Memo

www.tno.nl

Send to Software development and testing communities

Author Thijs Klooster, Swarna Kumarswamy- Das, Thomas Rooijakkers, Bert Jan te Paske

Date
05-06-2024
Our reference
TNO 2024 M10313

Subject Software Security Testing Techniques and Tools

1. Introduction

1.1. Goal and scope

The importance of cybersecurity in our digitalising society is nowadays well-understood. In practice however, cybersecurity is too often an afterthought. Countermeasures are taken in reaction to vulnerability exposures and cyber incidents as they happen. A transition towards inherently cyber-resilient systems starts with early and systematic testing of these systems and the software that drives them. Software security testing is complementary to regular (functional) software testing and focuses specifically on discovering security risks and vulnerabilities.

This memo summarises the field of software security testing, providing a comprehensive overview of the techniques and tools employed within this field of testing. It aims to make the connection between current practice and newer technologies. This is important as newer technologies can greatly improve the effectiveness and efficiency of current software security testing techniques. In particular, by applying smart automation they can reduce the human effort and required expertise for testing. This lowers the threshold and builds a business case for software security testing.

In this document, the capabilities and characteristics of technologies are discussed on a qualitative level, based on desk research and experience available within TNO. No extensive experimental assessment was performed to verify all functionalities or acquire performance metrics.

1.2. Intended audience

This document is primarily aimed at professionals in software development and testing. They may adopt the described techniques and tools to enhance the quality and efficiency of test pipelines. This includes complementing functional testing with testing for security-related vulnerabilities.

Other audiences include security professionals and IT staff interested in DevSecOps and “shift-left security” in the software development lifecycle.

1.3. Outline

Section 2 describes background knowledge related to software security testing, Section 3 provides an overview of software security testing techniques, Section 4 provides an in-depth analysis of several software security testing tools, Section 5 discusses our findings, compares capabilities and limitations of all software security testing techniques, their usability, and the challenges within the field of software security testing. Section 6 concludes the memo with key take-aways, results, and recommendations.

2. Background

Software security testing is only a subset of the broad field of software testing, and therefore we briefly summarise the overall software testing field. We define software security testing, and we discuss the (secure) software development lifecycle. Finally, we touch upon the shift-left security paradigm.

2.1. Software Testing

The goal of software testing is to find errors in software. The offensive motivation behind this is the exploitation of these defects. The defensive motivation is to be able to fix them to achieve a state of the application that is robust and fit for use. Software testing is performed to verify that an application does exactly what it is supposed to do. Software testing results in improved performance, bug prevention, and reduced development costs. There are several software testing types: unit-, integration-, regression-, functional-, performance-, usability-, stress-, and acceptance testing. Quality Assurance (QA) is the broad process of planning, design, creation, and execution of tests and test environments. Software testing can prevent architectural flaws, scalability issues, incorrect functionality, poor design decisions, and security vulnerabilities. Two decades ago software testing was done after a certain build was made, nowadays it is incorporated much earlier in the development lifecycle, and more often; this is known as continuous testing. Continuous testing is an automated process that is integrated into the development lifecycle, with the goal of validating new revisions of the software to improve its design, reliability, robustness, and to reduce risks early on.

2.2. Software Security Testing

Software security testing is a form of testing with the specific goal of discovering potential security-related flaws or vulnerabilities in the software being tested. The aim is to prevent defects from enabling (malicious) users to obtain sensitive information, gain unauthorised access to (parts of) the system, or influence the system in a way that alters the intended behaviour. There are several software security testing techniques, which are discussed in further detail in Section 3.

2.3. Secure Software Development LifeCycle (SSDLC)

The Software Development LifeCycle (SDLC) is a sequence of stages in the process of going from a concept to a software deliverable. This lifecycle can be used to deliver anything from smaller features up to large and complex systems. The SDLC comprises of a framework that provides a better understanding of requirements, early issues identification, cost reduction, and higher software quality delivery. The phases of such a cycle include collecting requirements, analysis, design, development, testing, deployment, and maintenance. These phases can be executed linearly, where we move to the next stage after the current one is finished. This is often used for mission-critical systems (that have the highest risk in terms of damages when something goes wrong). However, for other kinds of systems the requirements may change over time, and so this can lead to changes in design and development. This is one of the main reasons why the Agile SDLC was created, which focuses on quick iterations with smaller deliverables with a higher frequency. This means that phases of the SDLC can occur in parallel, and the cost of changes will be smaller.

Software testing and software security testing have their own place in the SDLC. As is the industry standard, Continuous Integration / Continuous Delivery (CI/CD) pipelines are used to integrate components of software developed by different people, and to deliver newer builds of the system to a client or a live environment. Such pipelines consist of several stages, including plan, code, build, test, release, deploy, operate, and monitor. Software testing techniques can be automated such that they can be automatically executed as part of the development pipeline. The software can be continuously tested, providing valuable feedback to developers that can fix any issues as soon as possible. This holds true for security testing as well; certain security testing tools execute during the coding stage of a pipeline, some execute during the testing stage, and others execute during the deployment stage.

2.4. Shift-Left Security

The integration of automated security testing within the CI/CD pipeline significantly benefits quality, robustness, and resilience aspects of the software under test. Instead of employing (security) testing at a stage in the SDLC where a new build is ready to be deployed, testing is done much earlier in the lifecycle. This way, defects are discovered sooner, such that they are easier to fix. For security testing, this results in two main benefits; patching of vulnerabilities is cheaper, and the risk of exploitation is a lot lower. This paradigm of incorporating security from the start of the SDLC is known as shift-left security. Both manual and automated software security testing techniques are fundamental in secure software development practices, especially when focusing on shift-left security.

3. Software Security Testing Techniques

Software security testing techniques can be placed into several categories. This section summarises each of these categories, and per category the capabilities and limitations are discussed, along with when to use the specific techniques. Additionally, the current state-of-the-art and possible trends are outlined. Lastly, we aim to answer the question of how commonly a technique is used in practice, and which are the most used tools within the respective category.

3.1. Manual Code Review

Manual code review is checking source code thoroughly for logic errors, errors in the implementation of the specification, portions of code that divert from style guidelines, and potential other flaws or security vulnerabilities. Manual code review can be done at any point in the SDLC; at every commit[1], at every merge request[2], or at every release[3].

3.1.1. When to use

Manual code review is best used with each merge request. Automated code review is best used while developing, within the IDE (An Integrated Development Environment enables programmers to bring together different aspects of their code) or with every commit. This way, manual and automated code review nicely complement one another. Positive side-effects of manual code review include improved collaboration, knowledge sharing, maintainability, and developer productivity. Organisations employing manual quality assurance should include verification of security features and mitigations within the test plan [1].

3.1.2. Capabilities

Manual code review can take the intentions of the developer and general business logic into account, while automated code review processes cannot. Manual code review investigates specific issues and is overall more strategic. An automated code review process can easily miss flaws that will be flagged by manual code review. Manual code review can prevent late-stage defects, saving time and resources required to fix those defects later.

3.1.3. Limitations

Manual code review generally requires a great time investment; the automated counterpart is much faster. Additionally, manual code review requires an experienced developer that is familiar with the software and the corresponding requirements and design decisions. Also, manual code review can be subjective, and prone to human error.

3.1.4. State-of-the-art & possible trends

Best practices include both automated and manual code review to enforce the highest code quality and security standards possible. Manual review combined with the findings of automated tools like Static Application Security Testing (SAST) can significantly improve the security of applications. Standards like the OWASP Application Security Verification Standard can greatly aid the verification process of an application's security status [2]. Machine learning can be applied to identify code fragments that require manual review, improving the speed of reviews [3, 4].

[1] Git commit captures a snapshot of the project (https://www.atlassian.com/git/tutorials/saving-changes/git-commit)
[2] Merge request is a scheme to incorporate changes from a particular branch to another targeted branch (https://docs.gitlab.com/ee/user/project/merge_requests/)
[3] Releases helps to create a snapshot of the project for (internal or external) users (https://docs.gitlab.com/ee/user/project/releases)

3.1.5. How commonly used in practice

Manual code review is used almost everywhere with software projects that have a team of developers instead of a single developer. It is most often applied at every merge request.

3.1.6. Most commonly used tools

Manual code review tools that are most used in practice include GitLab[4], GitHub[5], BitBucket[6], and Azure DevOps[7].

3.2. Manual Security Testing (penetration testing / red teaming)

Manual security testing is when security experts assess an application with respect to security. They look at the source code, the configurations of the software, and its functionalities to discover potential security risks and vulnerabilities. They may use several different tools or proprietary scripts that aid in this regard.

3.2.1. When to use

Manual security testing should always be used when it concerns critical or high-risk software. For non-critical applications that process sensitive data, it is still advised to employ manual security testing. It also proves valuable with complex applications that are quite hard to test using automated tools. When specific security requirements are in place, manual security testing may no longer be optional. To get an overall risk assessment, manual security testing is also wise.

3.2.2. Capabilities

Manual security testing is effective due to human intelligence, expertise, and creativity. It can discover the more complex vulnerabilities that are potentially hiding in deep code paths. It also provides a realistic simulation of potential attacks. Manual security testing can discover lots of issues that are missed by automated security testing tools.

3.2.3. Limitations

Manual security testing can be very expensive and time-consuming. It is also prone to human error and subjectivity, and it is highly dependent on the availability of people with the required security expertise. Penetration testing is the most laborious and time-consuming form of security testing, requires specialised expertise, scales poorly and is challenging to quantify and measure [1].

3.2.4. State-of-the-art & possible trends

The state-of-the-art in manual security testing lies with companies that are specialised in this area. Companies, may look into machine learning and AI to automate certain processes or make them more efficient [5–8].

3.2.5. How commonly used in practice

Depending on the security requirements for the software under test, companies may or may not decide to hire an external security assessment company to test new releases of the software for security risks and vulnerabilities. For software with a lower requirement for security, in-house penetration testing can also be done. In practice, however, such advanced security assessments are not held that often (except for critical systems).

[4] https://about.gitlab.com/
[5] https://github.com/
[6] https://bitbucket.org/product/
[7] https://azure.microsoft.com/nl-nl/products/devops

3.2.6. Most commonly used tools

Manual security testing tools that are most used in practice include Tenable Nessus[8], Burp Suite[9], IDA Pro[10], OWASP Nettacker[11] / OWTF[12] / Amass[13], and proprietary scripts and software.

3.3. Functional Testing of Security Features

Functional testing of security features includes the planning and execution of unit- and integration tests aimed specifically at the implementations of the security functionality within the target application. Additionally, regression tests may be added to verify that previous bugs do not resurface, and code changes do not trigger new security issues in the original codebase.

3.3.1. When to use

Organisations that use unit tests and other automated testing to verify the correct implementations of general features and functionality should extend those testing mechanisms to verify security features and mitigations designed into the software [1].

3.3.2. Capabilities

Verification of security features is a continuous process, and if a new revision of the software breaks the security functionality in some way, developers will be made aware immediately such that the issue can be fixed as early on as possible. This technique can also prevent regressions of previous bugs.

3.3.3. Limitations

This technique is non-exhaustive, and it is only as effective as the tests that are written. Advanced tests that thoroughly exercise the complete security functionality are required to take full advantage of this testing technique. Another limitation while performing functional testing is bias, which can be a hidden pitfall as the author of those tests may not conceive of some potential errors, which will consequently not be covered by test cases.

3.3.4. State-of-the-art & possible trends

While unit tests are often written by hand (ideally when some new functionality is added to the application, and the tests for it are written simultaneously), there is a research area called Search-Based Software Testing (SBST), which focusses on automated generation of test suites [9]. SBST optimises (the generation of) test suites, such that they are as efficient as possible (requiring less time and computational resources), hence contributing towards sustainable ICT. Test case prioritisation, test suite minimisation, and optimising test oracles (mechanisms to determine whether tests have failed or passed) are the main objectives of SBST [10]. The concept of continuous test generation (integrating automated test generation into CI/CD pipelines [11]) has led to tools like EvoSuite [12] and Pex [13].

3.3.5. How commonly used in practice

Unit tests are commonly used in practice, and security functionality is often addressed by some of the tests as well. However, this does not typically concern advanced tests that thoroughly exercise the complete security functionality, but instead only cover the basics of the security functionality. Regression tests are not that commonly included either, although it can greatly benefit application security.

[8] https://www.tenable.com/products/nessus
[9] https://portswigger.net/burp
[10] https://hex-rays.com/ida-pro/
[11] https://owasp.org/www-project-nettacker/
[12] https://owasp.org/www-project-owtf/
[13] https://owasp.org/www-project-amass/

3.3.6. Most commonly used tools

Tools for functional testing of security features that are most used in practice include JUnit[14], JBehave[15], NUnit[16], xUnit[17], Robot[18], and PyTest[19].

3.4. Software Composition Analysis (SCA)

Software Composition Analysis (SCA) is an automated software security testing technique that looks for versions of libraries and other third-party pieces of software (dependencies) that have known (licensing) issues or vulnerabilities associated with them. If such instances are found in a project, it is best to upgrade the respective dependency to a new (patched) version as soon as it becomes available, to prevent exploitation of those vulnerabilities. Container scanning can also be categorised under SCA, as it scans third-party components for known issues as well. The only difference is that these components concern Docker images (or similar) instead of software libraries.

3.4.1. When to use

It is estimated that open-source code makes up 80 to 90 percent of the code composition of applications.[20] This comprises a large attack surface that must be secured. It is recommended to always apply SCA due to its low resource usage, ease of integration into CI/CD, and potential return on investment with respect to the prevention of the exploitation of known vulnerabilities.

3.4.2. Capabilities

SCA can pinpoint which dependency (and which version) is vulnerable, and even suggest upgrades to newer versions that do not contain the known vulnerability (remediation). CI/CD integration is easily achieved. SCA can also identify transitive (indirect) dependencies and monitor them for issues as well. SCA can help with licensing issues by identifying the open-source licenses that are used. Lastly, SCA can generate a Software Bill Of Materials (SBOM).

3.4.3. Limitations

SCA only scans third-party and open-source dependencies, and it only flags known vulnerabilities. It lacks contextual analysis, which can result in false positives. To verify the potential impact of a finding, manual review is still required.

3.4.4. State-of-the-art & possible trends

Advanced SCA tools scan transitive dependencies, combine multiple vulnerability databases, and additionally support container scanning (identifying vulnerabilities within containers and their components) [14]. Machine learning and AI are being looked into to improve the accuracy of SCA and the prioritisation of its findings [15, 16].

3.4.5. How commonly used in practice

SCA is the low-hanging fruit of automated software security testing techniques to be applied in any software project. It is very commonly used in practice already. There is however a difference in the level

[14] https://junit.org/junit5/
[15] https://jbehave.org/
[16] https://nunit.org/
[17] https://xunit.net/
[18] https://robotframework.org/
[19] https://docs.pytest.org/
[20] https://snyk.io/learn/application-security/sast-vs-sca-testing/#sca

of SCA tools applied; some do basic dependency scanning for known vulnerabilities, and others include the more advanced features of SCA as well (transitive dependencies, multiple vulnerability databases, and container scanning).

3.4.6. Most commonly used tools

SCA tools that are most used in practice include HCL[21], Snyk[22], GitLab[23], GitHub[24], Mend[25], Black Duck[26], SOOS[27], and Contrast[28].

3.5. Static Application Security Testing (SAST)

Static Application Security Testing (SAST) is an automated (whitebox) software security testing technique that scans all files in a software project (among which the source code) for known vulnerabilities, security issues, and insecure coding practices. It does so without executing the software (no running application and no test cases are required), and therefore this is a static testing method. Secret scanning / detection can also be categorised under SAST, as secret detection scans all text files (and thus source code) of a repository for possible leaked secrets, e.g., API tokens or passwords.

3.5.1. When to use

Including SAST in software security testing pipelines will improve code quality and application security. SAST tools do have false positives and false negatives in their findings, and so further analysis and additional software security testing techniques are still required. When using Continuous Integration, including a SAST testing stage is among the low-hanging fruit.

3.5.2. Capabilities

SAST detects defects and security vulnerabilities in software and pinpoints their exact location in the code. As a result, root-cause analysis is often not necessary and patching can be done faster. It does not require test cases like other testing techniques do. SAST can rank its findings in terms of severity and provide a description for what exactly the problem is. SAST does not require a running application (only source code), and it is easily integrated into CI/CD or even into a developer's IDE. Multiple analysis types are available, including configuration-, semantic-, dataflow-, control flow-, and structural analysis.[29]

3.5.3. Limitations

SAST tools can produce many false positives and false negatives, resulting in the requirement for manual verification of the findings. This results in a lower return on investment than with SCA. Additionally, the programming language of the software under test determines the availability of legacy or advanced SAST tools. SAST tools are only as good as the knowledge base behind them, and so incomplete knowledge bases give a false sense of security.

3.5.4. State-of-the-art & possible trends

More advanced SAST tools have greatly improved accuracy compared to legacy tools, however there are still false positives in play [17]. Real-time scanning in the IDE of a developer is offered, with in-line remediation suggestions [18]. Learning-based approaches for SAST are a topic of research [19], and AI

[21] https://www.hcltechsw.com/appscan
[22] https://snyk.io/product/open-source-security-management/
[23] https://docs.gitlab.com/ee/user/application_security/dependency_scanning/index.html
[24] https://github.com/features/security
[25] https://www.mend.io/sca/
[26] https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html
[27] https://soos.io/products/sca
[28] https://www.contrastsecurity.com/contrast-sca
[29] https://snyk.io/learn/application-security/sast-vs-sca-testing/#vs

is mentioned to play a role in the knowledge base behind certain SAST tools and in the reduction of unactionable findings of SAST tools [20]. Comparisons and synergies between SAST and AI tools are being researched as well [21]. The combination of SAST and DAST approaches in IAST is a trend that is gaining popularity [22].

3.5.5. How commonly used in practice

SAST is next to SCA one of the most frequently used software security testing tools. SAFECode recommends using SAST as part of the SSDLC [1].

3.5.6. Most commonly used tools

SAST tools that are most used in practice include SonarQube[30], HCL[31], Snyk[32], GitLab[33], GitHub[34], Mend[35], and Contrast[36].

3.6. Dynamic Application Security Testing (DAST)

Dynamic Application Security Testing (DAST) is an automated software security testing technique that requires a running application and a set of testcases. It regards the software under test as a black box, and therefore no source code access is required. With every run, the testcases are sent to the application and the corresponding responses are analysed for anomalies that point to faults, regressions, or vulnerabilities. DAST assesses the software from the perspective of an outside (possibly malicious) user.

3.6.1. When to use

DAST can be used to weed out false positives from other testing techniques through the analysis of the application at runtime. It can be used for regression testing, known vulnerability testing, or testing third-party components. DAST provides an extra layer of security testing and adding it to your pipeline next to SCA and SAST further strengthens your security posture.

3.6.2. Capabilities

DAST can simulate real-world attack scenarios, providing higher accuracy with regards to the findings of the technique. It can also be used for testing regressions within the project. It is programming language independent as it treats the application as a black box (does not require source code either). DAST also generates fewer false positives compared to SAST, due to its dynamic nature. DAST can be integrated into CI/CD as well. DAST is particularly well-suited for testing web applications and APIs.

3.6.3. Limitations

DAST does not provide automatic root-cause analysis (finding which line of code is responsible for a discovered fault), and so developers are still required to do this step manually. Additionally, DAST slows down the testing process, as it requires a running application and dynamic execution, which is time-consuming. DAST can also require some effort to set up properly, depending on the type of interface (and authentication complexity) to the software under test that is used. DAST techniques cannot identify all types of software weaknesses. DAST is only as effective as the set of testcases that are provided, as those determine the application coverage and types of attacks that are included in the testing technique (false negatives occur). DAST is time-consuming and resource-intensive.

[30] https://www.sonarsource.com/products/sonarqube/
[31] https://www.hcltechsw.com/appscan
[32] https://snyk.io/product/snyk-code/
[33] https://docs.gitlab.com/ee/user/application_security/sast/index.html
[34] https://github.com/features/security
[35] https://www.mend.io/sast/
[36] https://www.contrastsecurity.com/contrast-scan

3.6.4. State-of-the-art & possible trends

Combining DAST with techniques like SCA and SAST is gaining popularity, mostly in the form of Interactive Application Security Testing (IAST) [22]. Findings from static techniques can be verified by dynamic ones. As DAST does require effort to set-up, configure, and provide meaningful testcases for, these steps are being looked into to automate further [23]. Automated remediation of DAST findings is a topic of research as well [24]. Fuzzers (see Section 3.9) can aid in the generation of additional testcases (valid, invalid, and malformed) to further improve application coverage (the portion of the application's source code covered by the tests).

3.6.5. How commonly used in practice

DAST is less frequently used in practice than SCA and SAST, due to the effort required to set-up and configure it properly. However, DAST is still used more often than IAST, RASP or Fuzzing. SAFECode recommends using DAST as part of the SSDLC [1].

3.6.6. Most commonly used tools

DAST tools that are most used in practice include GitLab[37], SOOS[38], HCL[39], OWASP ZAP[40], and Burp Suite[41].

3.7. Interactive Application Security Testing (IAST)

Interactive Application Security Testing (IAST) is an application security testing technique that combines elements of both SAST and DAST technologies. IAST aims to provide more accurate and actionable security testing results by actively analysing an application's source code and behaviour at runtime during the testing phase. It inserts sensors in the software under test to capture actual application interactions and security events in real-time. Therefore, IAST should be able to produce fewer false positives and false negatives compared to SAST or DAST. As IAST is a technique which performs runtime testing in production, the tool must be GDPR compliant.
Note that TNO has not extensively validated to what extent state-of-the-art tools already deliver on the promises that IAST makes.

3.7.1. When to use

When you employ SAST and DAST techniques, and there are often a lot of false positives and false negatives, IAST could aid in this regard. IAST can be helpful in identifying security issues that may only become manifest under specific runtime conditions and user interactions. If you already employ SCA and SAST, adding IAST into you testing pipeline may be a good idea, as it covers dynamic code paths and interactions with external components. IAST is often used in web application testing, but it can also be used on third-party libraries.

3.7.2. Capabilities

IAST should be capable of producing fewer false positives and false negatives compared to SAST or DAST, due to the sensors embedded in the target application. It can also identify vulnerabilities in areas that are difficult for SAST to discover and assess, such as third-party libraries and dynamic code execution paths. IAST can be integrated into CI/CD pipelines as well, continuously testing the software and providing feedback to developers. IAST can be used to partially weed out false positives from other techniques like SAST.

[37] https://docs.gitlab.com/ee/user/application_security/dast/index.html
[38] https://soos.io/products/dast
[39] https://www.hcltechsw.com/appscan
[40] https://www.zaproxy.org/
[41] https://portswigger.net/burp

3.7.3. Limitations

IAST does incur runtime performance overhead, as sensors are embedded in the target application. The integration of IAST sensors into the software requires some setup and configuration steps that can become more complex when dealing with complex software. IAST is not available for all programming languages and technology stacks. IAST cannot cover all code paths in the software, especially those that require specific inputs or conditions to trigger. As IAST may capture sensitive data during testing, regulations like GDPR must be taken into consideration as well. Additionally, the quality of the testcases has a large impact on the effectiveness of IAST. Furthermore, IAST is no replacement for SAST and DAST. IAST may still produce some false results, and so proper tuning is essential.

3.7.4. State-of-the-art & possible trends

By combining the benefits of both static and dynamic analysis, IAST offers a more comprehensive and accurate assessment of application security [22]. While IAST can be effective in terms of low false positives and false negatives [25], its effectiveness depends on the testcases used [26]. Fuzzers (see Section 3.9) can step in to provide such testcases. Benchmarks show that IAST can outperform both SAST and DAST tools [25].

3.7.5. How commonly used in practice

IAST is gaining popularity as an effective security testing approach for modern software development. However, most companies still use tried and tested techniques like SCA, SAST, and DAST.

3.7.6. Most commonly used tools

IAST tools that are most used in practice include Optiv[42], Veracode[43], Synopsis[44], Contrast Assess[45], Burp Suite[46], and HCL[47].

3.8. Runtime Application Self Protection (RASP)

Runtime Application Self Protection (RASP) is an automated software protection technique that is usually employed with software running in production. RASP ensures that all requests made to the software are secure and are being validated properly. When used in the testing stage, alarms are generated when requests do not satisfy these requirements (which is quite like IAST). When used in production, RASP can either block a specific operation, or terminate the session the request was made from altogether. The application behaviour (in terms of traffic and activity) is closely monitored, such that in the case of impending unintended behaviour, the offending operation can be stopped in its tracks automatically. RASP can function as an extra layer of security for a running application; basic attacks against the application can be dealt with in real-time. To provide a broad overview of secure software development practices, RASP is included in this memo even though it is more a protection technique rather than a testing technique. However, RASP can still indirectly lead to the patching of vulnerabilities and regression testing. IAST and RASP apply similar technology, only in different lifecycle stages. Note that, just like for IAST, TNO has not validated the actual capabilities of state-of-the-art RASP tools.

3.8.1. When to use

[42] https://www.optiv.com/
[43] https://www.veracode.com/
[44] https://www.synopsys.com/software-integrity/security-testing/interactive-application-security-testing.html
[45] https://www.contrastsecurity.com/contrast-assess
[46] https://portswigger.net/burp/documentation/desktop/tools/infiltrator
[47] https://www.hcl-software.com/appscan

RASP is complementary to security testing techniques, as it monitors and protects applications that are already in production instead of in the testing stage. It adds an additional layer of protection, but it cannot protect against all kinds of attacks. Due to the effort required to properly setup RASP and the overhead that it incurs, it is often used for mission-critical software. RASP can also be useful for protecting applications that have known vulnerabilities, as it can mitigate the risks associated with these weaknesses while developers work on fixing them. RASP can be used as a last line of defence when already using e.g., Web Application Firewalls and Intrusion Detection Systems.

3.8.2. Capabilities

RASP offers real-time protection, detection, and response to security threats as they occur during application execution. It enforces security policies that define what behaviours and activities are considered malicious or suspicious. These policies are often customisable based on the specific needs of the application and organisation. RASP can dynamically adapt its responses based on the severity of the threat. It can block or mitigate attacks without disrupting legitimate application functionality (low false positives), and it can even partially protect against zero-days as well. It offers continuous security monitoring, and possible actions include blocking malicious requests, terminating suspicious sessions, or alerting administrators about potential threats. Based on the collected data of such attacks, the underlying problem can be patched in the codebase and regression tests can be added to the pipeline that verify similar attacks are no longer possible.

3.8.3. Limitations

RASP does not do advanced scanning and it does not provide comprehensive protection. It also incurs a performance overhead in production (as RASP operates within the application itself), but the CI/CD pipeline is not affected (in terms of an additional stage that slows down the pipeline). Additionally, as it intercepts and inspects the traffic going into the application, before applying RASP, regulations like GDPR must be taken into consideration as well. And even if RASP blocks a certain attack in its tracks, the application still has to be patched to ensure such an attack would no longer be possible in the future. In addition, RASP can have false negatives, where malicious or suspicious activity goes undetected. Configuring RASP properly can be complex, and even after configuring, further tuning remains necessary to optimally use this technique.

3.8.4. State-of-the-art & possible trends

There is a lack of effective protection provided by traditional perimeter-based application security solutions (intrusion prevention systems and web application firewalls) [27]. Instrumentation[48]-based solutions like RASP can be more effective than perimeter solutions because of continuous runtime observability from within the application. At the time of writing there does not seem to be a lot of ongoing academic research on RASP. However, there is a research area looking into self-healing cybersecurity, where the focus lies on the regeneration of parts of a system whenever a (malicious) anomaly has been detected [28].

3.8.5. How commonly used in practice

Currently RASP is not widely used, most companies still focus on tried and tested techniques like SCA, SAST and DAST. Most RASP tools focus on Java and .NET applications.

3.8.6. Most commonly used tools

[48] Runtime instrumentation is adding several features and runtime patches to observe software behaviour. https://mas.owasp.org/MASTG/techniques/generic/MASTG-TECH-0051/; https://link.springer.com/chapter/10.1007/978-3-642-16612-9_23

RASP tools that are most used in practice include Digital.ai[49], Contrast Protect[50], Guardsquare[51], Micro Focus Fortify[52], Imperva[53], Signal Sciences[54], and Waratek[55].

3.9. Fuzz Testing (Fuzzing)

Fuzzing is an automated software testing technique that generates many different inputs and provides them to the software under test, to trigger unexpected behaviour, failing assertions, memory leaks or even crashes of the software. These findings reveal faults in the software under test (particularly in input validation and error-handling mechanisms), including potential security vulnerabilities. A fuzzer generates (semi-random) inputs based on sample inputs, a specification of the expected input structure, coverage feedback, mutation strategies, and more, depending on the specific fuzzing tool that is used. By subjecting the application to various unexpected inputs, fuzzing helps uncover edge cases and weaknesses that might not be apparent in traditional testing methods. Fuzzing relates to DAST and IAST as it is also a dynamic testing technique that can instrument the target application. However, fuzzers can improve code coverage (and thus the probability of triggering bugs) through the feedback they obtain from the target application, by generating new testcases in a smart and automated way. Another way of increasing coverage is to complement a fuzzer with other techniques such as symbolic execution or taint analysis. TNO cybersecurity department, has conducted extensive research in this area and has built-up in-depth knowledge on fuzzing.

3.9.1. When to use

Fuzzing requires both more computational resources and time than the aforementioned techniques, this should be taken into consideration before choosing to employ fuzzing. When these resources are available, including fuzz testing is recommended, as fuzzers can find defects and vulnerabilities that other techniques may miss, thereby reducing the risk of potential exploits and data breaches. Fuzzing software that interacts with untrusted data, such as data from user input, files, network packets, or APIs, can be very effective, as fuzzers excel in those areas. Fuzzing is a valuable tool in manual security testing as well.

3.9.2. Capabilities

Fuzzers are a powerful automated tool in (smart) test case generation and defect discovery. Fuzzing can be integrated into the CI/CD pipeline to automatically test software under development. This allows for continuous validation, providing immediate feedback to developers about potential (security) flaws. Blackbox fuzzers can be employed when no source code is available for the software under test (e.g., third-party libraries). White-box and grey-box fuzzers are more effective, but do require source code, access to the application, or feedback from the target application to guide the fuzzing process. Fuzzers aim to maximise code coverage, and they may reach parts of the software that remain untested by other testing techniques. Fuzzing is a dynamic testing technique, resulting in no false positives if correctly configured compared to other testing techniques. Fuzzers also provide inputs that triggered unintended behaviour, enabling developers to reproduce the issue and pinpoint the root cause more easily using e.g., stack traces.

3.9.3. Limitations

[49] https://digital.ai/solutions/build-secure-software/
[50] https://www.contrastsecurity.com/contrast-protect
[51] https://www.guardsquare.com/runtime-application-self-protection-rasp
[52] https://www.microfocus.com/en-us/cyberres/application-security
[53] https://www.imperva.com/products/runtime-application-self-protection-rasp/
[54] https://www.signalsciences.com/products/rasp-runtime-application-self-protection/
[55] https://waratek.com/resources/introduction-to-runtime-application-security-protection-rasp/

Fuzzers are quite resource-heavy in terms of computational power and run time. Also, a one-time manual action has to be taken before a fuzzer can be employed effectively. This action consists of writing a so-called fuzzing harness, which acts as the entry point to the software under test, enabling a fuzzer to be able to interact with the target software. The better the harness, the more effective the fuzzer will be. However, fuzzing is not a replacement for other testing techniques; it complements them. Fuzzers are not able to achieve full code coverage due to increasing complexity of applications. Without a grammar or seed inputs for the fuzzer to generate new inputs from, the fuzzer may be less effective. Fuzzers are unable to detect every possible form of unintended behaviour, and they are dependent on additional tools called sanitisers to detect more elaborate classes of unintended behaviour properly.

3.9.4. State-of-the-art & possible trends

With the current state of the art in fuzzing techniques, fuzzers aim to achieve the highest amount of coverage within the software under test, as to exercise as many of the code paths in the program as possible [29]. This is what is known as coverage-guided fuzzing. Additionally, there is a newer technique called directed fuzzing, which is distance-guided, and has the potential to become the new state of the art in the future [30]. Directed fuzzers aim to spend most of their effort on exercising specific parts of the software under test, which are deemed to be more likely to contain faults. Such parts can be determined by other software security testing techniques, or through version control systems that steer the fuzzer towards the code changes of new commits. Directed fuzzing in its current state is especially effective for specific applications, including patch testing, crash reproduction, static analysis verification, and 1-day proof of concept generation [30]. Additionally, coverage-guided fuzzing still remains the state of the art. Fuzz testing is being integrated into CI/CD pipelines more and more, providing early feedback to developers about potential (security) flaws [31]. This aids the shift-left security paradigm, where the cost of patching and the risk of exploitation decreases. Lastly, there is ongoing research in the area of combining fuzzing with generative AI to improve code coverage and defect detection [32]. Also, Code Intelligence and the team behind OSS-Fuzz are researching automated harness generation (through LLM models), which can increase code coverage by thirty percent, and has shown its capability of finding crashes that the current developer-written harnesses missed [33-35]. Academia is researching the same direction [36].

3.9.5. How commonly used in practice

Fuzzing is widely adopted and common practice in the field of software security and quality assurance. It is used in a variety of industries and domains, including software development, security research, penetration testing, and product security assessments. With its integration into CI/CD, it became even easier to add to your security testing pipeline. Google is providing free CI/CD fuzz testing[56] on its servers for common open-source software projects, with the goal of making them more stable and secure. Additionally, fuzzers themselves are often open sourced as well, meaning that they can easily be employed. Fuzzing has become a recommended best practice in various industry standards and security guidelines. It is a crucial tool used by security researchers, penetration testers, and red teams to discover vulnerabilities in applications and network protocols. SAFECode recommends using fuzzing as part of the SSDLC [1].

3.9.6. Most commonly used tools

Fuzzers that are most used in practice include AFL[57], AFL++[58], libFuzzer[59], and honggfuzz[60].

[56] https://google.github.io/oss-fuzz/
[57] https://lcamtuf.coredump.cx/afl/
[58] https://aflplus.plus/
[59] https://llvm.org/docs/LibFuzzer.html
[60] https://honggfuzz.dev/

4. Software Security Testing Tools

Within the application security testing domain, there are many companies in the market to choose from. A recent publication by Gartner (see **Figure 4.1**) shows the top-12 players within this field, ranked based on their ability to execute and their completeness of vision. We observe that Synopsys[61] is the current leader of the field, according to Gartner, offering solutions for all software security testing techniques discussed in the previous section. However, our analysis will initially be based on open source / freely available tooling, since availability of the tool, its implementation, and the features it (partially) supports are public knowledge instead of hidden behind marketing schemes of companies offering their solutions.

Figure 4.1: 2023 Gartner® Magic Quadrant™ for Application Security Testing

For the four most popular automated software security testing techniques described in the previous section, we are going to consider one of the most used open-source tools in further detail. Per tool, we discuss our findings, our vision on its potential usability, and possible improvements that could be made. We do not consider manual software security testing tools, as we want to focus on automated tooling. We also leave out RASP tools, since RASP is a technique used more in production than testing. Fuzz testing tools are out of scope for this memo as well because we have already investigated many fuzzers before.

The process of selecting which tool to consider per category (SCA, SAST, DAST, and IAST) consists of several aspects. As mentioned earlier, we only consider open source / freely available tools. Secondly, we only consider tools that are popular and commonly used in the field. Thirdly, we only consider tools that have a relatively high maturity level. Lastly, tools that were included in benchmarks / papers /

[61] https://www.synopsys.com/software-integrity/solutions/application-security-testing.html

studies more often are given a higher preference, as those results will strengthen our analysis. In the end, we selected OWASP Dependency-Track, SonarQube CE, Zed Attack Proxy, and Contrast Assess CE.

4.1. SCA – OWASP Dependency-Track

OWASP Dependency-Track[62] is an open-source SCA tool that is described as an intelligent Component Analysis platform that allows organisations to identify and reduce risk in the software supply chain. Features include detection of known vulnerabilities in third-party components, security and license policy evaluation, prioritising mitigations based on the Exploit Prediction Scoring System, outdated version detection of third-party components, usage of Software Bill of Materials (SBOM), integration with multiple vulnerability databases (NVD, OSS Index, GitHub, Snyk, OSV, and VulnDB), producing Vulnerability Disclosure Reports (VDR) and using Vulnerability Exploitability eXchange (VEX), and providing an API for ease of integration with other systems. Dependency-Track is integrated into CI/CD instead of into IDEs of developers, and it analyses complete projects across all versions and branches based on the SBOMs it ingests from those pipelines.

The main aim of Dependency-Track is to answer the question of what is affected, and where. Whenever a new vulnerability report comes out, software owners and maintainers want to know which of their applications, (operating) systems, and platforms are affected by the new vulnerability. Dependency-Track provides this insight, such that (prioritised) remediation can start immediately, without the need to pinpoint the locations of the vulnerability across all the tracked projects. Dependency-Track can be summarised as a continuous SBOM analysis platform, where all projects and their versions are tracked in a single dashboard.

Dependency-Track also provides dependency graphs for each tracked project. Additionally, it contains a vulnerability auditing framework that can be used to investigate discovered vulnerabilities per project in further detail. For instance, if the vulnerability exists in a portion of the code that is unreachable (the vulnerable method is not used), or certain preconditions are not met for the vulnerability to manifest, it can be flagged as "Not affected" and the alert can be suppressed. The auditing framework allows for comments to be placed as well, which is beneficial for collaborative efforts and traceability. Furthermore, Dependency-Track allows to set custom (security) policies, such that all projects are evaluated for adherence to those policies. For instance, if a certain (version of a) library is disallowed company-wide, a policy can be set to automatically verify it is not used in any of the tracked projects. The same can be done for licensing policies, and Dependency-Track generates alerts when there are licensing violations.

4.1.1. Findings

Dependency-Track is an advanced open-source SCA tool in the form of a continuous SBOM analysis platform that – among many other features – also provides security and licensing policy evaluations. It can be easily deployed anywhere using a simple Docker command. It is supposed to run continuously, downloading new vulnerability reports as they are announced in the linked vulnerability databases, and ingesting SBOMs from CI/CD pipelines of the projects that are being tracked. This means that the SBOM creation (at build-time in the CI-pipeline) and sending it to Dependency-Track has to be done using e.g., the Jenkins Plugin[63] or the GitHub Action[64], but it can also be done using a `curl` command.

OWASP Dependency-Track is much more than a regular SCA tool, and so if only a command-line interface (or Maven/Ant/Jenkins plugin) for a simple dependency scan is required, OWASP Dependency-

[62] https://owasp.org/www-project-dependency-track/ and https://dependencytrack.org/
[63] https://plugins.jenkins.io/dependency-track/
[64] https://github.com/marketplace/actions/upload-bom-to-dependency-track

Check[65] would suffice. Dependency-Check can be automated as part of a CI/CD pipeline as well, but it lacks all the additional features that the Dependency-Track platform provides. It is just a tool to detect publicly disclosed vulnerabilities contained within a project's dependencies.

4.1.2. Potential usability

The tool is very easily deployed anywhere by leveraging Docker containers. The usage of the tool is well-documented and the API as well as the UI are both easy to work with and very understandable. The tool provides an all-in-one overview of all tracked projects (including software, libraries, frameworks, applications, (operating) systems, et cetera). Whenever new vulnerability intel is available, affected projects are immediately brought to the attention of the user. Policies concerning security and licensing can also be easily enforced.

Dependency-Track can be used to map all dependencies of your own software projects, but it can also be used during software procurement. SBOMs from vendors can be ingested by the tool, analysed thoroughly, and compared to one another. Based on the outcome, a more informed decision can be made which vendor to go with.

Dependency-Track has a lot of integrations built-in for different vulnerability intelligence sources, for different notification options, for different CI platforms, for most of the popular package managers, and for integration with vulnerability tracking dashboards like DefectDojo[66].

4.1.3. Possible improvements

The most interesting possible improvement that can be made to Dependency-Track is probably (semi-) automated remediation. Dependency-Track would suggest dependency upgrades that no longer have a certain vulnerability, and maybe Dependency-Track could even be configured to automatically trigger the more straightforward upgrades without user interaction. Other than this, Dependency-Track is already a very mature tool and there are no important features missing. It can always be extended to support more types of risk that can be identified, extended to integrate with more sources of vulnerability intelligence, and extended to support more package managing tools in addition to Cargo, Composer, Gems, Hex, Maven, NPM, NuGet, and PyPI.

4.2. SAST – SonarQube CE

SonarQube[67] is an automatic code review tool which can help in detection of bugs and vulnerabilities in software during the testing process. Some features of this tool include several languages support for code quality checks. This tool can be integrated to the DevOps and CI pipelines and cycles. SonarLint[68] is a free and open-source IDE plugin which helps finding and fixing coding issues. This tool provides support for more than 25 languages and can highlight common errors, bugs, and vulnerabilities. SonarQube provides feedback via UI, email and in decorations on pull or merge requests as notifications.

Analysis of code is performed by requesting a snapshot of the code from the server and then analysis is performed. Results are then sent back to the server in a report. SonarQube provides support for several plugins and so it can be extended based on requirements.

4.2.1. Findings

[65] https://owasp.org/www-project-dependency-check/
[66] https://www.defectdojo.org/
[67] https://www.sonarsource.com/open-source-editions/sonarqube-community-edition/
[68] https://www.sonarsource.com/products/sonarlint/

Each SonarQube instance has three main components, namely *SonarQube server* with a compute engine, search server and a webserver with its user interface; *a database* to store metrics, issues, and configurations and finally *one or more scanners* as configured into continuous integration for analysis[69]. This can be hosted on servers on premise or as a self-managed cloud environment. Hence providing possibilities to integrate into development environments such as GitHub or Azure DevOps.[70] The reporting features include insights into key metrics, security reports with coverage for OWASP ASVS, OWASP Top 10, PCI DSS and CWE Top 25. These are only available in the commercial edition and not the free community version. The community edition provides support to analyse only the main branch for each project and does not support any feature branch analysis.

4.2.2. Potential usability

This tool is easily deployed as a Docker image, can be installed as a desktop application on the system of a tester, or can be integrated into CI/CD pipelines. The usage of the tool is well-documented and the UI is both easy to work with and very understandable. New projects and scans are easy to run, but advanced features do require a bit more effort to set-up and configure properly to get the highest test efficiency. There are certain prerequisites such as that Java must be installed on the machine where SonarQube will run, and certain hardware requirements as provided in their documentation.[71]

4.2.3. Possible improvements

Support for feature branch analysis in community edition can be added. This is a free tool and it is available for more testing before implementing the commercial tool across all platforms. Since the number of false positives are quite high for SAST tools, a combination of IAST and fuzzing could improve code quality and reduce false positives.

4.3. DAST – Zed Attack Proxy (ZAP)

Zed Attack Proxy[72] is an open-source DAST tool that is aimed at testing web applications. There are a lot of add-ons that can be used with ZAP, providing many kinds of web app scanning, testing and attack techniques. ZAP works by proxying the traffic between a browser and the web application being tested. All the data and messages can be intercepted, inspected, modified, and forwarded. ZAP can be deployed as a background process (for automation purposes) or in the form of a standalone application. The tool can be installed as a Java desktop application, deployed as a Docker container, or integrated into CI/CD pipelines. ZAP has a large and active community, and the tool is well-documented. ZAP will join The Software Security Project[73] (a new initiative from The Linux Foundation) in 2023, and will therefore leave OWASP, which it has been a part of since 2012.[74]

ZAP helps to automatically find security vulnerabilities in web applications during development and testing stages. It is also a great tool for experienced pentesters to use for manual security testing. ZAP works by first executing a passive scan, crawling the web application in search for as many endpoints as it can find and seeing whether any potential problems are apparent. Afterwards, an active scan can be started that tries many known attacks against the application. For both stages, the findings will be reported in terms of which endpoint, the risk, the confidence, the parameters used, the attack that was used, corresponding CWE and WASC IDs, a description of the issue, additional info, a general solution, and a reference to the source of the issue information. This enables developers to understand the issue

[69] https://docs.sonarsource.com/sonarqube/latest/setup-and-upgrade/install-the-server/
[70] https://www.sonarsource.com/blog/sq-sc_guidance/
[71] https://docs.sonarsource.com/sonarqube/latest/requirements/prerequisites-and-overview/
[72] https://www.zaproxy.org/
[73] https://softwaresecurityproject.org/
[74] https://www.zaproxy.org/blog/2023-08-01-zap-is-joining-the-software-security-project/

more easily and to be able to fix the problem more quickly. In addition to automated scans, tests, and attacks, ZAP also provides manual exploration, inspection, and attack functionality.

Advanced features of ZAP include the ZAP Marketplace (where (community) plugins can be downloaded from), automation options (in the form of Docker, GitHub Actions, the Automation Framework, and API and Daemon mode), a built-in SCA tool for JavaScript libraries (Retire.js), support for WebSockets, different authentication options, OpenAPI specifications, and the integration of fuzzers to fuzz HTTP or WebSocket requests.

4.3.1. Findings

ZAP is a widely used web app scanner with a lot of powerful features that can be used in both manual and automated dynamic testing of web applications. It can automatically explore apps to discover existing endpoints and issues that are apparent. Based on the discovered endpoints, known attacks can automatically be run against the app to test its security. ZAP produces a nice report with all the findings, categorised by severity. ZAP is extensible through the plugins and custom scripts that it supports. Therefore, it can be used in many testing scenarios that require tailor-made solutions.

OWASP Benchmark results from 2016 show a true positive rate of 20% combined with a 0% false positive rate for ZAP version 2016-09-05 (DAST). For SonarQube version 3.14 (SAST), the benchmark results show a true positive rate of 51% combined with a 17% false positive rate [37]. Another evaluation of ZAP shows that the tool can detect all of the OWASP Top 10 security risks for web applications [38]. There do not seem to be recent public benchmark results that compare ZAP to its counterparts from both the open-source community and commercial companies.

4.3.2. Potential usability

The tool is very easily deployed anywhere by leveraging Docker containers, installed as a desktop application on the system of a tester, or integrated into CI/CD pipelines. The usage of the tool is well-documented and the API as well as the UI are both easy to work with and very understandable. The automated scans and attacks are quite straightforward to run, but the more advanced features do require a bit more effort to set-up and configure properly to get the highest test efficiency. The readily available ZAP GitHub Actions and Docker containers really ease the automation of DAST, making it an accessible way to dynamically test web applications at all stages of their SDLC.

4.3.3. Possible improvements

Passive and active scanner rules, spiders, community scripts and known attacks can always be extended and improved further. Support for different authentication mechanisms and protocols can also be added. ZAP could also provide some kind of agent that can be incorporated in test versions of the web application, such that it could provide feedback from within the application during scanning. This would make ZAP a more hybrid tool that could fall in the IAST category of testing techniques. ZAP could also incorporate multiple different network / protocol / API fuzzers, which would be a big step towards an all-in-one tool for web application testing (it would support manual security testing, SCA, DAST, IAST and advanced Fuzzing).

4.4. IAST – Contrast Assess CE

Contrast Assess[75] is an interactive application security testing (IAST) tool. This tool combines static and dynamic application security testing. This tool deploys agents and sensors which monitor the applications from within. Contrast Assess then identifies vulnerabilities and bugs in the code which can

[75] https://www.contrastsecurity.com/contrast-community-edition

be fixed by developers. Main advantages of this tool over SAST and DAST tools are reduction in false positives, code coverage and vulnerability coverage as mentioned by Contrast.[76]

Key features of this tool include extensive code coverage while having an overview on the vulnerabilities as included in the OWASP Top Ten. Contrast helps in analysis of third-party code which aids in improving code quality. Another important feature of Contrast is to have a unified bill of materials which can be used for tracking changes on security details. Contrast can generate architectural diagrams to view different components as well.

4.4.1. Findings

With the rise in microservices and applications, keeping security in mind is crucial to ensure robust and reliable software. Although IAST has many similarities to DAST with its focus on application behaviour in runtime, it still combines black-box testing, scanning, and analysing internal flows with the agents. This brings together benefits from DAST and source code like analysis of SAST. Hence enhancing the possibilities of early detection of vulnerabilities and bugs.

One of the shortcomings of IAST is its dependency on the programming language. It is also time intensive and considerable effort would be required to set up the environment required for testing. IAST is also dependent on the existing code coverage. Hence one main drawback is that if your code coverage testing is not high, this would impact the effectiveness of IAST. Contrast Assess Community Edition has limited capabilities when compared to the commercial version. At the time of writing, there are no open-source IAST tools available.

4.4.2. Potential usability

To set up IAST tooling in a CI/CD pipeline, agents are deployed in the applications under consideration. It can be automated with Docker based deployment and integrated into the pipeline. Contrast Assess IAST tooling has good documentation and a user community for support. The agents being deployed support several languages, which are Java, .NET framework, .NET core, Node.js, PHP, Python, Ruby and Go. It is also possible to integrate the Contrast Platform itself with other management systems.

4.4.3. Possible improvements

Support for new languages, authentication mechanisms and new protocols can still be added. Contrast Assess could also incorporate different network / protocol / API fuzzers, which would be a big step towards an all-in-one tool for application testing. This would be adding capabilities to the Contrast Secure Code Platform to include more aspects of testing techniques. It would then support SCA, RASP, IAST and advanced fuzzing techniques.

[76] https://www.contrastsecurity.com/glossary/interactive-application-security-testing

5. Discussion

Table 1 provides a mapping of the different testing techniques and their primary features. The considered techniques are code coverage, actionability, and reliability. *Code coverage* is an important feature to consider not only for the security testing aspect but also for code quality. Increased code coverage during testing is a relevant indicator to consider while choosing a particular testing technique, as it fosters confidence that the software under test behaves as intended. When performing security testing, knowing the precise location of an issue or bug is highly relevant as this helps in reducing the amount of effort that is required to resolve the bug or issue. Such a feature, *actionability*, helps in understanding why a certain technique would have an advantage over others. When looking at fuzzers, the findings include inputs that triggered unintended behaviour, enabling developers to reproduce the issue and pinpoint the root cause more easily. When looking at static analysis techniques, the findings directly include specific locations in the source code, resulting in high actionability. Furthermore, when considering *reliability*, certain testing techniques such as static analysis perform worse, as there are high false positive rates involved. This is a particular drawback, as developers and testers will have to manually work through the false positives to reach the true positives that have to be fixed. This results in a lack of adoption in the development and testing cycle, which subsequently results in a higher likelihood that production software contains vulnerabilities and bugs.

Table 1: Indicative mapping of software security testing techniques and their primary features, ranked Low/Medium/High.

	SCA	SAST	DAST	IAST	Fuzzing
Code coverage	High	High	Low	Medium	Medium
Actionability	High	High	Low	Medium	Medium
Reliability	Medium	Low	High	Medium	High

In Table 2, a mapping of software testing techniques and their secondary features is provided. These features are focused on the setup and use of the testing techniques themselves. While the primary features are more important, the secondary features influence the process of selecting testing techniques as well. While the *testing time consumption* is high for fuzzing, the findings from longer-running fuzzing campaigns provide extensive and valuable insights. Although Table 2 suggests both SCA and SAST perform very well, combining SCA and SAST is insufficient as a complete solution for increasing software security, as both techniques come with drawbacks indicated by Table 1. Furthermore, it is important to note that the type of issues that can be discovered vary per technique. Hence, one must interpret both tables with caution. The *setup time and required expertise* are considered high for fuzzing. However, with the advent of generative AI, both of these aspects are expected to be reduced by automating certain tasks in the chain (see 3.9.4).

Table 2: Indicative mapping of software security testing techniques and their secondary features, ranked Low/Medium/High.

	SCA	SAST	DAST	IAST	Fuzzing
Testing time consumption	Low	Low	Medium	Medium	High
Setup time	Low	Low	Medium	High	High
Required expertise for setup	Low	Low	Medium	High	High
Knowledge required in interpreting results	Low	Medium	High	High	High

Both tables show that certain testing techniques such as SCA and SAST have a low barrier to be applied; they require low effort to be integrated into the SDLC, while achieving relatively good results considering the effort invested. Afterwards, other (complementary) techniques can be added to the SDLC, which further increases the ability to find security and quality issues that have a higher complexity or severity. Combining different testing techniques results in a better software security posture.

There are trade-offs with each form of testing, which is why most mature security programs employ multiple approaches [1]. This would be our recommendation as well; increasing diversity in testing is the best way of finding and resolving more vulnerabilities. This would include not only automated tooling, but the manual aspect of software quality assurance as well. While some automated security tools can provide highly accurate and actionable results, automated security tools tend to have a higher rate of false positives and negatives than a skilled human security tester manually looking for vulnerabilities [1]. Nevertheless, automated tools are adopted because of the speed and scale at which they run, and because they allow organisations to focus their manual testing on their riskiest components, where they are needed most [1]. Given the current labour market of cyber security specialists (where there is a shortage of), automation in software testing is the most promising solution for secure software development.

There is lots of ongoing research on all the software security testing techniques in an effort to improve security and privacy aspects of the digital society. In recent studies, several SAST, DAST, and IAST tools have been compared to one another and different combinations of them have been benchmarked to investigate possible synergies [25]. IAST tooling achieved the highest scores in the benchmark, even though IAST is not as mature as SAST and DAST. This is quite promising, and IAST can potentially become the new standard in secure software development practices in the future.

Recent work also shows efforts to open-source the integration of software security testing techniques into CI/CD pipelines. SonarQube (SAST) and OWASP ZAP (DAST) have been successfully integrated into CI/CD already, providing a report of their combined findings for every commit [39, 40]. The modularity of the designed system enables the easy integration of additional software security testing tools and the authors aim to include SCA and IAST tools in their system in the future [39]. Other works show CI/CD integration of many security tests including SonarQube (SAST), Clair (container scanning), GitLeaks (secret scanning), OWASP Dependency-Check (SCA), and OWASP ZAP (DAST) can be done in a single pipeline, and the results can be easily aggregated and displayed in DefectDojo (vulnerability tracking dashboard) [41]. All these efforts lower the threshold for developers to incorporate software security testing into their own projects, resulting in secure software development being applied more broadly.

Contrast Security aims to integrate several testing techniques in a single platform providing a complete view on the software testing process with different techniques. However, this is a commercial option and no open-source platform with similar functionality exists (i.e. that integrates SCA, SAST, IAST, RASP and testing of serverless applications)[77]. Other platforms (from Synopsys and CheckMarx) integrate some aspects of the application security chain but not all testing techniques are included in a single platform. Support for multiple testing techniques in a single platform will improve the ease of adoption of new techniques, ultimately improving the quality and security of software.

Creating a connection between current practice and newer technologies is important as newer technologies can greatly enhance current software security testing techniques and principles, and therefore improve the overall digital safety and security of our society. Academic progress should be applied to current security practices faster, to be able to protect against security threats more effectively. To apply the academic state-of-the-art progress into current security practices, more time and effort should be invested increasing the usability and applicability of such academic proofs of principle enabling the adoption by a wider audience.

Realising a safe and secure future requires awareness and continuous improvement of security testing practices. There is a shared responsibility for realising a secure (digital) future. By extending and sharing knowledge on the topic both academically and in industry this can be realised.

Want to learn more? Please reach out if you are interested in expanding your understanding of software security testing techniques, and how they can strengthen the security posture of your organisation.

[77] https://www.contrastsecurity.com/platform

6. References

[1] S. Simpson, "SAFECode Whitepaper: Fundamental Practices for Secure Software Development 2nd Edition," in *ISSE 2014 Securing Electronic Business Processes*, Springer, 2014, pp. 1–32. doi: 10.1007/978-3-658-06708-3_1.

[2] OWASP, "Application Security Verification Standard 4.0.3," 2021.

[3] M. Staron, M. Ochodek, W. Meding, and O. Soder, "Using Machine Learning to Identify Code Fragments for Manual Review," in *Proceedings - 46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020*, IEEE, 2020, pp. 513–516. doi: 10.1109/SEAA51224.2020.00085.

[4] S. T. Shi, M. Li, D. Lo, F. Thung, and X. Huo, "Automatic code review by learning the revision of source code," in *33rd AAAI Conference on Artificial Intelligence, AAAI 2019, 31st Innovative Applications of Artificial Intelligence Conference, IAAI 2019 and the 9th AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019*, 2019, pp. 4910–4917. doi: 10.1609/aaai.v33i01.33014910.

[5] C. Greco, G. Fortino, B. Crispo, and K. K. R. Choo, "AI-enabled IoT penetration testing: state-of-the-art and research challenges," *Enterp. Inf. Syst.*, p. 2130014, 2022, doi: 10.1080/17517575.2022.2130014.

[6] D. R. McKinnel, T. Dargahi, A. Dehghantanha, and K. K. R. Choo, "A systematic literature review and meta-analysis on artificial intelligence in penetration testing and vulnerability assessment," *Comput. Electr. Eng.*, vol. 75, pp. 175–188, 2019, doi: 10.1016/j.compeleceng.2019.02.022.

[7] A. Chowdhary, D. Huang, J. S. Mahendran, D. Romo, Y. Deng, and A. Sabur, "Autonomous security analysis and penetration testing," in *Proceedings - 2020 16th International Conference on Mobility, Sensing and Networking, MSN 2020*, IEEE, 2020, pp. 508–515. doi: 10.1109/MSN50589.2020.00086.

[8] J. Schwartz and H. Kurniawati, "Autonomous Penetration Testing using Reinforcement Learning," *arXiv Prepr. arXiv1905.05965*, 2019, [Online]. Available: http://arxiv.org/abs/1905.05965

[9] D. Gonzalez, P. P. Perez, and M. Mirakhorli, "Barriers to shift-left security: The unique pain points of writing automated tests involving security controls," in *International Symposium on Empirical Software Engineering and Measurement*, 2021, pp. 1–12. doi: 10.1145/3475716.3475786.

[10] M. Khari and P. Kumar, "An extensive evaluation of search-based software testing: a review," *Soft Comput.*, vol. 23, no. 6, pp. 1933–1946, 2019, doi: 10.1007/s00500-017-2906-y.

[11] J. Campos, A. Arcuri, G. Fraser, and R. Abreu, "Continuous test generation: Enhancing continuous integration with automated test generation," in *ASE 2014 - Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014, pp. 55–66. doi: 10.1145/2642937.2643002.

[12] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2011, pp. 416–419. doi: 10.1145/2025113.2025179.

[13] N. Tillmann and J. De Halleux, "Pex-white box test generation for .NET," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer, 2008, pp. 134–153. doi: 10.1007/978-3-540-79124-9_10.

[14] Snyk, "Guide to Software Composition Analysis (SCA)," 2022. https://snyk.io/series/open-source-security/software-composition-analysis-sca/ (accessed Aug. 08, 2023).

[15] N. Phadnis, "How software composition analysis can help you go from good to great," *Sonatype*, 2023. https://blog.sonatype.com/how-software-composition-analysis-sca-can-help-you-go-from-good-to-great (accessed Aug. 02, 2023).

[16] Y. Chen, A. E. Santosa, A. M. Yi, A. Sharma, A. Sharma, and D. Lo, "A Machine Learning Approach for Vulnerability Curation," in *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020*, 2020, pp. 32–42. doi: 10.1145/3379597.3387461.

[17] J. Yang, L. Tan, J. Peyton, and K. A Duer, "Towards Better Utilizing Static Application Security Testing," in *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2019*, IEEE, 2019, pp. 51–60. doi:

10.1109/ICSE-SEIP.2019.00014.

[18] Synopsys, “Static Application Security Testing.” https://www.synopsys.com/glossary/what-is-sast.html (accessed Aug. 08, 2023).

[19] R. Croft, D. Newlands, Z. Chen, and A. M. Babar, “An empirical study of rule-based and learning-based approaches for static application security testing,” in *International Symposium on Empirical Software Engineering and Measurement*, 2021, pp. 1–12. doi: 10.1145/3475716.3475781.

[20] X. Yang, Z. Yu, J. Wang, and T. Menzies, “Understanding static code warnings: An incremental AI approach,” *Expert Syst. Appl.*, vol. 167, p. 114134, 2021, doi: 10.1016/j.eswa.2020.114134.

[21] O. S. Ozturk, E. Ekmekcioglu, O. Cetin, B. Arief, and J. Hernandez-Castro, “New Tricks to Old Codes: Can AI Chatbots Replace Static Code Analysis Tools?,” in *ACM International Conference Proceeding Series*, 2023, pp. 13–18. doi: 10.1145/3590777.3590780.

[22] Y. Pan, “Interactive application security testing,” in *Proceedings - 2019 International Conference on Smart Grid and Electrical Automation, ICSGEA 2019*, IEEE, 2019, pp. 558–561. doi: 10.1109/ICSGEA.2019.00131.

[23] T. Rangnau, R. V. Buijtenen, F. Fransen, and F. Turkmen, “Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines,” in *Proceedings - 2020 IEEE 24th International Enterprise Distributed Object Computing Conference, EDOC 2020*, IEEE, 2020, pp. 145–154. doi: 10.1109/EDOC49727.2020.00026.

[24] V. Bril, “Automation of Remediation of Configuration Vulnerabilities Reported by the DAST Scanning Procedure.” Dublin, National College of Ireland, 2023.

[25] 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, pp. 1–26, 2020, doi: 10.3390/app10249119.

[26] Invicti, “Interactive application security testing (IAST).” https://www.invicti.com/learn/interactive-application-security-testing-iast/ (accessed Aug. 08, 2023).

[27] Contrast Security, “Runtime Application Self Protection (RASP) Security.” https://www.contrastsecurity.com/glossary/rasp-security (accessed Aug. 08, 2023).

[28] B. Gijsen, R. Montalto, J. Panneman, F. Falconieri, P. Wiper, and P. Zuraniewski, “Self-Healing for Cyber-Security,” in *2021 Sixth International Conference on Fog and Mobile Edge Computing (FMEC)*, IEEE, 2021, pp. 1–7.

[29] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, “AFL++: Combining incremental steps of fuzzing research,” in *WOOT 2020 - 14th USENIX Workshop on Offensive Technologies, co-located with USENIX Security 2020*, 2020.

[30] M. Böhme, V. T. Pham, M. D. Nguyen, and A. Roychoudhury, “Directed greybox fuzzing,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2017, pp. 2329–2344. doi: 10.1145/3133956.3134020.

[31] T. Klooster, F. Turkmen, G. Broenink, R. Ten Hove, and M. Böhme, “Continuous Fuzzing: A Study of the Effectiveness and Scalability of Fuzzing in CI/CD Pipelines,” in *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, IEEE, 2023, pp. 25–32.

[32] J. Hu, Q. Zhang, and H. Yin, “Augmenting Greybox Fuzzing with Generative AI,” *arXiv Prepr. arXiv2306.06782*, 2023.

[33] OSS-Fuzz, “Fuzz target generation using LLMs,” 2023. https://google.github.io/oss-fuzz/research/llms/target_generation/ (accessed Sep. 15, 2023).

[34] D. Liu, J. Metzman, and O. Chang, “AI-Powered Fuzzing: Breaking the Bug Hunting Barrier,” 2023. https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html

[35] Code Intelligence, “Breaking the Barrier of Dynamic Testing: Detect and Autoconfigure Entry Points With CI Spark,” 2023. https://www.code-intelligence.com/blog/ci-spark

[36] C. Zhang *et al.*, “Understanding Large Language Model Based Fuzz Driver Generation,” *arXiv Prepr. arXiv2307.12469*, 2023.

[37] M. C. Páez, “Application Security Testing Tools Study and Proposal,” 2020.

[38] D. H. A. Gonçalves, "DevSecOps for web applications: a case study." 2022.
[39] C. Aparo, C. Bernardeschi, G. Lettieri, F. Lucattini, and S. Montanarella, "An Analysis System to Test Security of Software on Continuous Integration-Continuous Delivery Pipeline," in *2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, IEEE, 2023, pp. 58–67.
[40] H. Setiawan, L. E. Erlangga, and I. Baskoro, "Vulnerability analysis using the interactive application security testing (iast) approach for government x website applications," in *2020 3rd International Conference on Information and Communications Technology (ICOIACT)*, IEEE, 2020, pp. 471–475.
[41] F. Freitas, "Application security in continuous delivery." Universidade do Porto (Portugal), 2021.