANS top 25 most dangerous software errors," https://www.sans.org/top25-software-errors/, accessed: 2022-05-22.

[15] "Defense acquisition guidebook," https://www.dau.edu/tools/dag, accessed: 2022-05-22.

[16] "OWASP," https://owasp.org/, accessed: 2022-05-22.

[17] "Web Application Security Consortium," http://www.webappsec.org/, accessed: 2022-05-22.

[18] R. M. Brady, R. J. Anderson, and R. C. Ball, "Murphy's law, the fitness of evolving species, and the limits of software reliability," University of Cambridge, Computer Laboratory, Tech. Rep., 1999.

[19] R. Anderson, "Security in open versus closed systems—the dance of boltzmann, coase and moore," 2002.

[20] O. H. Alhazmi and Y. K. Malaiya, "Quantitative vulnerability assessment of systems software," in *Proc. of Annual Reliability and Maintainability Symposium*, 2005, pp. 615–620.

[21] E. Rescorla, "Is finding security holes a good idea?" *IEEE Security & Privacy*, vol. 3, no. 1, pp. 14–19, 2005.

[22] J. D. Musa and K. Okumoto, "A logarithmic poisson execution time model for software reliability measurement," in *Proc. of the International Conference on Software Engineering*. Citeseer, 1984, pp. 230–238.

[23] O. H. Alhazmi, Y. K. Malaiya, and I. Ray, "Measuring, analyzing and predicting security vulnerabilities in software systems," *Computers & Security*, vol. 26, no. 3, pp. 219–228, 2007.

[24] J. Kim, Y. K. Malaiya, and I. Ray, "Vulnerability discovery in multi-version software systems," in *Proc. of High Assurance Systems Engineering Symposium*. IEEE, 2007, pp. 141–148.

[25] E. Condon, M. Cukier, and T. He, "Applying software reliability models on security incidents," in *Proc. of International Symposium on Software Reliability*. IEEE, 2007, pp. 159–168.

[26] K. Kanoun, M. R. de Bastos Martini, and J. M. De Souza, "A method for software reliability analysis and prediction application to the tropico-r switching system," *IEEE Transactions on Software Engineering*, vol. 17, no. 4, p. 334, 1991.

[27] A. L. Goel and K. Okumoto, "Time-dependent error-detection rate model for software reliability and other performance measures," *IEEE Transactions on Reliability*, vol. 28, no. 3, pp. 206–211, 1979.

[28] S. Yamada, M. Ohba, and S. Osaki, "S-shaped software reliability growth models and their applications," *IEEE Transactions on Reliability*, vol. 33, no. 4, pp. 289–292, 1984.

[29] J. Duane, "Learning curve approach to reliability monitoring," *IEEE transactions on Aerospace*, vol. 2, no. 2, pp. 563–566, 1964.

[30] T. Khoshgoftaar, "Nonhomogeneous poisson processes for software reliability growth," in *Proc. of International Conference on Computational Statistics*, 1988, pp. 13–14.

[31] H. Okamura, M. Tokuzane, and T. Dohi, "Optimal security patch release timing under non-homogeneous vulnerability-discovery processes," in *Proc. of International Symposium on Software Reliability Engineering*. IEEE, 2009, pp. 120–128.

[32] H. Joh, J. Kim, and Y. K. Malaiya, "Vulnerability discovery modeling using weibull distribution," in *Proc. of International Symposium on Software Reliability Engineering*. IEEE, 2008, pp. 299–300.

[33] A. Younis, H. Joh, and Y. Malaiya, "Modeling learningless vulnerability discovery using a folded distribution," in *Proc. of SAM*, vol. 11. Citeseer, 2011, pp. 617–623.

[34] X. Wang, R. Ma, B. Li, D. Tian, and X. Wang, "E-wbm: an effort-based vulnerability discovery model," *IEEE Access*, vol. 7, pp. 44 276–44 292, 2019.

[35] A. Anand, N. Bhatt, and O. H. Alhazmi, "Modeling software vulnerability discovery process inculcating the impact of reporters," *Information Systems Frontiers*, vol. 23, no. 3, pp. 709–722, 2021.

[36] V. H. Nguyen and F. Massacci, "A systematically empirical evaluation of vulnerability discovery models: A study on browsers' vulnerabilities," *arXiv preprint arXiv:1306.2476*, 2013.

[37] H. Okamura, M. Tokuzane, and T. Dohi, "Quantitative security evaluation for software system from vulnerability database," 2013.

[38] Y. Kansal, P. Kapur, U. Kumar, and D. Kumar, "Effort and coverage dependent vulnerability discovery modeling," in *Proc. of International Conference on Telecommunication and Networks*. IEEE, 2017, pp. 1–6.

[39] D. Stuttard and M. Pinto, *The web application hacker's handbook: Finding and exploiting security flaws*. John Wiley & Sons, 2011.

[40] "OWASP mobile top 10," https://owasp.org/www-project-mobile-top-10/, accessed: 2022-05-22.

[41] "OWASP desktop app security top 10," https://owasp.org/www-project-desktop-app-security-top-10/, accessed: 2022-05-22.

[42] "Core Flight System: A paradigm shift in flight software development," https://cfs.gsfc.nasa.gov/, accessed: 2022-05-22.

[43] K. Shibata, K. Rinsaka, and T. Dohi, "Metrics-based software reliability models using non-homogeneous poisson processes," in *Proc. of International Symposium on Software Reliability Engineering*, 2006, pp. 52–61.

[44] M. Cova, V. Felmetsger, and G. Vigna, "Vulnerability analysis of web-based applications," in *Test and Analysis of Web Services*. Springer, 2007, pp. 363–394.

[45] "The state of the state of application exploits in security incidents," Cyentia Intstitute, and F5 Labs, Report, 2021.

[46] "Vulnlab," https://github.com/Yavuzlar/VulnLab, accessed: 2022-05-22.

[47] "Damn vulnerable web application," https://github.com/digininja/DVWA, accessed: 2022-05-22.

[48] "Xtreme vulnerable web application," https://github.com/s4n7h0/xvwa, accessed: 2022-05-22.

[49] "OWASP juice shop," https://github.com/juice-shop/juice-shop, accessed: 2022-05-22.

[50] S. Rafique, M. Humayun, B. Hamid, A. Abbas, M. Akhtar, and K. Iqbal, "Web application security vulnerabilities detection approaches: A systematic mapping study," in *Proc. of International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. IEEE/ACIS, 2015, pp. 1–6.

[51] R. S. Sandhu and P. Samarati, "Access control: principle and practice," *IEEE communications magazine*, vol. 32, no. 9, pp. 40–48, 1994.

[52] L. Burkhalter, N. Küchler, A. Viand, H. Shafagh, and A. Hithnawi, "Zeph: Cryptographic enforcement of end-to-end data privacy," in *Proc. of Symposium on Operating Systems Design and Implementation*. USENIX, 2021, pp. 387–404.

[53] G. Deepa and P. S. Thilagam, "Securing web applications from injection and logic vulnerabilities: Approaches and challenges," *Information and Software Technology*, vol. 74, pp. 160–180, 2016.
[54] W. G. Halfond, J. Viegas, A. Orso *et al.*, "A classification of sql-injection attacks and countermeasures," in *Proc. of International Symposium on Secure Software Engineering*, vol. 1. IEEE, 2006, pp. 13–15.
[55] A. Ron, A. Shulman-Peleg, and A. Puzanov, "Analysis and mitigation of nosql injections," *IEEE Security & Privacy*, vol. 14, no. 2, pp. 30–39, 2016.
[56] T. P. Vuong, G. Loukas, D. Gan, and A. Bezemskij, "Decision tree-based detection of denial of service and command injection attacks on robotic vehicles," in *Proc. of International Workshop on Information Forensics and Security*. IEEE, 2015, pp. 1–6.
[57] A. Shostack, *Threat modeling: Designing for security*. John Wiley & Sons, 2014.
[58] A. Apvrille and M. Pourzandi, "Secure software development by example," *IEEE Security & Privacy*, vol. 3, no. 4, pp. 10–17, 2005.
[59] H. Nina, J. A. Pow-Sang, and M. Villavicencio, "Systematic mapping of the literature on secure software development," *IEEE Access*, vol. 9, pp. 36 852–36 867, 2021.
[60] B. Eshete, A. Villafiorita, and K. Weldemariam, "Early detection of security misconfiguration vulnerabilities in web applications," in *Proc. of International Conference on Availability, Reliability and Security*. IEEE, 2011, pp. 169–174.
[61] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *Proc. of International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2020, pp. 23–43.
[62] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," *arXiv preprint arXiv:1811.00918*, 2018.
[63] R. Bird, I. Gopal, A. Herzberg, P. Janson, S. Kutten, R. Molva, and M. Yung, "Systematic design of two-party authentication protocols," in *Proc. of Annual International Cryptology Conference*. Springer, 1991, pp. 44–61.
[64] D. Wang and P. Wang, "On the usability of two-factor authentication," in *Proc. of International Conference on Security and Privacy in Communication Networks*. Springer, 2014, pp. 141–150.
[65] J. Bonneau and S. Preibusch, "The password thicket: Technical and market failures in human authentication on the web." in *WEIS*. Citeseer, 2010.
[66] P. A. Hallgren, D. T. Mauritzson, and A. Sabelfeld, "Glasstube: A lightweight approach to web application integrity," in *Proc. of ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2013, pp. 71–82.
[67] M. Babiker, E. Karaarslan, and Y. Hoscan, "Web application attack detection and forensics: A survey," in *Proc. of International Symposium on Digital Forensic and Security*. IEEE, 2018, pp. 1–6.
[68] J. Arulraj, M. Perron, and A. Pavlo, "Write-behind logging," *Proc. of the VLDB Endowment*, vol. 10, no. 4, pp. 337–348, 2016.
[69] B. Jabiyev, O. Mirzaei, A. Kharraz, and E. Kirda, "Preventing server-side request forgery attacks," in *Proc. of Symposium on Applied Computing*. ACM, 2021, pp. 1626–1635.
[70] R. Hertzog, J. O'Gorman, and M. Aharoni, "Kali linux revealed," *Mastering the Penetration Testing Distribution*, 2017.
[71] "OWASP Zed Attack Proxy," https://www.zaproxy.org/, accessed: 2022-05-22.
[72] "Burp Suite," https://portswigger.net/burp, accessed: 2022-05-22.
[73] C. Bracquemond and O. Gaudoin, "A survey on discrete lifetime distributions," *International Journal of Reliability, Quality and Safety Engineering*, vol. 10, no. 1, pp. 69–98, 2003.
[74] W. Padgett and J. Spurrier, "On discrete failure models," *IEEE Transactions on Reliability*, vol. R-34, no. 3, pp. 253–256, aug 1985.
[75] J. Soler, "Croissance de fiabilite des versions d'un logiciel," *Revue de statistique appliquée*, vol. 44, no. 1, pp. 5–20, 1996.
[76] G. Adams and R. Watson, "A discrete time parametric model for the analysis of failure time data," *Australian Journal of Statistics*, vol. 31, no. 3, pp. 365–384, 1989.
[77] A. Salvia and R. Bollinger, "On discrete hazard functions," *IEEE Transactions on Reliability*, vol. R-31, no. 5, pp. 458–459, 1982.
[78] O. H. Alhazmi and Y. K. Malaiya, "Application of vulnerability discovery models to major operating systems," *Transactions on Reliability*, vol. 57, no. 1, pp. 14–22, 2008.
[79] R. L. Burden and J. D. Faires, *Numerical Analysis*. Boston, MA: BrooksCole, 2011.
[80] B. Littlewood and J. L. Verrall, "A Bayesian reliability model with a stochastically monotone failure rate," *IEEE Transactions on Reliability*, vol. 23, no. 2, pp. 108–114, 1974.
[81] T. Minohara and Y. Tohma, "Parameter estimation of hyper-geometric distribution software reliability growth model by genetic algorithms," in *Proc. International Symposium on Software Reliability Engineering*. IEEE, 1995, pp. 324–329.
[82] N. Karunanithi, D. Whitley, and Y. K. Malaiya, "Prediction of software reliability using connectionist models," *IEEE Transactions of Software Engineering*, vol. 18, no. 7, pp. 563–574, jul 1992.
[83] H. Okamura, Y. Watanabe, and T. Dohi, "An iterative scheme for maximum likelihood estimation in software reliability modeling," in *Proc. IEEE International Symposium on Software Reliability Engineering*, 2003, pp. 246–256.