IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 1 August 2023, accepted 10 September 2023, date of publication 14 September 2023, date of current version 19 September 2023.

*Digital Object Identifier 10.1109/ACCESS.2023.3315595*

RESEARCH ARTICLE

# Open Source Solutions for Vulnerability Assessment: A Comparative Analysis

**DINIS BARROQUEIRO CRUZ, JOÃO RAFAEL ALMEIDA, AND JOSÉ LUÍS OLIVEIRA**
Institute of Electronics and Informatics Engineering of Aveiro (IEETA), Department of Electronics, Telecommunications and Informatics (DETI), LASI, University of Aveiro, 3810-193 Aveiro, Portugal

Corresponding author: Dinis Barroqueiro Cruz (cruzdinis@ua.pt)

This work was supported by the European Union/European Federation of Pharmaceutical Industries and Associations (EU/EFPIA) Innovative Medicines Initiative 2 Joint Undertaking under Grant 806968.

**ABSTRACT** As software applications continue to become more complex and attractive to cyber-attackers, enhancing resilience against cyber threats becomes essential. Aiming to provide more robust solutions, different approaches were proposed for vulnerability detection in different stages of the application life-cycle. This article explores three main approaches to application security: Static Application Security Testing (SAST), Dynamic Application Security Testing (DAST), and Software Composition Analysis (SCA). The analysis conducted in this work is focused on open-source solutions while considering commercial solutions to show contrast in the approaches taken and to better illustrate the different options available. It proposes a baseline comparison model to help evaluate and select the best solutions, using comparison criteria that are based on community standards. This work also identifies future opportunities for application security, highlighting some of the key challenges that still need to be addressed in order to fully protect against emerging threats, and proposes a workflow that combines the identified tools to be used for vulnerability assessments.

**INDEX TERMS** Application security, static application security testing, dynamic application security testing, software composition analysis, vulnerability assessment.

## I. INTRODUCTION

Continuous security threats pose a significant challenge for organizations, as new vulnerabilities and attacks are constantly emerging [1], [2], [3]. These vulnerabilities are introduced during the development process [4], [5] and these issues are often recurring [6], meaning that their detection is far simpler than a zero-day attack. Public efforts exist to keep track of vulnerabilities like the CVE program[1] and efforts to prevent security risks are seen within the CWE[2] program which presents hardware and software weakenesses that can have security ramifications.

According to ENISA's Threat Landscape report for 2022 [7], these known vulnerabilities are still problematic and can be identified in the majority of web solutions. This report identifies 11 006 occurrences of such vulnerabilities in different domains (as shown in Table 15 of the report). In order to stay ahead of these threats, it is important for organizations to adopt a proactive approach to security that involves continuously testing and analyzing their applications for vulnerabilities. Continuous secure development practices allow this approach by integrating security testing and analysis into the software development process, organizations may reduce the impact of known vulnerabilities [8].

The creation and maintenance of safe online applications depend heavily on secure software development which entails using best practices and procedures that put security first at every stage of the development process [9]. Organizations are prioritizing security as web applications have become integral to the internet and as such provide motivation for their wrong usage, this leads to a number of common patterns in vulnerabilities such as SQL Injections or Cross-site Scripting (XSS) [10]. ENISA's Threat Landscape report for 2022 further supports this purpose by identifying web applications as one of three main data breach vectors.

The associate editor coordinating the review of this manuscript and approving it for publication was Hailong Sun.

[1]https://cve.mitre.org
[2]https://cwe.mitre.org

The analysis on AppSec tools has already been performed previously. Curphey and Araujo [11] provided one of the first analyses of this type by introducing threat modelling concepts and mapping out different types of vulnerabilities for web application security. The comparison produced is simple by considering the main features each tool considered has along with when it is used during the SDLC and the expertise level necessary to operate them and understand their results. As vulnerabilities became more intricate and specialized, reviews such as the one proposed by Alzahrani et al. [12] narrow down the focus to specific problems, namely focused on Insufficient Transport Layer Protection, Information Leakage, Cross-Site Scripting, and SQL Injection. However, these comparisons rely on simple aspects such as the features each tool has and the source-code availability. Amankwah et al. [13] documents a comparison that provides insight not only into the tools selected but also into strategies for analysis. This work brings small contributions into benchmarks for comparison between tools.

Besides these comparisons, there has been an effort to create and test different benchmarks for comparison between solutions [14], [15], [16], [17]. One of the issues that we identified in the mentioned reviews is the lack of comparison metrics since those are mostly based on tool's functionalities. The work proposed in this article builds upon the research done by previous work and it also expands the list of comparison of tools resulting in a purposive review of automated tools to analyse web applications. The analysis was based on the three main approaches for application security: Static Application Security Testing (SAST), Dynamic Application Security Testing (DAST), and Software Composition Analysis (SCA). This work is focused on open-source solutions while considering commercial solutions as well to show contrast in the approaches taken and better illustrate the universe of options. This analysis aims to provide guidelines to software engineers regarding which tools should be incorporated in their CI/CD pipelines, to better identify known vulnerabilities at early stages in the development process.

## II. BACKGROUND

Given the importance of web applications, some metrics and concepts were established to simplify the comparison between automated solutions developed for vulnerability scanning.

### A. APPLICATION SECURITY

Application Security (AppSec) encompasses the methodologies used to identify, address, and protect software against known vulnerabilities within the applications. AppSec started as a manual process but this was expanded to include automatic operations that can be reproducible and faster to execute. It is employed throughout the Software Development Life Cycle (SDLC) and it may involve a multidisciplinary effort [18].

The major objective is to find and address software security vulnerabilities prior to them being exploited. In the testing process, vulnerabilities are discovered and information about them is produced through analysis, and reporting. Such testing offers advantages beyond a well-written project; it aids in the detection and prevention of possible issues, which is crucial for the majority of developed solutions. While different subdivisions exist, to simplify its characterization, we can consider the seven subdivisions of AppSec for our purposes.

For monitorization, Web Application Firewall (WAF) [19] serves as a defense mechanism by inspecting HTTP traffic that reaches the application and traffic that gets returned from it, essentially protecting the server from exposure. Similar to WAF, Runtime Application Self-Protection (RASP) [20] is a technique used to track and analyze user behavior during runtime and report on the potential exploitation of known vulnerabilities. Unlike perimeter-based protections like WAF, RASP can rely on the current context of the application to report on potential attacks. It also has a wider range of actions it can take, as it can influence the application in real-time [20].

Some analysis don't require running software and analyse problems statically. Software Composition Analysis (SCA) creates a list of third-party components of the solution it is analysing. With this list, SCA can then report on vulnerabilities that have been disclosed about the third-party software for specific versions, this also takes into account dependency graphs, which can sometimes produce false positives in real-life projects if they are analyzed statically [21]. Dynamic calls can help address this issue.

When considering the direct analysis of the application source code, it is used Static Application Security Testing (SAST). This inspects static source code and reports on weaknesses found, ranging from syntax errors to invalid insecure references. The most common approach is to create symbol tables, Abstract Syntax Trees (AST), control-flow graphs, and the main program control graph. These structures are then queried based on the type of vulnerability the tool is looking for, using simple or sophisticated algorithms [22].

In case of being possible to have a running instance of the product in a controlled environment, Dynamic Application Security Testing (DAST) can be used to conduct large-scale scans from an attacker's perspective. This simulates malicious inputs and collects information on how the application responds to this data. This is done without access to the source code. DAST contrasts with SAST by working following a black-box-like approach by performing attacks. SAST works as white-box testing, finding vulnerable patterns that might not be vulnerable in production [23].

On a hybrid approach, Interactive Application Security Testing (IAST) [24] employs techniques between SAST and DAST. It interacts with the application using a dynamic approach, whether it be manual or automated, and then correlates the results with a static analysis performed to give the root cause of the vulnerabilities discovered. This process is much more complex than SAST or DAST but also produces much more comprehensive results [24].

Finally, Mobile Application Security Testing (MAST) [25] is a broad term that encompasses a variety of techniques used in mobile security testing. These techniques can utilize both DAST and SAST approaches, as well as recover forensic data from the device. Threats and attacks to mobile devices can include sniffing, spamming, spoofing, phishing, and many others [25]. In our analysis we focused on exploring SAST, DAST and SCA techniques.

### B. VULNERABILITY ASSESSMENT

Vulnerability assessment, in a broader sense, is the process of identifying, documenting, and classifying vulnerabilities within a system. In the context of information technology systems, this definition narrows down to finding, documenting, classifying, and possibly mitigating security weaknesses within an information system, as seen in Figure 1. In this context, a vulnerability is a flaw that can be exploited to disrupt the normal functionality of a program [26].

AppSec relies heavily on vulnerability assessment to produce information on potential risks allowing for a prioritization and remediation of problems. A continuous AppSec process requires ongoing vulnerability assessment to maintain its efficiency. However, to promote consensus and facilitate communication within the community, some standards were defined. The CWE Top 25 is a list compiled by the community of the 25 most serious flaws in software security. These flaws are not vulnerabilities, but rather conditions that, in certain situations, can lead to a vulnerability [27]. The OWASP Top 10 serves as a standardized awareness resource for developers and web application security. It encompasses a wide consensus on the most significant security risks associated with web applications [28].

Both the OWASP Top 10 and CWE Top 25 have been around for some time but are kept up to date and as such, while older versions can be useful for historic reasons, the most recent versions represent the reality of the most critical problems. It is important that AppSec tools report on the most critical security risks in web applications, and the OWASP Top 10 represents a broad consensus on these risks.

### C. CYBERSECURITY INTEROPERABILITY

To share information and safeguard the cyberspace, there is a necessity for information sharing inside this environment. The primary prerequisites are a structured format (preferably machine readable) and the capacity to provide as much context and information as feasible [29]. While vulnerability assessment requires standards for a consensus on what is most critical, it also requires standards for how the information of a certain issue is described.

Standards like the Structured Threat Information eXpression (STIX) format allow representation of cybersecurity threats, this is possible due to it's range of data objects to represent different types of information, the data objects include information about a threat actor, malware, Tactics, Techniques and Procedures (TTP) and incident information [30].

While STIX offers a broader representation of cybersecurity incidents, standards like OVAL (Open Vulnerability and Assessment Language) narrow down the usability for vulnerability specific information sharing. Based on the same concepts of data objects, OVAL includes specific fields for vulnerability information, definition of the problem, testing conditions and details about the affected component [31].

With AppSec using automated scanning tools to produce results on security issues, the Static Analysis Results Interchange Format (SARIF) follows STIX and OVAL in conceptual detail but narrows down it's focus even more on tool output and as such the data structures offered refer to tool and issue information along with detailed information on the location of the problem.

### D. SOFTWARE PACKAGING

Containers are a way to package software and all of its dependencies with the major objective of being executable/deployed efficiently while offering a certain amount of isolation. The first containers were simple applications of isolation technologies like chroot, with other solutions being built on top of Linux kernel features like namespaces and cgroups [32]. With the microservices architecture gaining ground, containers became a solution for the deployment and isolation of these services [33] using technologies like Docker,[3] LXC,[4] or OpenVZ.[5] Container orchestration was the next logical step, with solutions like Kubernetes and Docker Swarm providing load balancing, auto-scaling, limit control, and overall management of containers.

Virtualization technologies have become very important with the boom in cloud computing, and containers have become the preferred lightweight virtualization option as development becomes faster and more agile. Containers offer better mechanisms for orchestration and efficiency, although they lack in isolation compared to virtual machines [34]. Virtual machines and Linux containers have been compared [35], demonstrating that neither method significantly increases CPU or memory utilization, but virtual machines exhibit an issue with I/O access because it requires more layers. While virtual machines have improved over time, containers began off in a superior position. As a result, containers have far less space for efficiency growth. The majority of bottlenecks within containers seem to be caused by methods that reduce the efficiency of the interfaces while making them more user-friendly.

The container virtualization technology Docker provides a way to standardize program packaging and execution through the use of *images* and *containers*. Images act as both the fundamental building pieces and the framework of a

[3]https://www.docker.com
[4]https://linuxcontainers.org
[5]https://openvz.org

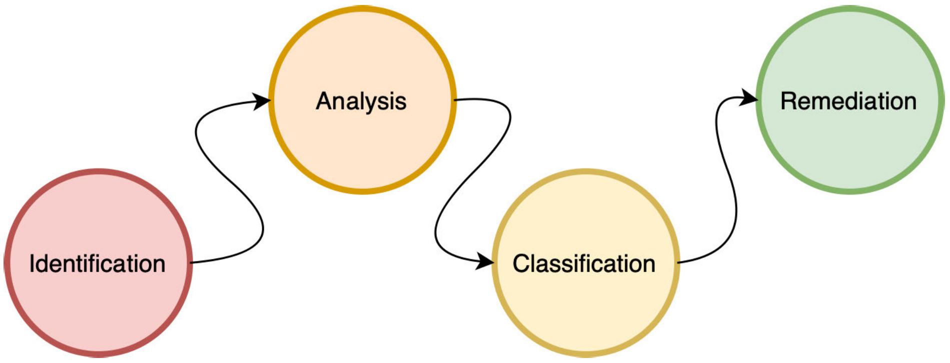

**FIGURE 1.** **Vulnerability assessment model.**

container. These images give the environment a fundamental configuration while allowing the user to add components. Using containers, which are instances of Docker images, they can be started, stopped or destroyed [36]. Docker employs certain mechanisms to achieve a certain level of isolation based on internal security mechanisms and their relationship to the host. Bui[6] analysed this relation discovering that containers implement filesystem, device, process, and resource isolation. This entails providing each container with its own view of the filesystem and preventing write access to specific directories and files, as well as whitelisting controllers with cgroups for device isolation, using Linux namespaces for process isolation, and using cgroups to restrict resources and fend off Denial of Service (DoS) attacks. Additionally, network namespaces, which provide each container with a different IP address and routing table, are used to achieve network isolation.

Combe et al. [37] verifies Bui's findings while also highlighting certain security issues. The research contends that while defence against external threats is taken into account, defence against other containers is weak. The report adds that since insecure images might potentially be an issue, it is crucial to assure image distribution by having developers sign their images. The study also shows how insecure local configurations can be a problem and can jeopardize security. For instance, sharing the host network with containers can facilitate certain options and tasks, but it can also compromise security. A literature survey by Sultan et al. [38] creates a complete risk model when considering containers. This model takes into consideration four use cases: i) protecting a container from the application it is running; ii) protecting a container from another container; iii) protecting a host from a container; and iv) protecting a container from a malicious host. The first three cases can be addressed with existing software based on Linux security features, and a hardware solution is proposed to address the last case.

Application Security tools must be packaged in a way that allows: i) consistency when ran in different infrastructures; ii) portability by packaging all dependencies in one unit; and iii) efficiency by being as lightweight as possible. This is made possible by using technologies like Docker which is also widely adopted within the community.

[6]https://arxiv.org/pdf/1501.02967

### E. CONTINUOUS INTEGRATION, DELIVERY AND DEPLOYMENT

The goal of Continuous Integration, Delivery and Deployment practices (CI/CD) is to provide a product frequently to customers by integrating automation into the various stages of development, essentially improving process efficiency and reducing the likelihood of human error.

#### 1) SOFTWARE DEVELOPMENT LIFE CYCLE

The traditional methodology of software development is based on very well-defined phases such as planning, designing, implementing, testing, and delivering the solution. These phases are done sequentially, and one phase cannot start until the previous one is completed. In an agile approach, the development is still grounded in these phases, but the process is divided into small increments, where each iteration uses each of the phases of development. Leau et al. [39] discuss the differences, advantages, and disadvantages of traditional software development strategies to the Software Development Life Cycle (SDLC) and agile methodologies. These two are generalized and illustrated in Figure 2.

However, agile methodologies have become more prominent, as it is more business-driven with requirements being defined as time goes on, making testing and client feedback continuous instead of leaving these aspects until the end of the SDLC. Traditional approaches do make it easier to estimate the cost of a process, as the totality of requirements must be defined at the start of the process. The continuous integration, deployment, and delivery are a solution to the incremental nature of agile SDLC, since in traditional methods, deployment/delivery and testing were left until the end of the process [40].

#### 2) CONTINUOUS INTEGRATION

The term Continuous Integration (CI) as it is known today was introduced by Beck [41]. It aims to provide rapid

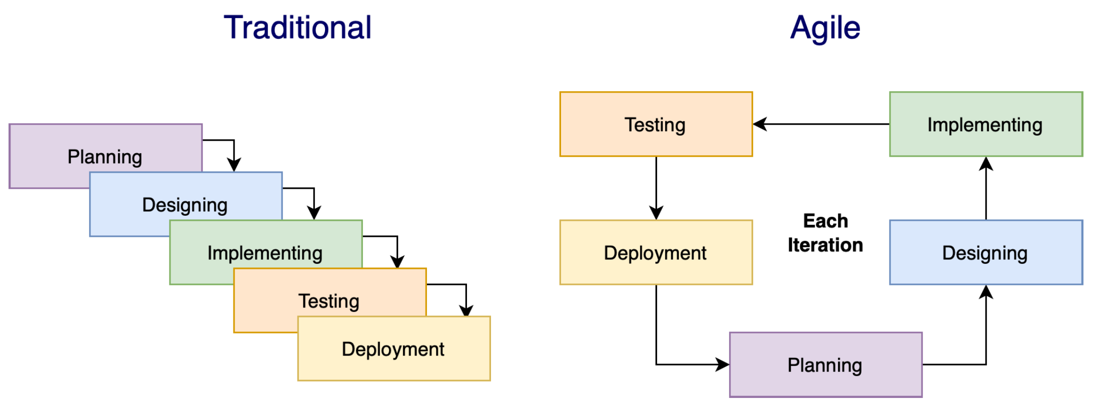

**FIGURE 2.** Traditional and agile software development life cycle.

integration of new developments in a software solution. This considers the building and testing of the application. Fowler and Foemmel [42] present a comprehensive description of CI practices. The process is described as having a single source repository, to enable code changes that can then be merged using a version control system. The code is then built automatically to ensure the changes did not cause a compilation error. The build should also include a suite of tests to cover the application. Other practices are also described, such as the importance of everyday commits, the immediate fixing of broken builds, the efficiency needed for good integration, and making the testing as close to production as possible. Transparency of the process is also seen as a major factor in the success of this approach, as it produces more communication between partners. A final remark is made mentioning that continuous deployment is the next logical step [42].

CI has been normally used alongside an agile software development process. It can automate the building of software, but it can also automate testing, whether that may be unit testing or more complex forms. The benefits of such methods include: i) finding integration bugs earlier; ii) speeding up the development process; iii) an easier way to pinpoint errors when fewer parts are tested at a time; iv) helping to follow the agile development process by design; and v) avoiding most last-minute problems. Some drawbacks are also encountered like i) initial work and commitment to creating the CI infrastructure; ii) smaller projects can become unnecessarily complex with CI; iii) CI does not have intrinsic value and is mostly based on how good the testing is; and iv) tracking deliverables and builds can become complex if there is a lot of code created in small amounts.

### 3) CONTINUOUS DELIVERY AND DEPLOYMENT

Following in the footsteps of automation, continuous delivery and deployment address the last stage of the development life cycle by automatically packaging and distributing the solution. Continuous delivery and deployment are the practices that allow the inclusion of new features, configuration changes, correction of issues, and even experiments in a production environment. These inclusions must be safe, relatively quick, and repeatable. This practice is better suited for an agile SDLC by design, with each feature developed or major release being available for testing or validation.

Continuous delivery is a subset of continuous deployment, because continuous delivery produces deployable artifacts, but they are then deployed manually. Chen [43] shows a real-life company application of continuous delivery (and continuous integration) in which benefits and challenges are not only considered from a product point of view, but also how the company as a whole must adopt these practices. While continuous deployment might be the ultimate goal of automatic software development, Leppänen et al. [44] provide an inside look into the field of these practices within companies and found that continuous delivery is adopted, but automatic deployment, while possible, is actively not chosen by these companies because of a loss of control during the move to production. These findings are corroborated by Shahin et al. [45] in their empirical study, which also reflects an interesting point by clearly dividing the terms *Continuous Delivery* and *Continuous Deployment*, which are often combined or misused.

These techniques work in harmony with CI, with some challenges [46], as this can produce a build of the software and possibly test it, to be deployed (or delivered) later. Chaining of these events is possible and recommended to make sure that if a build or tests fail, the deployment is not produced, as this would not only lead to more resource consumption, but also a faulty deployment. The chaining of these practices in automation constitutes a CI/CD Pipeline. CD has benefits that include: i) low-risk releases; ii) higher-quality releases; iii) lower costs, as problems are caught earlier; and iv) improved productivity and efficiency. This automation does have some drawbacks which include: i) difficulty in testing certain projects as automation can be challenging; ii) infrastructure scaling can be complex, considering the resources needed for automating a deployment; and iii) documentation needs to be updated with the deployment process.

#### 4) PIPELINES

The goal of CI/CD pipeline is to fully automate the processes of integrating CI/CD effectively, dependably, and repeatedly. The degree of automation in these pipelines might vary and they are likely to change as teams become more used to the procedures by discovering new ways to automate each phase [47]. The benefits of a CD/CD pipeline are closely tied to the benefits of CI/CD itself. These benefits include i) reduced deployment time; ii) early detection of errors (resulting in better code quality); and iii) the creation of an efficient, sometimes reusable infrastructure that leads to tighter feedback loops. While these benefits are also a direct result of agile development methods, they are realized through the implementation of a CI/CD pipeline [48].

These pipelines need two primary components: a server to conduct the many agreed-upon phases of CI/CD that adhere to the given specifications and a version control system to store and maintain code repositories. The CI/CD server can be self-hosted or managed by a third party, just like the version control system. Although a self-hosted version will demand more labor and upkeep, it can provide finer control over the solution. A third-party alternative, however, offers fewer possibilities for customisation. An important aspect of a CI/CD pipeline is modularity, where each step is independent and easier to maintain and troubleshoot [49]. This can also make the pipeline more efficient, as the entire process can be shortened if the application fails in earlier steps. Besides, we need to consider different applications of such pipelines. For instance, the use of CI/CD in blockchain distributed applications [50], [51].

The seven types of security application testing described previously can be integrated into a CI/CD pipeline. These tests can be seen as testing the application, with SAST and SCA typically being performed during continuous integration, since they involve analyzing the software without running it. Tests that require running the software, such as DAST or IAST, are commonly conducted after the software has been deployed in a testing environment. Monitoring solutions are not typically considered part of the testing process, but they can be used to complement the deployment [52].

## III. EXPLORATION TOOLS

In this work, we search for open-source tools and free solutions that can be used to perform SAST, SCA and DAST assessments. This research also contains some commercial solutions to better illustrate some of the solutions available. The search was primarily done within GitHub.[7] To create a baseline of comparison, selection criteria were adapted from OWASP [53].

This research was directed towards a project which uses Django as the development framework and as such solutions are geared towards Python and JavaScript, we also considered container solutions and dynamic analysis focused on web application. The selection criteria is listed bellow in order of importance (from most to least), but not all items should be seen has having the same impact. Depending on the needs of a certain project, the intrinsic value of each criteria can change and the importance given on the initial ranking can change as well. The ranking described here can be adapted to what better applies to a specific project. Therefore, the final list of considerations is:

1) **Programming Language** - The tool is designed for the language standards that the solution we are analyzing uses
2) **Budget** - The solution is free, has a free version or is paid
3) **CI/CD** - The tool is available for integration with CI/CD pipelines (This criteria also reflects the ease of this integration)
4) **OWASP Top Ten** - The tool has the ability to detect vulnerabilities based on the OWASP Top 10
5) **SARIF** - The tool's output is interoperable (Provides SARIF reporting)
6) **Framework Understanding** - The tool is able to understand frameworks/libraries that the project relies on
7) **Ease of Setup** - The tool is of easy installation and usage, the preferred method for deployment is docker

Some criteria is divided into 4 levels: i) Level 0 (blank space) indicates that the solution does not conform at all to the comparison criteria; ii) Level 1 (low) indicates that the solution conforms minimally to the comparison criteria; iii) Level 2 (medium) indicates that the solution conforms to the selection criteria with small limitations; and iv) Level 3 (high) indicates that the solution completely conforms to the criteria.

These levels don't always apply, the interoperability comparison criteria is seen as binary, either the tool suports output in this format or it doesn't (In this case the criteria is assessed using a ✓symbol).

The study performed is more or less uniform when considering the amount of solutions for each type of scan (Figure 3). Dependency scanning and container scanning lack a bit behind in the total number of solutions but we believe the universe of tools to be vast enough in all types of application security testing considered.

This uniformity trend is not seen when considering the amount of solutions that are recommended for interoperability and ease of deployment (Figure 4). Secret scanning, dependency scanning and dynamic scanning all found only one solution each that meets the comparison metrics.

Most tools are still actively maintained (Figure 5) which is important because outdated tools will not keep up with the current issues that should be reported. This classification was performed considering that if the solution has not been updated this year, is a public archive or is explicitly stated that it is no longer maintained then it was classified as such.

The visualization of information is an important aspect of each tool. While vulnerability identification is the main focus

[7]https://www.github.com/

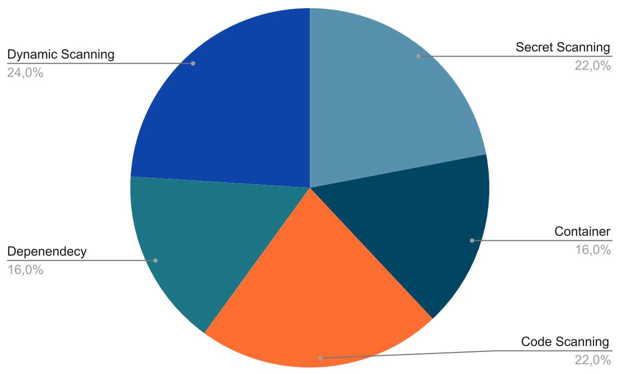

**FIGURE 3.** Distribution of the number of tools by type of application security considered.

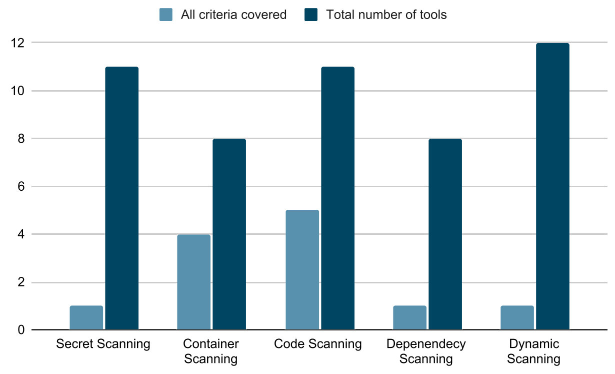

**FIGURE 4.** Number of tools that are recommended by application security scanning type.

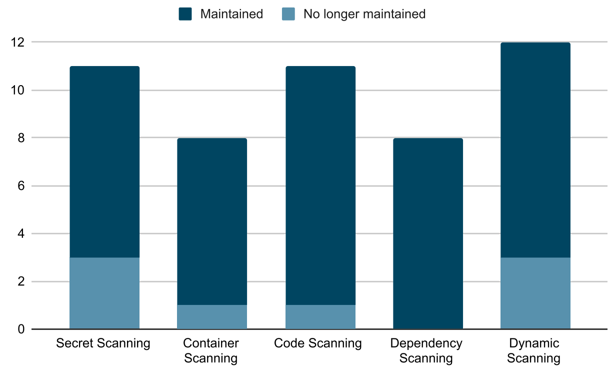

**FIGURE 5.** Number of tools are actively maintained by type of scan.

of the solutions explored, it is important that the information be presented in adequate formats for further integration with other components or for pure user visualization. Figure 6 presents three types of visualization of results that were considered: i) Command-line; ii) Report generation and iii) User interface with dashboards or information and graphs. The results highlight that graphical interface is not highly represented and most solutions have report formats for exporting results (The results contemplate the fact that most solutions will have more than one type of visualization available). Figure 7 considers specific report types and these results are divided by scanning type.

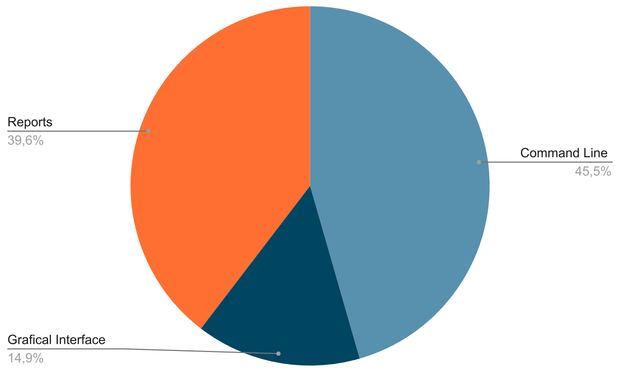

**FIGURE 6.** Percentage of all visualization types provided by all solutions analysed.

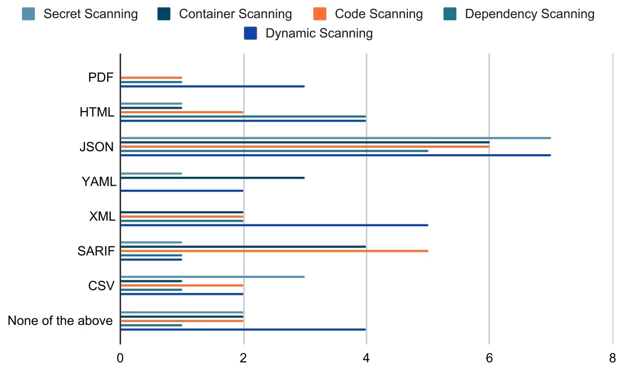

**FIGURE 7.** Types of report formats devided by scanning type.

### A. STATIC APPLICATION SECURITY TESTING (SAST)

SAST tools analyze the source code or compiled versions of the source code to identify security problems. These tools can be integrated with CI/CD pipelines to continuously analyze code and report vulnerabilities, by offering: i) good scalability; ii) the detection of previously known and documented vulnerabilities; and iii) reporting which illustrates the location of the problem for an easier fix. However, not all types of vulnerabilities can be accounted for with static analysis and the level of false positives can be problematic.

#### 1) SECRET SCANNING

Version control platforms like GitHub have grown in popularity and give access to massive amounts of public information, however, not all of this information should be public. For instance, during the development stages, some private information can be added to a commit by mistake. Knothe and Pietschmann[8] created a tool that could harvest emails from public commits and create relationships between no-reply emails and user emails, eventually

[8]https://arxiv.org/pdf/1908.05354.pdf

identifying a person and making them vulnerable to highly targeted phishing and spam attacks. Organizations that use public repositories for collaborative software development, may disclosure private information when making project information public when releasing it as open-source. Secret scanning involves looking for keys, passwords, tokens, and API keys that were inadvertently left inside a repository and can be accessed by people without the proper authorization. Some new approaches include using machine learning to enhance the process and reduce false positives, but most secret detection is made using only regex patterns and looking for patterns with high entropy [54].

Techniques for secret detection were described first by Sinha et al. [55]. These included, GitHub search, regex search, and program flow to make sure the analysis is not happening on meaningless bytes. It also described some mitigation techniques, such as the use of pre-commits and the disallowance of committing certain files. Meli et al. [56], on the other hand, conducted a large-scale analysis on GitHub, which resulted in a massive amount of data that provided insight into the types of secrets being committed by mistake. It created statistics that are worrying, but also provided root cause analysis on these issues while building better mechanisms on top of an already existing solution for secret scanning. An interesting takeaway is the fact that randomness is associated with a good secret, but building structure into these secrets to make them more easily detectable does not compromise their security. Both these studies reported that a committed secret is a far greater problem than it appears, as version control systems keep a history of development and the entire history must be rewritten to remove these secrets. However, the information can be seen as compromised, and caching services can be used to recover these secrets.

Based on the selection and comparison criteria, these tools provide a good overview of what is available. However, some commercial solutions can be more user-friendly but less flexible when it comes to customizing them. While open-source options are more flexible but may lack user-friendly interfaces, they can be easily integrated in various CI/CD platforms. Table 1 illustrates the differences of each solution considered based on the comparison criteria and for a more detailed description of the solutions, the reader is redirected to the Secret Scanning Appendix.

Most solutions considered are command line options, **GitLeaks** and **TruffleHog** show up as the standard solutions for the problem by detecting secrets based on regular expressions and by scanning the repositories history. **Git-Secrets** and **SecretLint** provide a different approach by allowing both a traditional scan as well as the implementation of pre-commit hooks to prevent secrets from being committed to a repository in the first place. **Git-Hound** and **Detect-Secrets** deviate from the standard methodology in that the former bases its design on building a baseline of secrets and reporting updates to that baseline, while the latter enables domain searches in GitHub for use in bug bounty schemes.

False positives, play a big role in secret detection solutions and while solutions like **GittyLeaks** take a more verbose approach to the problem by reporting anything that remotely resembles a secret, **Repo-Supervisor** and **Whispers** provide only a set of languages they can scan to focus the analysis and diminish false positives. Commercial solutions like **GitGuardian** and **SpectralOps** allow for a web interface to show the findings and better integration with the most common CI/CD providers while allowing for the management of multiple projects.

When considering the comparison criteria previously defined and the history search features that **GitLeaks** contains. This tool can cover with efficiency concerns over secret scanning.

## 2) CONTAINER SCANNING

Building secure and reliable containers depends heavily on different factors, one important aspect is following best practices, which are established through collaborative work within the open-source community. Adhering to these practices leads to better-quality images. There are tools that can check containers for compliance with these guidelines, such as the CIS Benchmarks.[9] Container static scanning can also involve checking the software within a container image for known vulnerabilities and, in some cases, attempting to identify malware within the image. While static analysis can be useful for identifying known malware hashes, it is limited in its ability to detect custom programs or study behavior. Static analysis may not be as effective as identifying vulnerabilities using hybrid mechanisms that also incorporate dynamic techniques [57].

Gonzalez-Barahona et al. [58] initially created the idea of technical lag as a technique to gauge how up-to-date is the system. Then, Zerouali et al. [59] applied this concept to containers when considering about third-party software vulnerabilities. Technical lag in this sense refers to the discrepancy between the most recent version of a container and the version that is currently in use. It is also mentioned that package stability frequently takes precedence over security considerations because changing a particular library can result in the product breaking and necessitating more work to repair. Although dependency vulnerabilities can be found via container scanning, it is ultimately up to the developers to determine whether to update the dependencies [58], [59].

There are a wide range of options for container static security, and using a combination of different tools with different objectives can yield better results and better coverage of potential issues. Some solutions, such as **Anchore Engine** and **Clair**, have gained prominence in the literature for being part of implemented continuous integration and delivery CI/CD pipelines [60] and for comparisons in vulnerability reporting [61]. Other solutions, such as Aquasec[10] and

[9]https://www.cisecurity.org/cis-benchmarks/
[10]https://www.aquasec.com

**TABLE 1. Secret scanning solutions comparison, the recommended tool for easy deployment and interoperability is highlighted in grey.**

| | Programming Language | Budget | CI/CD Integration | OWASP Top 10 | SARIF | Framework Understanding | Ease of Setup |
|---|---|---|---|---|---|---|---|
| Detect-Secrets | High | High | Low | ✓ | | - | Medium |
| Git-hound | Low | High | | ✓ | | - | Low |
| Git-secrets | High | High | High | ✓ | | - | Medium |
| GitGuardian | High | Low | High | ✓ | | - | Medium |
| GitLeaks | High | High | High | ✓ | ✓ | - | High |
| GittyLeaks | High | High | Medium | ✓ | | - | Medium |
| Repo-supervisor | Low | High | Medium | ✓ | | - | Medium |
| Secretlint | High | High | High | ✓ | | - | High |
| SpectralOps | High | Low | High | ✓ | | - | Medium |
| TruffleHog | High | High | High | ✓ | | - | High |
| Whispers | Medium | High | High | ✓ | | - | High |

Anchore,[11] offer fully-fledged management systems for cloud infrastructure, which are typically managed by teams of engineers. These systems are often based on smaller tools that can be integrated into other solutions. It's worth noting that container scanning can overlap with software composition analysis SCA as checking for vulnerable packages installed is an important aspect of container scanning.

The tools listed on Table 2 give a overview of which tools are available for dealing with various concerns when using container scanning, based on the selection and comparison criteria. This table illustrates the differences of each solution considered based on the comparison criteria. By analyzing Docker files and running rule checks for bad practices, **Dockle** and **Hadolint** demonstrate the simplest method for scanning containers. **Hadolint** also analyzes the shell code contained in the files. The command line tools, such as **Trivy** and the duo **Grype and Syft**, which were developed by organizations but are widely used, can analyze the image and the packages inside them, reporting on known vulnerabilities for these dependencies.

The architectures of **Clair** and **Dadga** are more complex than those of the prior solutions, but the former allows for the activation of a monitorization mode to report on newly discovered vulnerabilities, while the latter shifts the focus to untrusted images by additionally scanning for trojans and malware within images. **Docker Bench** aims to assist in directing the safe development of Docker containers into production. **Falco** can recognize shells operating inside containers or containers running in privileged mode and find vulnerabilities in a dynamic manner. By taking into consideration the comparison criteria and the different aspects some tools cover. The combination of **Dockle** and **Hadolint** for Dockerfile linting can be one of the best approaches based on this analysis. The use of both allows the identification of different aspects. The use of **Trivy** or **Grype** with **Syft** provides only package scans. **Grype** may also generate some overlap with **Dockle** or **Hadolint** because of a wider coverage of problems.

[11]https://anchore.com

### 3) CODE SCANNING

Code scanning looks for vulnerable patterns within the source code (or compiled code) of a project, such as code smells, anti-patterns, or simple security concerns. Lebanidze [62] shows a conceptual road map for static analysis tools, focusing on the technical aspects of static code analysis rather than the tools that implement them. The first generation of these tools was simple, using basic methods of word matching to report potential problems. However, this led to limited analysis and a high number of false positives. This was built upon with a shift from simple one-man scripts to open-source software created by the industry. Instead of simple word matching, these tools used token construction of the program and separated their vulnerability data set for better knowledge sharing. The knowledge base also expanded to cover more problems, and some tools started supporting multiple languages. One of the first tools to show these second-generation concepts was Lint, developed by Johnson [63] with simple rule checks for the C programming language.

More sophisticated ways of addressing the problem were later developed, introducing the Abstract Syntax Tree for program representation. Some tools will model the program and update the model as they analyze it line by line, producing simpler results than a process that includes the entire project and can correlate different files and instructions. The knowledge base, sharing methods, and capability for multi-language analysis kept growing along with the methods developed. This may raise some questions about of what a program does and how to model its dependencies on other components, which seems to be the direction the industry is taking. However, the remarks about a closer relationship with a code reviewer to address more software flaws rather than bugs themselves do not seem to have taken effect, as the current approach to authorization problems, for example, seems to be addressed using dynamic testing [62].

**TABLE 2.** Container scanning solutions comparison, the recommended tools for easy deployment and interoperability are highlighted in grey.

| | Programming Language | Budget | CI/CD Integration | OWASP Top 10 | SARIF | Framework Understanding | Ease of Setup |
|---|---|---|---|---|---|---|---|
| Clair | High | High | Medium | ✓ | | - | Medium |
| Dadga | High | High | Low | ✓ | | - | Low |
| Docker Bench | High | High | High | ✓ | | - | Medium |
| Dockle | High | High | High | ✓ | ✓ | - | High |
| Falco | | High | Low | ✓ | | - | Medium |
| Grype and syft | High | High | High | ✓ | ✓ | - | High |
| Hadolint | High | High | High | ✓ | ✓ | - | High |
| Trivy | High | High | High | ✓ | ✓ | - | High |

The solutions in Table 3 illustrate tools that report security vulnerabilities for a certain language domain, to the common theme of an integrated solution with access to dashboards. Python and JavaScript are the languages considered here, as the universe is too vast to explore for all programming languages.

Source code analysis can target a variety of issues. For example, **Radon** reports on straightforward problems like code complexity, **PyLint** searches for code smells, anti-patterns, and coding convention issues, and **Bandit** branches out to focus on security issues. In JavaScript, **ESLint** performs similarly to **PyLint**. **Semgrep** advances this strategy by reporting on more intricate vulnerabilities that call for context and is available in multiple languages. To achieve better results, several open-source tools choose to combine other open-source items. **Flake8** creates a straightforward analysis of errors and coding conventions for Python. While **Prospector** builds on this work and adds security considerations (using **Bandit**), **Horusec** offers a comprehensive collection of tools for various languages and covers various aspects of code analysis while also scanning for dependency issues.

Commercial solutions like **DeepSource**, **SonarQube**, and **Codacy** combine proprietary static scanning tools with open-source projects to provide analysis on a variety of languages and access to issues within a management dashboard to view a number of projects. These projects also enable integration with the main CI/CD providers and offer suggested fixes for issues that are discovered.

The project specifications may define in the most part the tools that can be used for code scanning. For overall coverage with a simple solution, **Horusec** provides already integration of a variety of tools. However, the separate usage of **Bandit** and **Semgrep** can give more granular control over the solutions.

## *B. SOFTWARE COMPOSITION ANALYSIS (SCA)*

SCA tools analyze the third-party dependencies of a project and report on vulnerabilities for the versions in use. The analysis is performed using a package manager to identify dependencies. These tools can be integrated with CI/CD pipelines to continuously analyze code and report vulnerabilities. This type of analysis provides a security report on the third-party components in use while some tools offer monitoring and notifications for new vulnerabilities. However, they do not provide an actual risk assessment, and the analysis on large codebases can result in significant technical debt.

The question of whether open-source software is more secure than proprietary software is one that is up for debate, according to Payne [64]. Despite the lack of empirical data in this study, it is generally agreed that open-source software has the potential to be safer, although this is not always the case. Instead, assessments by knowledgeable developers are what eventually results in secure software. Kula et al. [65] identified that the community regularly uses open-source software in their projects without sufficiently considering security risks.

The first approach to creating SCA solutions was focused on reading the software manifest and comparing the packages' versions with a database of vulnerabilities. More recent approaches try to address the excess of false positives produced by this method, with the inclusion of a library as a dependency, this does not mean that vulnerable components are used in the project. Some solutions try to determine if vulnerable components are reachable through static or dynamic analysis of the code, with the best results coming from a combination of both [66], [67].

Table 4 contains some options for SCA, including commercial and open-source solutions that range from standard approaches to improvements on traditional version matching. These solutions are primarily geared towards Python and JavaScript (specifically their package managers). However, some of them support a wide range of programming languages. This table illustrates the differences of each solution considered based on the comparison criteria.

The recovery of dependencies and matching of vulnerable versions to the manifest are common practices used by all solutions; **Satefy** is a well-known example that is exclusively applicable to Python. Since they already have the platform, some dependency managers have developed package audit tools. Examples include **Yarn audit** and **Npm audit**, which enable automatic reporting when a new dependent is added to a project. A single solution that manages multiple package

**TABLE 3.** Code scanning solutions comparison, the recommended tools for easy deployment and interoperability are highlighted in grey.

| | Programming Language | Budget | CI/CD Integration | OWASP Top 10 | SARIF | Framework Understanding | Ease of Setup |
|---|---|---|---|---|---|---|---|
| Bandit | Medium | High | High | ✓ | ✓ | Low | High |
| Codacy | High | Low | High | ✓ | | High | High |
| DeepSource | High | Medium | High | ✓ | | High | Medium |
| ESLint | Medium | High | High | ✓ | ✓ | Medium | High |
| Flake8 | Medium | High | High | | ✓ | Medium | High |
| Horusec | High | High | High | ✓ | ✓ | High | High |
| Prospector | Medium | High | High | ✓ | | Medium | High |
| Pylint | Medium | High | Medium | ✓ | | Medium | Medium |
| Radon | Medium | High | Medium | | | | Medium |
| Semgrep | High | High | High | ✓ | ✓ | High | High |
| SonarQube | High | Low | High | ✓ | | High | Medium |

**TABLE 4.** Dependency scanning solutions comparison, the recommended tool for easy deployment and interoperability is highlighted in grey.

| | Programming Language | Budget | CI/CD Integration | OWASP Top 10 | SARIF | Framework Understanding | Ease of Setup |
|---|---|---|---|---|---|---|---|
| Black Duck | High | Low | High | ✓ | | - | High |
| FOSSA | High | Medium | High | ✓ | | - | High |
| Npm audit | Medium | High | High | ✓ | | - | High |
| OWASP D.C | High | High | High | ✓ | ✓ | - | High |
| Safety | Medium | High | High | ✓ | | - | High |
| SourceClear | High | Low | High | ✓ | | - | High |
| Steady | | High | Medium | ✓ | | - | Medium |
| Yarn Audit | Medium | High | High | ✓ | | - | High |

managers may provide a more comprehensive solution to the problem, as is the case with **OWASP Dependency Check**, which updates itself using the NVD feed, and **FOSSA**, which offers a management system for numerous projects with CI/CD integrations but maintains the same analytical foundations.

Some other approaches include **Steady**'s effort to reduce false positives by reporting only on vulnerabilities where the vulnerable code inside libraries is used within the project. This process produces more accurate results but can lead to issues being missed by a tool since the vulnerability database is far smaller. **Source Clear** is a commercial solution that provides access to a proprietary database of vulnerabilities for further reporting. **Black Duck Software Composition Analysis** can scan third-party components inside the code base to prevent from proprietary code being used where it should not.

The only solution providing software deployment flexibility and interoperability of results is **Dependency Check**. This solution does have the drawback of lack of efficiency as database vulnerabilities are downloaded on each scan.

### *C. DYNAMIC APPLICATION SECURITY TESTING (DAST)*

DAST tools analyze applications by performing actual attacks on a running instance of the solution from a black-box perspective, where the inner workings of the application are not considered. These tools can be integrated with CI/CD pipelines, which involve deploying the application to a testing environment and conducting attacks to gather data on potential vulnerabilities. The attacks use a general approach that may miss specific issues, the root cause of issues is not reported, and these scans take much longer to complete than those of SAST or SCA. However, this type of analysis is language-independent and checks for vulnerabilities that were previously not accounted for such as injection, authentication, or server configuration issues [68].

Dynamic web analysis complements static analysis methods by providing black-box-type tests to simulate attacks. These scans often use fuzzing as a technique to test the application and understand how it responds to invalid, unexpected, or random input. Fuzzing was introduced by Miller et al. [69] with the goal of understanding if the application handles unexpected input properly or not, and what consequences it has on the application. Fuzzing began as a simple technique of submitting random data to a system in order to check for crashes and how the application handles such data. Over time, fuzzing has evolved into two main types: I) mutation-based; and ii) generation-based. Mutation-based fuzzing involves modifying valid input (a seed) and testing the system within a universe of valid inputs. Generation-based fuzzing takes this a step further by creating valid inputs from scratch using a provided model [69].

Originally, the concept of "fuzzing" referred to black-box fuzzing, in which no internal logic was considered. However, other approaches have been developed to automate the process by creating inputs based on the internal logic and source code of a system. These techniques were first introduced by Godefroid et al. [70], [71] in their work on symbolic execution for automated fuzzing. When applied to web applications, fuzzing can be used to produce attacks or variations of attacks such as SQL Injection, XSS, and other types of injections.

An attack's readiness to launch is merely a portion of the issue; their placement must also be taken into account. The configuration can be manual, but most web scanners implement spidering (Web Crawlers) to map an entire application by starting with a simple URL and identifying the hyperlinks that are retrieved from the web pages in a recursive way [72]. Most spidering that was directed at application security, has improved upon the first web crawlers to also identify possible injection points like URL query parameters and POST parameters. With the introduction of AJAX to improve the web experience, spidering became harder as every state change in the application does not necessarily have an associated REST URL. The solutions that load web pages, discover event triggers, and create a module of the changed state when those inputs are triggered have been developed as a response to this problem [73].

The solutions present in Table 5 are only a subset of dynamic analysis, focused on web applications. These solutions are also transversal to all web applications by being language-independent. In most cases, these take a target URL as input and work from that domain to identify other locations. This is done by performing black-box testing injection techniques to produce alerts on possible problems. This table illustrates the differences between each solution considered based on the comparison criteria.

Although **OWASP Zap** shows a strong standard for security dynamic scanning, **Wfuzz** seeks to make the implementation of fuzzing attacks more straightforward while **Arachni** also learns from the modifications it makes within the application. A unified foundation for already created tools is provided by **Golismero**. Other methods take into account many factors, such as the source code of web pages, like **Wapiti** does. **Nikto** is an effective tool for server reconnaissance using banner grabbing, analysis may also be done on the client-side using browser analysis and **The Browser Exploitation Framework**. In order to protect TLS setups and scan for known flaws in the protocol, **Vega** offers many of the same features as **Zap** while adding a strong focus to TLS. **Nogotofail** is exclusively made for this use.

Commercial solutions start scaling these concepts and allow for project management, CI/CD pipeline configuration and monitorization features like **Detectify** provides, **Stack-Hawk** provides much of the same but focuses its analysis on web API testing. By offering a comprehensive perspective of an organization's assets and their weaknesses, **Invicty** brings things to a whole new level of complexity.

**OWASP Zap** provides the only compatibility of results with SARIF, making it the best option for interoperability while also having a solid and repeatable way for deployment.

## IV. DISCUSSION

This section aims to spark the discussion on certain issues and opportunities left unsolved in Application Security. These issues range from the combination of different tools to the vast amount of false positives reported, to the paradoxical concerns with the creation of one true standard for vulnerability information sharing and simplifying the integration of different outputs into a centralised document.

### A. INTEROPERABILITY AND DEPLOYMENT

From the research done, two major requirements are identified for a good and functional solution: i) deployment flexibility; and ii) interoperability. Deployment flexibility refers to the ability of the solution to be deployed and run on any host, regardless of the underlying infrastructure. This becomes relevant to enable the use of the solution in a variety of contexts and environments, making it more widely applicable and useful. Interoperability, on the other hand, refers to the ability of different systems, devices, or applications to work together and exchange information seamlessly. In the context of this solution, interoperability is crucial to facilitate the integration of different solutions and the combination of their results. As we have seen before, Docker offers a platform for software packaging that makes the use of application security tools efficient, consistent, and repeatable in different underlying infrastructures. Many of the solutions explored either have native support for Docker installation (provided by the developer) or have users who have "Dockerized" the application, this can be seen in table 6. This demonstrates the widespread adoption of Docker as a tool for normalizing environments and facilitating the deployment and execution of software.

The use of SARIF enables the uniformization of scan results, providing a variety of options after all scans are completed. It is possible to integrate SARIF with GitHub [74] to present results and display problems. This common format also allows for the production of PDF or HTML documents. Interoperability also allows for better handling of vulnerabilities, including the ability to ignore or remove certain issues. This addresses the problem of False Positives, where issues might be overlooked because of context.

Leveraging the deployment ease and the combination of results, one could create an orchestrator and provide a framework for the integration of AppSec tools that report in SARIF and have been "dockerized". A final pipeline of tools could include, **Horusec** for code scanning, **Dockle**, **Hadolint** and **Trivy** for a big coverage of image issues, **GitLeaks** for secret scanning and repository search history, **Dependency Check** for an analysis on the third-party components the solution relies on, and finally, for some dynamic coverage, **OWASP Zap** should be used.

**TABLE 5. Dynamic scanning solutions comparison, the recommended tool for easy deployment and interoperability is highlighted in grey.**

| | Programming Language | Budget | CI/CD Integration | OWASP Top 10 | SARIF | Framework Understanding | Ease of Setup |
|---|---|---|---|---|---|---|---|
| Arachni | - | High | High | ✓ | | - | Medium |
| BeEF | - | High | | ✓ | | - | Medium |
| Detectify | - | Low | High | ✓ | | - | Medium |
| Golismero | - | High | Medium | ✓ | | - | Low |
| Invicti | - | Low | High | ✓ | | - | Medium |
| Nikto | - | High | Medium | ✓ | | - | Medium |
| Nogotofail | - | High | | ✓ | | - | Low |
| OWASP Zap | - | High | High | ✓ | ✓ | - | Medium |
| Stackhawk | - | Low | High | ✓ | | - | Medium |
| Vega | - | High | | ✓ | | - | Low |
| Wapiti | - | High | High | ✓ | | - | Medium |
| Wfuzz | - | High | | ✓ | | - | Medium |

**TABLE 6. Docker providers per solution that reports in SARIF.**

| | GitLeaks | Hadolint | Dockle | Grype and Syft | Trivy | ESLint |
|---|---|---|---|---|---|---|
| Docker by providers | ✓ | ✓ | ✓ | ✓ | ✓ | |
| Docker by outsiders | | | | | | ✓ |
| | Flake8 | Bandit | Semgrep | Horusec | OWASP Dependecy Check | OWASP Zap |
| Docker by providers | | | ✓ | ✓ | ✓ | ✓ |
| Docker by outsiders | ✓ | ✓ | | | | |

## B. VULNERABILITY MANAGEMENT SYSTEMS

As a natural byproduct of software innovation, most vulnerability assessment systems in use today are restricted to their fields of expertise. Using static code scanning or actual application attacks, the sections on SAST, SCA, and DAST demonstrate several methods for analysis on specific issues.

Currently, there is a critical need for comprehensive vulnerability management solutions on a global level. But some commercial tools provide what are referred to be vulnerability management solutions. These solutions aim to offer a comprehensive assessment of a project's or business' security state, emphasising areas in need of improvement. They aim to develop a prioritisation system, identify the attack surface, and search for potential vulnerabilities across a range of challenges. This system typically rates vulnerabilities based on the threat they pose, from critical, posing a serious risk that requires immediate attention, to low, posing little risk.

These systems' capacity to continuously monitor services and provide results in aesthetically pleasant ways is a major feature. Additionally, some sophisticated solutions might provide automatic correction of identified issues. They can also incorporate data from penetration tests into the list of vulnerabilities found.

One important factor to keep in mind is that these systems gather information and provide it in a way that is simple to understand, making it simpler to comprehend the overall security posture. Prioritising vulnerabilities is essential for efficiently addressing security issues. These systems depend on certain structures built on proprietary languages to ensure interoperability, highlighting the significance of the interoperability debate already explored in this article.

The concept of Global Vulnerability Management Systems is paramount. Such systems use interoperability and deployability concepts and these are essential for enhancing overall cybersecurity practices. As the existing tools confirm, bridging the gap with comprehensive, globally accessible vulnerability management systems remains a vital goal.

## C. INFORMATION OVERLOAD

The information retrieved from vulnerability assessments must be reliable and relevant even though standards provide a common basis for information exchange. While problems are discovered, it is never entirely obvious at first if the discovered material actually poses a cybersecurity risk. This is particularly true for automated scans, which produce findings on *possible* vulnerabilities that are afterwards categorized as either a problem or something to be ignored because they are predicated, for the most part, on pre-defined rules independent of context.

Confusion matrices help visualize how well a particular algorithm performs [75]. In this model, true positives are vulnerabilities identified by automated scans and pose a security problem, false negatives are vulnerabilities not identified that can cause harm and false positives are issues identified by automated tools but in reality do not correspond to actual risk.

False positives in application security testing tools are a problem felt by developers and security engineers in the industry. Tools can be overly verbose in indicating warnings. Static analysis relies on inferring context from the code being analyzed, rather than actually running it. This can lead to false positives. Dynamic analysis is not immune to false positives, but the conceptual architecture of the two approaches is a key factor in their occurrence. False positives can be divided into two types: i) those caused by errors in the scanning tool; and ii) those that are correctly identified but deemed to be non-problems by virtue of the context they are inserted in.

The first type is considered an actual mistake with the intended purposes of the tool performing the analysis. It can happen because some rule triggered when it was not supposed to, if some flow was adopted when that was not the objective, or if it classifies a certain variable as a wrong type. These issues are a design problem from the scanning solution. However, almost all widely used solutions have either eradicated these problems or are in the continuous process of doing it.

The second relates to warnings or errors reported by scanning tools that are later analyzed and deemed not problems because they don't pose a security threat. When evaluated, the security engineer or the developer may flag the issues as non-problematic considering the context around it. One example of this is issues that are mitigated in another place of the project, the tool that discovered the problem could not relate the instance of the error and the mitigation and reported the instance as a problem. Contextual false positives become even more problematic when the analytical tools are used in legacy projects or projects that did not start the development cycle considering this type of analysis.

### D. ONE TRUE STANDARD

SARIF acts as a standard syntax for interoperability in vulnerability reporting. Although its declared scope is limited, it is primarily focused on scanning tools and context-specific data. In the past, various other standards have been proposed, each presenting essential information to share. Although these standards have gained acceptance, using multiple standards creates interoperability challenges, and dilutes the purpose.

The problem is further compounded by the fact that while standards offer syntax interoperability, a structured set of rules for knowledge transmission, the embedded information within this ''grammar'' remains specific to its author, with varying levels of detail and relevant data about the problem.

Standards often encompass diverse data structures to represent different vulnerability types. The issue lies in the optional nature of these fields, as seen in SARIF, where a vulnerability can be reported with minimal information like a message, severity level, and rule identifier. This flexibility makes the quality and quantity of the information hostage to the producer of the report.

The idea of having one true standard for vulnerability reporting is appealing, but it comes with inherent trade-offs. A broad spectrum of information to cover would result in a highly complex specification while reporting information too superficially would render the standard impractical for various scenarios.

To address these challenges effectively, researchers must explore ways to extend and unify vulnerability definition languages. Striking the right balance is essential, enabling comprehensive reporting without overwhelming complexity. By doing so, we can foster better interoperability and information exchange.

### E. TOOLS AS SOLUTIONS

In this study, we identified that by combining specific tools, we are able to detect at earlier stages some of the vulnerabilities reported by ENISA's threat landscape 2022 report [7]. Most of them through dynamic analysis (XSS, SQL injection and CSRF) in a direct way. SAST and SCA can catch issues that are indirectly connected to broader types of vulnerabilities (Incorrect Authorisation, Improper Authentication).

Projects using a version control system can better introduce to these tools in order to produce meaningful results in a repeatable way. If a CI/CD pipeline is already implemented the process should be smoother. However, when this is not the case, one can be implemented to provide a baseline for security and for the development process, including the deployment automation within the project. Smaller projects can use these tools with more specific goals in mind with a one-time use to find problems.

Most solutions cover only some aspects of the broader concepts they encompass. For example, on container scanning solutions, tools like **Dockle** and **Hadolint** provide coverage for Dockerfile linting while Trivy scans for vulnerabilities in installed packages.

All different subdivisions of AppSec topics were discussed to provide a set of tools and a framework for a pipeline to be built with different specifications. This is true for secret scanning as some projects may use git and have that version control system implemented. This enables the history search for secrets. Code Scanning shows better coverage when combining tools that search for formatting problems, look for common anti-patterns scans and also scan for security focussed issues. Dynamic scans can be completed around **Zap** to fill in the gaps which are missing, such as banner grabbing by **Nikto**. Dependency scanning solutions complement each other if they cover different types of package installation systems. To better encapsulate the results the identified tools that report in SARIF give a baseline of a full pipeline for expansion.

### F. THREATS TO VALIDITY

The research conducted is a purposive review and as such does not follow a rigorous method of solution or article selection, while this provides more flexibility and the potential for the exploration of works that falls outside the predefined purpose. For example, the consideration of solutions that cannot work for the project in question but are useful to conceptualize ideas on certain topics. It does have its shortcomings as it offers little assurance of a balanced perspective on the issue. We try to mitigate this issue by creating a set of comparison criteria based on community standards but the issue will always persist.

## V. CONCLUSION

Application security tools play a big role in cybersecurity. To understand and compare these tools effectively, it is essential to evaluate them against a common baseline. These factors include aspects such as CI/CD environment integration, results interoperability, deployment flexibility, and the types of vulnerabilities they report.

The number of options available is extensive, and this article offers a comprehensive analysis and comparison of solutions across different types of Application Security. Through this work, valuable insights are provided, highlighting the shared characteristics among these tools. Such analysis aids the cybersecurity community in selecting appropriate tools for integration.

However, it is important to acknowledge that additional concerns exist, which could serve as future research directions. These concerns revolve around the interoperability formats that exist, the need to address false positives effectively, and the development of an original and open standard for vulnerability reporting. Recognizing these ongoing challenges emphasizes the importance of continued research in the field of application security.

By addressing these concerns and fostering collaboration, we can collectively strive for stronger cybersecurity practices, benefiting organizations and individuals alike.

## APPENDIX A
## SECRET SCANNING APPENDIX

**GitLeaks**[12] is a command-line tool created to detect hard-coded secrets. This detection is based on regular expressions with some generic expressions to catch high entropy strings and some more strict expressions to find known types of secrets. It scans for secrets within the repository's history, meaning secrets that were once present within the repository will also be flagged. However, it involves the use of some sort of version control based on Git, but adds further security as with public software, someone could do a GitHub search and find secrets that are still useful for an attack.

**GittyLeaks**[13] is a command-line tool that takes a much more verbose approach when finding secrets. It is designed to find emails, passwords, and usernames. This tool is much more sensitive to anything that remotely resembles secrets, causing the false positive rate to skyrocket. This is due to generic regex expressions looking for words like "password" or "key", and while this can be useful, it will require much more validation from post-analysis.

**Detect-Secrets**[14] is a command-line tool that takes a different approach. Instead of performing individual scans, this tool hopes to create a baseline of secrets and have the developer audit and fix any issues with the information. This baseline can then be used as a comparison point for future scans. However, implementing this tool with a CI/CD solution can be challenging as this baseline would have to be stored and updated within the environment.

**TruffleHog**[15] is a command-line tool with over 700 credential detectors. Each detector represents a different type of secret that can be found using the tool. The developers of TruffleHog encourage the addition of new detectors by treating them as modules. This tool also has the ability to scan remote repositories and file systems to locate secrets. It will search through the entire history of a repository, across all branches, to ensure that no secrets are left behind, even if they are not currently present in the repository.

**Git-Secrets**[16] is a command-line tool that encourages a more preventive approach to secret management by suggesting the use of pre-commit hooks[17] to prevent secrets from being committed to the repository in the first place. This approach is generally preferred because it is easier to remove secrets from a repository than it is to remove them from its history. Git-Secrets also has a scanning function that can be integrated into CI/CD pipelines. Its detection is made with modular providers that can be installed to create a more minimalist solution, allowing users to install only the providers that are relevant to their specific projects.

**Secretlint**[18] is a command-line tool that offers both scanning and pre-commit options to prevent secrets from being committed to a repository. One key difference between Secretlint and other tools is its flexibility in creating and modifying rules for identifying and accepting secrets. It uses JSON syntax to make it easy to add or remove rules, making the solution modular and allowing users to install only the parts that are relevant to their specific projects. The goal of Secretlint is to make it as easy as possible to customize secret detection and prevention.

**Git-Hound**[19] is a command-line tool that searches exclusively through GitHub content for sensitive information using its search capabilities. Its creator primarily intended it to be used for cashing in on bug bounties, allowing users to search

[12]https://github.com/zricethezav/gitleaks
[13]https://github.com/kootenpv/gittyleaks
[14]https://github.com/Yelp/detect-secrets
[15]https://github.com/trufflesecurity/trufflehog
[16]https://github.com/awslabs/git-secrets
[17]https://pre-commit.com
[18]https://github.com/secretlint/secretlint
[19]https://github.com/tillson/git-hound

within GitHub by domain to narrow down the search space. While Git-Hound still uses regex rules that can be added and modified, it also has a scoring system to distinguish between real secrets and false positives. However, its focus on bug bounties and domain search makes it difficult to integrate into a continuous integration and delivery CI/CD pipeline.

**Repo-Supervisor**[20] created by Auth0,[21] is a tool that can be used on the command line to scan local repositories or integrated with GitHub's webhooks,[22] which send a POST request to a specified endpoint when an event such as a pull request occurs. In both cases, the main goal is to reduce the number of false positives. To achieve this, Repo-Supervisor only allows certain types of files to be scanned (JavaScript, JSON, YAML), and it reads the specified structure of these files rather than just scanning through bytes. This allows it to use language parsing, which helps to reduce the false positive rate by avoiding meaningless bytes.

**Whispers**[23] is a command line tool that supports a variety of languages to be parsed and analysed, the specification of files leaves fewer false positives, it does lack however in special formats of secrets, having the most known credential types but missing a lot of standard formats, it focuses mainly on base64 detection and ASCII, easily detectable secrets by other solutions are not considered because of the simplicity of this solution with it's matching rules. It will also detect secret files like java properties files.

**GitGuardian**[24] is a tool designed to analyze source code and detect secrets based on over 300 known types of secrets. It offers a web interface and integration with GitHub repositories. In addition to finding secrets, it can also identify the active developers involved in the process. The simpler tool for identifying secrets is ggshield,[25] a command-line tool that is based on user authentication through an account created within the GitGuardian framework. It cannot be used independently to detect secrets. This solution is free for small teams or teams using public GitHub repositories, with some restrictions.

**SpectralOps**[26] is a commercial, fully-fledged secret scanning solution that is known for its user-friendly interface, which allows users to manage multiple projects and track all found secrets in each one. It also checks for security misconfigurations. Integrations with CI/CD pipelines can be easily made through the SpectralOps web interface, which serves as a central location for future analysis. SpectralOps offers limited free usage.

[20]https://github.com/auth0/repo-supervisor
[21]https://auth0.com
[22]https://docs.github.com/en/developers/webhooks-and-events/webhooks/about-webhooks
[23]https://github.com/Skyscanner/whispers
[24]https://www.gitguardian.com
[25]https://github.com/GitGuardian/ggshield
[26]https://github.com/zricethezav/gitleaks

## APPENDIX B
## CONTAINER SCANNING APPENDIX

**Hadolint**[27] is a tool that helps build Docker images based on best practices. It is written in Haskell and parses the Dockerfile into an AST, allowing it to perform rule checks on the tree. Hadolint also uses ShellCheck,[28] a bash linter, to check for issues in bash code within the Dockerfile. While the simplicity of Hadolint's specific checks can result in fewer false positives, it may not provide as much information as other tools.

**Dockle**[29] is a simple command-line tool for building Docker images based on best practices. It uses the CIS Benchmarks as checkpoints to report potential problems in the image being analyzed. Dockle, which is written in Go, shares many characteristics with **Hadolint**.

**Falco**[30] consumes kernel events and correlates information from Kubernetes to alert on suspicious behavior. It can detect incidents such as a shell running inside a container, a container running in a privileged mode, a suspicious process spawn, or a suspicious network connection. In addition to being a prevention system, Falco also functions as a continuous monitoring system that can alert in real time.

**Dadga**[31] performs static analysis on docker images/containers to find known vulnerabilities, as well as trojans, viruses, malware, and other malicious threats. This is particularly useful in an uncontrolled environment, where the volatility of images and containers makes it harder to trust that they are free of malware. To run an analysis, Dadga first loads CVE and malware signatures into a MongoDB database, then compares the software within the images to the stored information. However, the solution has many moving components, which can make it difficult to integrate into a CI/CD pipeline.

**Clair**[32] can scan docker images and OCI[33] images for known vulnerabilities. The process consists of three phases: first, the image is indexed and a representation of it is created using the manifest; then, this representation is matched against known vulnerabilities, often by checking for vulnerable versions of installed software; finally (optional if the scan is a one-time occurrence), if the service is running in monitoring mode, Clair will send a notification if a new vulnerability is discovered. Clair has a more integrated approach, with dashboards and a more complex architecture, and the Clair API serves as the front for all communication with the system.

**Docker Bench**[34] has a specialized approach to Docker, using the CIS Docker benchmarks.[35] The command-line tool

[27]https://github.com/hadolint/hadolint
[28]https://github.com/koalaman/shellcheck
[29]https://github.com/goodwithtech/dockle
[30]https://github.com/falcosecurity/falco
[31]https://github.com/eliasgranderubio/dagda
[32]https://github.com/quay/clair
[33]https://github.com/opencontainers/image-spec/blob/main/spec.md
[34]https://github.com/docker/docker-bench-security
[35]https://www.cisecurity.org/benchmark/docker

is simple, with a single script running automated tests to check for the these benchmarks. The results are shown in the terminal, with each check having three possible levels (INFO, WARN, and PASS). The user can then choose to remediate WARN problems that appear, the tool also offers possible solutions for each problem.

**Grype**[36] **and Syft**[37] are command-line tools publicly available for widespread use, created by Anchore.[38] Syft creates SBOM (Software Bill of Materials) files, which are lists of the ingredients that make up the image it analyzes. With or without this file, Grype can then analyze the image using a vast database of known vulnerabilities, comparing them to the software present within the image. The combination of these tools produces a report on the vulnerabilities present within the software and filesystem of the analyzed image.

**Trivy**[39] is a command-line tool created by Aqua Security[40] for finding vulnerabilities and misconfigurations in container images, as well as in the files of a project or operating system, repositories, and Kubernetes. It catches issues such as problems with packages installed on certain container images and dependency problems within the project, and it also allows for the remote scanning of GitHub repositories. Trivy is widely available and shares many characteristics with the **Grype and Syft** combination.

## APPENDIX C
## CODE SCANNING APPENDIX

**Radon**[41] is a simple command-line Python tool that computes metrics about Python code. These metrics are lines of code,[42] McCabe cyclomatic complexity,[43] Halstead metrics,[44] and the maintainability index[45] (a Visual Studio metric that calculates the maintainability of code from 0 to 100).

**PyLint**[46] is a command-line tool that analyzes Python code, supporting versions 3.7.2 and above. It can enforce coding standards while looking for code smells and making suggestions for code refactoring. The tool parses the files and, on the first analysis, can infer values from the code to make the analysis more accurate. It allows for some flexibility when creating new checks for specific uses and relies on community plugins to enhance the experience.

**ESLint**[47] statically analyzes JavaScript code and reports problems found. It can run as a command-line tool, but it has a strong focus on being able to integrate with most major IDEs to make development easier and to catch problems earlier. It offers many options when configuring analysis for a project, and developers can even add comments within the JavaScript code to impact the analysis.

**Flake8**[48] is a wrapper around three tools: PyFlakes[49] which checks for errors within Python code, PyCodeStyle[50] which checks the Python code against the style conventions in PEP 8[51] and Ned Batchelder's McCabe script[52] which checks for the McCabe Cyclomatic Complexity[53] in Python scripts. The scanning done is very simple, which means some problems are left undiscovered, but it does what is intended strictly.

**Bandit**[54] is designed to find security issues in Python code. Each file is processed, and an AST is built and then analyzed to report problems. This tool works within a command-line interface and is installed via pip.

**Prospector**[55] analyzes Python code and outputs information about errors, potential problems, convention violations, and complexity using a command-line interface. It uses a compilation of open-source tools to do this, such as **Bandit** and **PyLint**. Prospector tries to differentiate itself from other competitors by offering customization to reduce the amount of useless information, while also providing some default configurations.

**Semgrep**[56] is a command-line tool that helps finding bugs. It can be used before committing code or during CI/CD pipelines, and it supports many languages, including Python and JavaScript. It will show higher-level vulnerabilities within the files and can detect specific vulnerabilities in known frameworks.

**Horusec**[57] uses other open-source tools to identify security flaws within a code base. This is extendable to other tools with some configuration. This compilation of other solutions covers 20 languages, with the most popular (such as Python, JavaScript, Ruby, Go, and Java) having multiple tools that cover different aspects, including dependencies, security concerns, and basic SAST analysis. Lesser-known languages have only one or two tools associated with them. This makes sense because the fewer they are known/used, the fewer tools are produced to help development with them. However, it is also paradoxical because one could argue that they are less known/used because there is less support for them. Horusec offers many configuration options, and vulnerability ignoring can be associated with a hash to avoid having to ignore each vulnerability with a specific tool, which would leave too

[36]https://github.com/anchore/grype
[37]https://github.com/anchore/syft
[38]https://anchore.com
[39]https://aquasecurity.github.io/trivy/v0.30.4/docs/vulnerability/scanning/image/
[40]https://www.aquasec.com
[41]https://radon.readthedocs.io/en/latest/
[42]https://en.wikipedia.org/wiki/Source_lines_of_code
[43]https://en.wikipedia.org/wiki/Cyclomatic_complexity
[44]https://en.wikipedia.org/wiki/Halstead_complexity_measures
[45]https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022
[46]https://pylint.pycqa.org/en/latest/
[47]https://eslint.org
[48]https://flake8.pycqa.org/en/latest/
[49]https://pypi.org/project/pyflakes/
[50]https://pycodestyle.pycqa.org/en/latest/
[51]https://peps.python.org/pep-0008/
[52]https://github.com/PyCQA/mccabe
[53]https://en.wikipedia.org/wiki/Cyclomatic_complexity
[54]https://github.com/PyCQA/bandit
[55]https://prospector.landscape.io/en/master/index.html
[56]https://semgrep.dev/docs/
[57]https://docs.horusec.io/docs/overview/

many configuration files to be created/understood. Horusec offers a CLI tool, as well as a local installation of a user interface that reflects the results of scans produced.

**DeepSource**[58] has a comprehensive and easy-to-use user interface with direct connections to GitHub, GitLab, and BitBucket, among others. It provides CI/CD features that can be installed within the repository for continuous monitoring of issues. Code scanning relies on smaller open-source tools and their rules, as well as some proprietary scanning. The open-source tools used are **Bandit**, **PyLint**, and **Flake8**. DeepSource also offers automatic fixes, with the option to implement changes with one click from the interface if it has access to the repository. It also offers code formatting options to follow the standard rules of popular formatting styles. While DeepSource is a commercial solution, it has a free plan for extensive individual use or for small teams (<3 members) to get started.

**SonarQube**[59] is a solution with an integrated user interface that focuses on "Code Quality" and "Code Security", using metrics such as code coverage. It also scans for security issues based on the OWASP Top 10 and supports multiple programming languages, including Python, JavaScript, and Java. As a commercial, integrated solution, it offers connections with common community standards for CI/CD workflows, including pull request decoration that connects directly to the user interface for easier analysis. SonarQube has a strong focus on continuous code quality, with new code being constantly analyzed.

**Codacy**[60] offers many of the same features as **SonarQube**, including integration with CI/CD pipelines and security concerns based on the OWASP Top 10. It also supports multiple programming languages, such as Python and JavaScript. However, Codacy's main differentiating point is its focus on automated code review at specific times, rather than constant scanning and analysis like **SonarQube**. Codacy also uses open-source vulnerability assessment tools such as **Bandit** and **Prospector**.

## APPENDIX D
## DEPENDENCY SCANNING APPENDIX

**Safety**[61] is a command-line tool that follows the traditional approach of reading dependency manifests and matching the versions to a database of vulnerabilities to report problems. Safety introduces a policy file that allows developers to track dependency policies, such as ignoring certain vulnerabilities, setting thresholds for alerts, and suppressing exit codes. It is exclusively for Python dependencies and can be used to scan a specific file or path of dependencies within the local environment.

**OWASP Dependency Check**[62] is a command-line interface that retrieves information (evidence) of dependencies in a project and matches them to a CPE.[63] It then lists the CVE entries found for that CPE in the report. OWASP Dependency Check keeps itself updated by using NVD Data Feeds[64] to retrieve new CPE and CVE.

**Yarn Audit**[65] is a vulnerability audit tool integrated with the yarn package manager.[66] It performs a scan on the yarn packages within a project and must be done online. Unlike some other tools, yarn audit uses a non-zero exit code strategy to immediately communicate the types of vulnerabilities found, as the sum of the codes does not produce ambiguous combinations.

**Npm Audit**[67] is a tool integrated with the npm package manager.[68] It submits a description of the dependencies within a project to the npm registry, which then returns a list of known vulnerabilities. Some fixes can be applied automatically, while others require manual intervention.

**Steady**[69] is a Java-specific solution that focuses on reducing false positives by using a code-centric approach that only flags installed libraries as potential issues if their vulnerable components are used or reachable within the code. This approach is limited to a subset of vulnerabilities described in ProjectKB,[70] reducing false positives at the cost of a smaller universe of vulnerabilities to report.

**FOSSA**[71] is a commercial solution for keeping software free of third-party component vulnerabilities with continuous monitoring and other quality-of-life services to help with integration. It supports a wide range of programming languages and integrates with various CI/CD environments and communication and alert platforms. It also offers license compliance scanning and code scanning capabilities to detect issues.

**SourceClear**[72] is a commercial solution developed by Veracode[73] that offers a dependency scanning solution for a variety of programming languages and their most common package managers. It differentiates itself from other solutions by not only reporting on vulnerabilities in the NVD, but also by maintaining a proprietary database of vulnerabilities that are not publicly available. It does this by searching for vulnerabilities in sources such as repositories, metadata, and patch notes. SourceClear also offers a dependency mapping feature to track multiple levels of dependencies.

**Black Duck Software Composition Analysis**[74] is a commercial solution that builds on classical dependency

[58]https://deepsource.io
[59]https://www.sonarqube.org
[60]https://www.codacy.com
[61]https://github.com/pyupio/safety
[62]https://github.com/jeremylong/DependencyCheck
[63]https://nvd.nist.gov/products/cpe
[64]https://nvd.nist.gov/vuln/data-feeds
[65]https://classic.yarnpkg.com/lang/en/docs/cli/audit/
[66]https://yarnpkg.com
[67]https://docs.npmjs.com/cli/v8/commands/npm-audit
[68]https://www.npmjs.com
[69]https://github.com/eclipse/steady
[70]https://github.com/sap/project-kb
[71]https://fossa.com
[72]https://www.veracode.com/blog/managing-appsec/closer-look-veracode-sourceclear-solution
[73]https://www.veracode.com
[74]https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html?intcmp=ref-bds

analysis by also identifying other sources of code, whether they are open-source or proprietary. By tracking these artifacts, it is possible to understand if proprietary code is being used and could pose a risk to the company. This allows for a more sophisticated approach to dependency management, as it tracks not only listed dependencies but also pieces of code or binaries.

## APPENDIX E
## DYNAMIC SCANNING APPENDIX

**Nikto**[75] is an open-source tool written in Perl that has the main objective of scanning web servers for vulnerabilities. This tool reports by checking for outdated versions of more than 1250 servers and can report specific version problems in almost 300 server versions. The number of vulnerabilities reported reaches more than 6,700. This solution will also report on configuration-related issues and SSL certificate scanning. It offers some other features, such as multiple port scanning on a simple web server, scanning with HTTP authentication, scanning through a proxy, and configuring the run according to certain parameters (scan time or some types of scans being excluded).

**OWASP Zap**[76] [76] is an open-source tool oriented towards penetration testing. It is, in essence, a proxy with some added features that help with the pentest process. These features include the ability to intercept and alter requests, active and passive scanners for traditional reporting on known vulnerabilities with standard attacks, traditional spidering, and AJAX spidering which can help identify certain endpoints that are available and possibly should not be. It also has brute force scanners, port scanners, and some features for web sockets. Authentication has also been addressed with the possibility to create a context where we can authenticate as a user and perform automated tasks while logged in. This solution comes with a user interface, but the scanning capabilities have been adapted into docker containers that can be easily deployed for fast and reliable scans. There are already implemented versions, for example, within GitHub Actions.[77]

**Golismero**[78] is an open-source framework for security testing. This solution is platform independent and has no native library dependencies, as the entire solution was written in pure Python (Python2 to be exact). This can make installing harder and, since Python2 is no longer supported, it could cause problems in the future. The command line is intuitive to configure the desired type of scan (adding or removing plugins) and is oriented towards making plugin development as easy as possible. This tool works as a unifier for well-known tools (**Nikto** being one of them, for example). It also integrates standards like CWE, CVE, and OWASP to make the representation broader. The project has not been updated since 2020, which might raise some concerns regarding the maintainability of the solution.

**Wapiti**[79] is an open-source web vulnerability scanner that performs its testing by crawling the web pages and looking through the source code for forms and scripts where injections might occur. This tool acts like a fuzzer but one that also automatically detects entry points to then attack. The objective being centered on injection allows for the reporting of problems like SQL Injections, XSS, Command Injection, or Cross-Site Request Forgery as the most common, or smaller information like unexpected HTTP methods or simple fingerprints of web applications. It allows for reporting in many standard formats with different levels of verbosity.

**Arachni**[80] is an open-source framework designed to help pentesters assess the security of web applications. It covers a wide range of elements, including forms, links, cookies, headers, and request data. In addition to general features like cookie-jar support, proxy authentication, and site authentication, it also offers plugins to report on specific problems within web applications, such as logs on uncommon headers and tracking of cookies. One of the key differentiators of Arachni is its adaptability to the dynamic nature of web applications. When changes are made to the application, the decision tree for the scan is impacted and the solution can learn from the execution flow of the web application. However, Arachni is nearing the end of its life cycle and its next-generation successor, SCNR,[81] is set to take its place.

**Vega**[82] is a free and open-source web application security scanner that scans for common vulnerabilities such as SQL Injections, remote file inclusion, shell injection, and XSS. It also has a major focus on assessing the quality of TLS configurations and can help improve such configurations. Vega has a graphical user interface, which rules out possible integration within CI/CD pipeline. It also has a feature similar to **OWASP Zap**, allowing it to act as a proxy to intercept and alter requests for manual assessment of web applications. This offers a hybrid solution for some automation but also gives freedom to the pentester.

**The Browser Exploitation Framework (BeEF)**[83] is a penetration testing tool that focuses on the web browser. This solution is different from others because it is solely focused on client-side attacks. BeEF does not consider other possibilities but the vulnerabilities present in the web browser. All attacks are launched to access the internal environment of a user (the browser). As a graphical interface application, it hooks a browser to then proceed with information gathering and later with attacks. It also offers, besides the interface, a restful API that allows for scripting within BeEF. This solution is not within the objectives of testing a web application per se, but it is definitely worth

[75]https://github.com/sullo/nikto
[76]https://www.zaproxy.org
[77]https://github.com/marketplace/actions/owasp-zap-baseline-scan
[78]https://github.com/golismero/golismero
[79]https://github.com/wapiti-scanner/wapiti
[80]https://github.com/Arachni/arachni
[81]footnotehttps://ecsypno.com/scnr-documentation/
[82]https://subgraph.com/vega/
[83]https://github.com/beefproject/beef

considering the robustness of the application the user is using to communicate with our solution.

**Wfuzz**[84] is an open-source web security assessment tool that takes a very simple premise very far. This premise is the replacement of the word "FUZZ" with the contents to be fuzzed. This can be applied to URL, POST requests, cookies, custom headers, or even authentication. The "FUZZ" word will be replaced with contents from a file, which means there needs to be a list of values to experiment with to see how the application will react. Even though the tool has a simple premise, some more advanced scenarios can be taken into consideration, such as payload combinations, different encoding combinations, result filtering, and the re-utilization of previous payloads. Working as a command-line tool, this solution is more oriented towards helping a pentester with certain types of payloads and not necessarily with the creation of repeatable fuzzing testing. Although this is possible and could be adapted to work within a CI/CD environment, this implementation would not be trivial and is not repeatable as requests and fuzzing technique will change with each project.

**Nogotofail**[85] is an open-source tool created by Google to test HTTPS connections and make sure that no mistakes were made in the configuration. These mistakes can come from defaults or just plain problems within a manual configuration.[86] This solution aims to catch known vulnerabilities within TLS/SSL connections. With a command-line interface, it is directed to make one attack and see the results, then move on to the next and so forth. It is available for use on Linux and Windows, as well as Android and Chrome OS.

**Detectify**[87] offers an integrated solution based on DAST techniques that allow for web application scanning while also adding a monitoring feature to keep track of changes even when the product is not actively being scanned. The two products provided are *surface monitoring*, which detects exposed files and misconfigurations, and *application scanning*, which finds security vulnerabilities. This solution not only emphasizes making sure the scans are configurable enough for end users, but also proposes a strong integration with CI/CD pipelines to make it as easy as possible to integrate within each company's workflow, including communication platforms, and not only traditional pipeline providers. Detectify also shows a comprehensive solution for testing against subdomain takeovers, in which an attacker claims a subdomain for themselves.[88] These types of problems will become more apparent when considering large-scale organizations, and Detectify, being a paid solution, might be useful for such customers, but an overkill on smaller projects.

[84]https://github.com/xmendez/wfuzz/

[85]https://github.com/google/nogotofail

[86]https://security.googleblog.com/2014/11/introducing-nogotofaila-network-traffic.html

[87]https://detectify.com

[88]https://ieeexplore.ieee.org/abstract/document/8679122

**StackHawk**[89] offers an integrated solution for web applications, but also shows some emphasis on web API testing, considering it an important part of web applications and services in a company. Another interesting feature that differs from other solutions is the possibility to add security within the CI/CD process. This is beyond the scope of DAST, to an extent (since we are targeting DAST for web applications), but it can be an interesting concern to tackle for more mature companies looking to add further security in a more in-depth approach, instead of securing only the perimeter (in-depth defense). StackHawk offers a free version with limited features for only 1 application.

**Invicti**[90] offers an integrated solution for web applications and services, but starts the process by identifying all web assets present within the company. This approach only makes sense for a magnitude of products that most companies will not achieve. The process this solution offers also targets root cause analysis of problems. DAST has the problem of working isolated from the code, and even though problems are found, the next step, mostly manual, will be to identify where the problem came from so it can be resolved. Invicti also employs IAST techniques to produce more detailed reports on problems and reduce the fixing time of an issue by cutting down the root cause analysis time.

## REFERENCES

[1] A. Chidukwani, S. Zander, and P. Koutsakis, "A survey on the cyber security of small-to-medium businesses: Challenges, research focus and recommendations," *IEEE Access*, vol. 10, pp. 85701–85719, 2022.

[2] S. Backman, "Organising national cybersecurity centres," *Inf. Secur., Int. J.*, vol. 32, pp. 9–26, 2015.

[3] Portuguese National Cybersecurity Centre. (2019). *Quadro Nacional de referência Para a Cibersegurança*. Accessed: Feb. 28, 2023. [Online]. Available: https://www.cncs.gov.pt/docs/cncs-qnrcs-2019.pdf

[4] T. W. Thomas, M. Tabassum, B. Chu, and H. Lipford, "Security during application development: An application security expert perspective," in *Proc. CHI Conf. Human Factors Comput. Syst.*, Apr. 2018, pp. 1–12.

[5] C. Nobles, "Botching human factors in cybersecurity in business organizations," *HOLISTICA J. Bus. Public Admin.*, vol. 9, no. 3, pp. 71–88, Dec. 2018.

[6] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, Sep. 2010, pp. 447–456.

[7] ENISA. (Nov. 2022). *Enisa Threat Landscape 2022*. Accessed: Feb. 27, 2023. [Online]. Available: https://www.enisa.europa.eu/publications/enisa-threat-landscape-2022/@@download/fullReport

[8] M. Soni, "Defect prevention: Reducing costs and enhancing quality," IBM, Armonk, NY, USA, Tech. Rep., 2006. [Online]. Available: https://iSixSigma.com

[9] A. Apvrille and M. Pourzandi, "Secure software development by example," *IEEE Secur. Privacy*, vol. 3, no. 4, pp. 10–17, Jul. 2005.

[10] ENISA. (Nov. 2022). *Etl2020—Web Application Attacks*. Accessed: Feb. 27, 2023. [Online]. Available: https://www.enisa.europa.eu/publications/web-application-attacks

[11] M. Curphey and R. Arawo, "Web application security assessment tools," *IEEE Secur. Privacy Mag.*, vol. 4, no. 4, pp. 32–41, Jul. 2006.

[12] A. Alzahrani, A. Alqazzaz, Y. Zhu, H. Fu, and N. Almashfi, "Web application security tools analysis," in *Proc. IEEE 3rd Int. Conf. Big Data Secur. Cloud (Bigdatasecurity) Int. Conf. High Perform. Smart Comput. (HPSC), Int. Conf. Intell. data Secur. (IDS)*, May 2017, pp. 237–242.

[89]https://www.stackhawk.com

[90]https://www.invicti.com

[13] R. Amankwah, P. Kwaku, and S. Yeboah, ''Evaluation of software vulnerability detection methods and tools: A review,'' *Int. J. Comput. Appl.*, vol. 169, no. 8, pp. 22–27, Jul. 2017.

[14] J. R. B. Higuera, J. B. Higuera, J. A. S. Montalvo, J. C. Villalba, and J. J. N. Pérez, ''Benchmarking approach to compare web applications static analysis tools detecting OWASP top ten security vulnerabilities,'' *Comput., Mater. Continua*, vol. 64, no. 3, pp. 1555–1577, 2020.

[15] G. Hao, F. Li, W. Huo, Q. Sun, W. Wang, X. Li, and W. Zou, ''Constructing benchmarks for supporting explainable evaluations of static application security testing tools,'' in *Proc. Int. Symp. Theor. Aspects Softw. Eng. (TASE)*, Jul. 2019, pp. 65–72.

[16] L. K. Seng, N. Ithnin, and S. Z. M. Said, ''The approaches to quantify web application security scanners quality: A review,'' *Int. J. Adv. Comput. Res.*, vol. 8, no. 38, pp. 285–312, Sep. 2018.

[17] H. H. AlBreiki and Q. H. Mahmoud, ''Evaluation of static analysis tools for software security,'' in *Proc. 10th Int. Conf. Innov. Inf. Technol. (IIT)*, Nov. 2014, pp. 93–98.

[18] G. McGraw, ''Software security,'' *IEEE Secur. Privacy*, vol. 2, no. 2, pp. 80–83, Aug. 2004.

[19] V. Clincy and H. Shahriar, ''Web application firewall: Network security models and configuration,'' in *Proc. IEEE 42nd Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 1, Jul. 2018, pp. 835–836.

[20] P. Cisar and S. M. Cisar, ''The framework of runtime application self-protection technology,'' in *Proc. IEEE 17th Int. Symp. Comput. Intell. Informat. (CINTI)*, Nov. 2016, pp. 000081–000086.

[21] N. Imtiaz, S. Thorn, and L. Williams, ''A comparative study of vulnerability reporting by software composition analysis tools,'' in *Proc. 15th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Oct. 2021, pp. 1–11.

[22] J. Yang, L. Tan, J. Peyton, and K. A Duer, ''Towards better utilizing static application security testing,'' in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng., Softw. Eng. Pract. (ICSE-SEIP)*, May 2019, pp. 51–60.

[23] F. M. Tudela, J.-R. B. Higuera, J. B. Higuera, J.-A. S. Montalvo, and M. I. Argyros, ''On combining static, dynamic and interactive analysis security testing tools to improve OWASP top ten security vulnerability detection in web applications,'' *Appl. Sci.*, vol. 10, no. 24, p. 9119, Dec. 2020.

[24] Y. Pan, ''Interactive application security testing,'' in *Proc. Int. Conf. Smart Grid Electr. Autom. (ICSGEA)*, Aug. 2019, pp. 558–561.

[25] J. Gao, X. Bai, W.-T. Tsai, and T. Uehara, ''Mobile application testing: A tutorial,'' *Computer*, vol. 47, no. 2, pp. 46–55, Feb. 2014.

[26] R. Shirey, *Internet Security Glossary, Version 2*, Informational, document RFC 4949, Aug. 2007.

[27] *Software Weaknesses, 2022*, CWE, MITRE, Bedford, MA, USA, 2022.

[28] *Owasp Top 10, 2022*, OWASP Foundation, College Park, MD, USA, 2022.

[29] K. Rantos, A. Spyros, A. Papanikolaou, A. Kritsas, C. Ilioudis, and V. Katos, ''Interoperability challenges in the cybersecurity information sharing ecosystem,'' *Computers*, vol. 9, no. 1, p. 18, Mar. 2020.

[30] S. Barnum, ''Standardizing cyber threat intelligence information with the structured threat information expression (STIX),'' *Mitre Corp.*, vol. 11, pp. 1–22, Jan. 2012.

[31] J. Banghart, S. Quinn, and D. Waltermire, ''Open vulnerability assessment language (OVAL) validation program derived test requirements,'' Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep., 2010.

[32] A. Randal, ''The ideal versus the real: Revisiting the history of virtual machines and containers,'' *ACM Comput. Surveys*, vol. 53, no. 1, pp. 1–31, Jan. 2021.

[33] V. Singh and S. K. Peddoju, ''Container-based microservice architecture for cloud applications,'' in *Proc. Int. Conf. Comput., Commun. Autom. (ICCCA)*, May 2017, pp. 847–852.

[34] D. Bernstein, ''Containers and cloud: From LXC to Docker to kubernetes,'' *IEEE Cloud Comput.*, vol. 1, no. 3, pp. 81–84, Sep. 2014.

[35] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, ''An updated performance comparison of virtual machines and Linux containers,'' in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2015, pp. 171–172.

[36] C. Anderson, ''Docker [software engineering],'' *IEEE Softw.*, vol. 32, no. 3, pp. 102–c3, May 2015.

[37] T. Combe, A. Martin, and R. D. Pietro, ''To Docker or not to docker: A security perspective,'' *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 54–62, Sep. 2016.

[38] S. Sultan, I. Ahmad, and T. Dimitriou, ''Container security: Issues, challenges, and the road ahead,'' *IEEE Access*, vol. 7, pp. 52976–52996, 2019.

[39] Y. B. Leau, W. K. Loo, W. Y. Tham, and S. F. Tan, ''Software development life cycle agile vs traditional approaches,'' in *Proc. Int. Conf. Inf. Netw. Technol.*, vol. 37, 2012, pp. 162–167.

[40] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, ''DevOps,'' *IEEE Softw.*, vol. 33, no. 3, pp. 94–100, May 2016.

[41] K. Beck, ''Embracing change with extreme programming,'' *Computer*, vol. 32, no. 10, pp. 70–77, 1999.

[42] M. Fowler and M. Foemmel, ''Continuous integration,'' Tech. Rep., 2006.

[43] L. Chen, ''Continuous delivery: Huge benefits, but challenges too,'' *IEEE Softw.*, vol. 32, no. 2, pp. 50–54, Mar. 2015.

[44] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö, ''The highways and country roads to continuous deployment,'' *IEEE Softw.*, vol. 32, no. 2, pp. 64–72, Mar. 2015.

[45] M. Shahin, M. A. Babar, M. Zahedi, and L. Zhu, ''Beyond continuous delivery: An empirical investigation of continuous deployment challenges,'' in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Nov. 2017, pp. 111–120.

[46] M. Virmani, ''Understanding DevOps & bridging the gap from continuous integration to continuous delivery,'' in *Proc. 5th Int. Conf. Innov. Comput. Technol. (INTECH)*, May 2015, pp. 78–82.

[47] F. Zampetti, S. Geremia, G. Bavota, and M. Di Penta, ''CI/CD pipelines evolution and restructuring: A qualitative and quantitative study,'' in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2021, pp. 471–482.

[48] A. M. Mowad, H. Fawareh, and M. A. Hassan, ''Effect of using continuous integration (CI) and continuous delivery (CD) deployment in DevOps to reduce the gap between developer and operation,'' in *Proc. Int. Arab Conf. Inf. Technol. (ACIT)*, Nov. 2022, pp. 1–8.

[49] M. Meyer, ''Continuous integration and its tools,'' *IEEE Softw.*, vol. 31, no. 3, pp. 14–16, May 2014.

[50] T. Górski, ''Continuous delivery of blockchain distributed applications,'' *Sensors*, vol. 22, no. 1, p. 128, Dec. 2021.

[51] N. K. Tran, M. A. Babar, and A. Walters, ''A framework for automating deployment and evaluation of blockchain networks,'' *J. Netw. Comput. Appl.*, vol. 206, Oct. 2022, Art. no. 103460.

[52] M. Shahin, M. A. Babar, and L. Zhu, ''The intersection of continuous deployment and architecting process: Practitioners' perspectives,'' in *Proc. 10th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, Sep. 2016, pp. 1–10.

[53] OWASP Foundation. (2023). *Owasp Source Code Analysis Tools*. [Online]. Available: https://owasp.org/www-community/Source_Code_Analysis_Tools

[54] A. Saha, T. Denning, V. Srikumar, and S. K. Kasera, ''Secrets in source code: Reducing false positives using machine learning,'' in *Proc. Int. Conf. Commun. Syst. Netw. (COMSNETS)*, Jan. 2020, pp. 168–175.

[55] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani, ''Detecting and mitigating secret-key leaks in source code repositories,'' in *Proc. IEEE/ACM 12th Work. Conf. Mining Softw. Repositories*, May 2015, pp. 396–400.

[56] M. Meli, M. R. McNiece, and B. Reaves, ''How bad can it git? Characterizing secret leakage in public Github repositories,'' in *Proc. NDSS*, 2019, pp. 1–15.

[57] O. Tunde-Onadele, J. He, T. Dai, and X. Gu, ''A study on container vulnerability exploit detection,'' in *Proc. IEEE Int. Conf. Cloud Eng. (ICE)*, Jun. 2019, pp. 121–127.

[58] J. M. Gonzalez-Barahona, P. Sherwood, G. Robles, and D. Izquierdo, ''Technical lag in software compilations: Measuring how outdated a software deployment is,'' in *Proc. IFIP Int. Conf. Open Source Syst.* Cham, Switzerland: Springer, 2017, pp. 182–192.

[59] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, ''On the relation between outdated Docker containers, severity vulnerabilities, and bugs,'' in *Proc. IEEE 26th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Feb. 2019, pp. 491–501.

[60] K. Brady, S. Moon, T. Nguyen, and J. Coffman, ''Docker container security in cloud computing,'' in *Proc. 10th Annu. Comput. Commun. Workshop Conf. (CCWC)*, Jan. 2020, pp. 0975–0980.

[61] M. Jagelid, ''Container vulnerability scanners: An analysis,'' M.S. thesis, KTH Roy. Inst. Technol., Stockholm, Sweden, 2020.

[62] E. Lebanidze, ''The need for fourth generation static analysis tools for security–from bugs to flaws,'' in *Proc. Appl. Secur. Conf.*, 2008, pp. 1–7.

[63] S. C. Johnson, "Lint, a C program checker," Bell Telephone Laboratories, Murray Hill, New York, NY, USA, Tech. Rep., 1977.
[64] C. Payne, "On the security of open source software," *Inf. Syst. J.*, vol. 12, no. 1, pp. 61–78, 2002.
[65] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Softw. Eng.*, vol. 23, no. 1, pp. 384–417, 2018.
[66] D. Foo, J. Yeo, H. Xiao, and A. Sharma, "The dynamics of software composition analysis," 2019, *arXiv:1909.00973*.
[67] S. E. Ponta, H. Plate, and A. Sabetta, "Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2018, pp. 449–460.
[68] S. Alazmi and D. C. De Leon, "A systematic literature review on the characteristics and effectiveness of web application vulnerability scanners," *IEEE Access*, vol. 10, pp. 33200–33219, 2022.
[69] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.
[70] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," Tech. Rep., 2008.
[71] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, pp. 40–44, 2012.
[72] M. AbuKausar, V. S. Dhaka, and S. Kumar Singh, "Web crawler: A review," *Int. J. Comput. Appl.*, vol. 63, no. 2, pp. 31–36, Feb. 2013.
[73] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling ajax-based web applications through dynamic analysis of user interface state changes," *ACM Trans. Web*, vol. 6, no. 1, pp. 1–30, Mar. 2012.
[74] P. Anderson, L. Kot, N. Gilmore, and D. Vitek, "SARIF-enabled tooling to encourage gradual technical debt reduction," in *Proc. IEEE/ACM Int. Conf. Tech. Debt (TechDebt)*, May 2019, pp. 71–72.
[75] S. V. Stehman, "Selecting and interpreting measures of thematic classification accuracy," *Remote Sens. Environ.*, vol. 62, no. 1, pp. 77–89, Oct. 1997.
[76] S. Bennetts, "OWASP zed attack proxy," Tech. Rep., 2013.

**DINIS BARROQUEIRO CRUZ** received the bachelor's degree in informatics engineering from the University of Aveiro, where he is currently pursuing the master's degree in cybersecurity. Since April 2022, he has been collaborating on a research grant by taking part in the secure development of the EHDEN portal. His research interests include application security, vulnerability assessment, and cryptographic applications to the medical field.

**JOÃO RAFAEL ALMEIDA** is currently a Researcher in the cybersecurity field. He is also a member of the Coordination Team of the Cybersecurity Office, University of Aveiro. He has authored or coauthored over 40 scientific papers in peer-reviewed journals and conferences, and coordinated/participated in more than ten research projects. His research interests include data privacy, data engineering, threat discovery, and cybersecurity in general.

**JOSÉ LUÍS OLIVEIRA** is currently the Director of the Institute of Electronics and Informatics Engineering of Aveiro (IEETA) and a Full Professor with the Department of Electronics, Telecommunications and Informatics (DETI), University of Aveiro. He has authored or coauthored over 300 scientific papers in peer-reviewed journals and conferences, and coordinated/participated in more the 40 research projects. His research interests include data engineering, text mining, distributed systems, and computational methods in biomedicine.