

# Getting Started With Static Code Analysis

## CONTENTS

- What Is Static Code Analysis?
- Types of Application Security Testing
- Comparison to DAST
- Benefits of Static Code Analysis
- Challenges of Static Code Analysis
- Getting Started With Static Code Analysis
- Advanced Static Code Analysis
- Conclusion

**JOHN VESTER**

SOFTWARE ARCHITECT, CLEANSLATE TECHNOLOGY GROUP

## WHAT IS STATIC CODE ANALYSIS?

Static code analysis is the practice of examining application's source, bytecode, or binary code without ever executing the program code itself. Instead, the code under review is analyzed to identify any defects, flaws, or vulnerabilities which may compromise the integrity or security of the application itself.

The roots for static code analysis actually pre-date the existence of computers and transistors themselves. In 1936, mathematician and computer scientist [Alan Turing](#) studied "the halting problem", which determines whether an arbitrary computer program will finish running or continue to run forever, based upon an input source of data. Turing concluded that — while it is possible to provide specific input to cause a given program to halt, it is not possible to create a generic algorithm that applies to all such computer programs. It was at this point the concept of static code analysis was born.

However, it was not until forty-two years later (in 1978) when static code analysis options started to emerge as commercially available products. The first product to reach the market was called "lint", a product of [Stephen C. Johnson](#) (AT&T Bell Laboratories). The term is a metaphorical reference to small programming errors that can result in big consequences — similar to small pieces of fabric which are caught in the lint trap of a drying machine used to launder clothes.

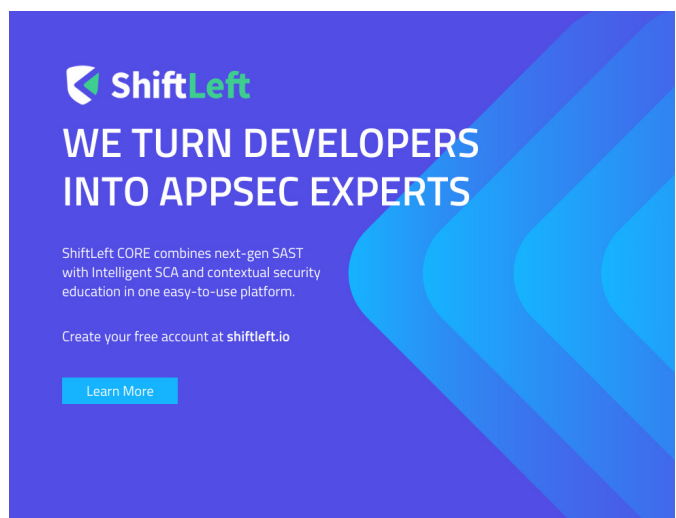
The lint program, originally examining C-based source code, processed in a manner more strictly than the C compiler to identify programming errors, bugs, code-style errors, and suspicious constructs. The lint program would also issue warning and error messages as well to assist the programmer.


Since the late 1970s, static code analysis tooling has continued to evolve and become part of the application security testing (AST) market segment. In fact, linters exist for most modern languages in use today, including interpreted languages (like JavaScript and Python), which do not contain a compiling phase.

## KEY COMPONENTS OF STATIC CODE ANALYSIS

Static code analysis solutions focus on one (or more) of the following aspects of the application under review:

- General vulnerability analysis
- Language and framework security
- Compliance
- End-to-end analysis



 **ShiftLeft**

## WE TURN DEVELOPERS INTO APPSEC EXPERTS

ShiftLeft CORE combines next-gen SAST with Intelligent SCA and contextual security education in one easy-to-use platform.

Create your free account at [shiftleft.io](https://shiftleft.io)

[Learn More](#)



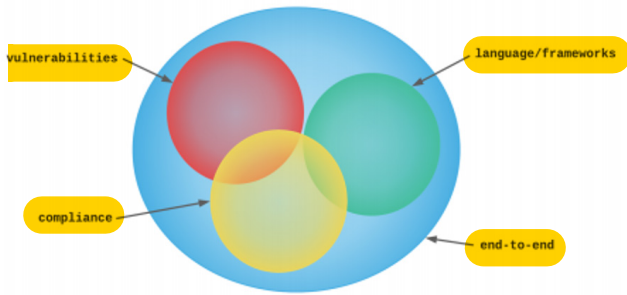
# WE TURN DEVELOPERS INTO APPSEC EXPERTS

ShiftLeft CORE combines next-gen SAST with Intelligent SCA and contextual security education in one easy-to-use platform.

Create your free account at [shiftleft.io](https://shiftleft.io)

[Learn More](#)

The illustration below provides a conceptual view of how these components work together, overlapping slightly, to protect the integrity of the source code being analyzed:



Static code analysis tools often do not focus on every aspect noted above, which is why categories of static code analysis were defined.

### GENERAL VULNERABILITY ANALYSIS

General vulnerability analysis includes the original work completed by the “lint” program, but has been expanded to include: logic flaws, hardcoded secrets, data leaks, authorization bypass, back doors, or logic bombs in the source code.

### LANGUAGE AND FRAMEWORK SECURITY

The language and framework security component focuses on locating items such as: cross-site request forgery (unauthorized requests), session fixation (assume a valid user’s session), and clickjacking (tricking the user into clicking an element disguised as another) within a given application instance.

### COMPLIANCE

The compliance component seeks potential violations with standards like: SOC 2 (Secure Data Management), PCI-DSS (Payment Card Industry/Data Security Standard), GDPR (EU General Data Protection Regulation), and CCPA (California Consumer Privacy Act).

### END-TO-END ANALYSIS

The end-to-end component focuses on customer-facing client tier validation and can also include services and APIs that are available for end-user consumption. While the end-to-end component does not interact with the lower-level categorizations, the results of non-compliance for those categories may surface to this level.

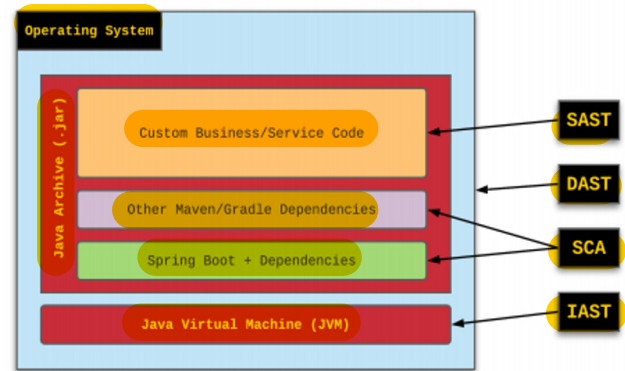
## TYPES OF APPLICATION SECURITY TESTING

To represent and differentiate the application stack, application security testing (AST) is branched into different categorizations:

- **Static AST (SAST):** Focuses on finding as many vulnerabilities in custom source code as possible
- **Dynamic AST (DAST):** Simulates attacks against an application, service, or API, discovering vulnerabilities that are triggerable

- **Interactive AST (IAST):** Focuses on vulnerabilities at the virtual-machine (or runtime) level
- **Software Composition Analysis (SCA):** Identifies similar challenges with open-source or third-party components within an application

Consider the following diagram for a Java-based RESTful microservice written using the Spring Boot framework:



The goal of this document is to concentrate on the AST categorization of static code analysis, focusing on custom code introduced to meet business needs and objectives.

## COMPARISON TO DYNAMIC ANALYSIS (DAST)

While SAST (Static AST) focuses on the custom business logic or service code, tools focused on Dynamic AST (or DAST) perform validations against the application, service, or APIs for vulnerabilities at that layer of the application stack.

DAST tools perform black box (or outside-in) security testing — where the tooling has no understanding of the technologies or frameworks used to create the service being validated. Because of this fact, DAST tooling does not analyze the source code in any way.

Using a DAST solution requires the application be in a running state and can locate run-time vulnerabilities which are not exposed with SAST tools.

The following table is intended to provide a simple comparison between SAST and DAST tooling:

Item	SAST	DAST
Inspects underlying source code	yes	no
Requires the application running	no	yes
Requires understanding of source design	yes	no
Performs hacker-like (black box) testing	no	yes
Utilized in development lifecycle	early	end
Average cost to remediate issues	low	higher

DAST solutions are not intended to replace SAST solutions, and both are required to reduce potential vulnerabilities in the application.

## BENEFITS OF STATIC CODE ANALYSIS (SAST)

The cost for a feature team to participate in a two-week sprint using the Agile methodology ranges between \$60,000 and \$100,000 (USD) — depending on the size and geographical location of the team. With this assumption in mind, time is certainly money when it comes to delivering new functionality.

Static code analysis tools can provide the following benefits to development teams:

- Identify vulnerabilities with a higher degree of effectiveness over human-based analysis
- Allow peer-review to focus on business rules implementation
- Reduce the amount of time required to complete the peer-review phase of development

## CHALLENGES OF STATIC CODE ANALYSIS (SAST)

Until recently, three major roadblocks impeded the necessary implementation levels of static code analysis:

- Static code analysis required an excessive amount of time
- Too many false positives were identified and needed to be marked as “safe”
- Solutions tended to lean on glorified grep patterns instead of purpose-built designs

Because of these limitations, static code analysis efforts were not part of the standard development lifecycle. CI/CD pipelines could not include static code analysis and such efforts were placed in the hands of application security engineers and executed on a less periodic basis. Consequently, this led to results that were often outdated by the time they reached the hands of the feature developer who introduced the custom program code that was identified.

## MODERN STATIC CODE ANALYSIS

For static code analysis tools to be effective, modernization is needed to address the challenges presented above. The next generation of SAST tools being considered should meet the following needs:

- **Performance tuned to run fast, without compromising accuracy.**

Most organizations release code multiple times a day. If code analysis takes hours or days to complete, it is a nonstarter. Also, if the results are riddled with false positives, which take valuable time to triage, developers will not want to use the tool.

- **Underlying design utilizes a comprehensive engine with the ability to recognize all code paths being reviewed.**

Application code is complex. Effective code analysis tools go beyond pattern matching techniques and can find deep rooted vulnerabilities that span across multiple files and could involve complex code paths.

- **Integrates seamlessly with CI/CD pipelines.**

Multiple developers work on different parts of the code at the same time. Security teams want to ensure that vulnerable code never makes it to the main source code branch since that’s the only way to keep the vulnerability from reaching production. For this, code analysis tools need to seamlessly integrate with CI/CD pipelines and enable security teams to implement and enforce the organization’s security policies.

With these needs in place, feature teams will inherently produce better source code, since any items noted by the static code analysis processing will be addressed while the code is fresh in the developer’s mind and before a peer review begins. In fact, a next-generation SAST design will serve as a mechanism to keep vulnerabilities inside the development environment.

With a comprehensive SAST engine, less false positives are expected, and additional vulnerabilities can surface from an understanding of all the paths in the original source code.

## GETTING STARTED WITH STATIC CODE ANALYSIS

Getting started with static code analysis involves seeking out vendors that provide a product in the SAST market. Some current vendors with known CI/CD support are noted below:

- [Checkmarx Software Security Platform](#)
- [HCL Software AppScan](#)
- [MicroFocus Fortify](#)
- [ShiftLeft CORE](#)
- [Veracode Application Analysis](#)

## COMPARING SOLUTIONS WITH OWASP BENCHMARK

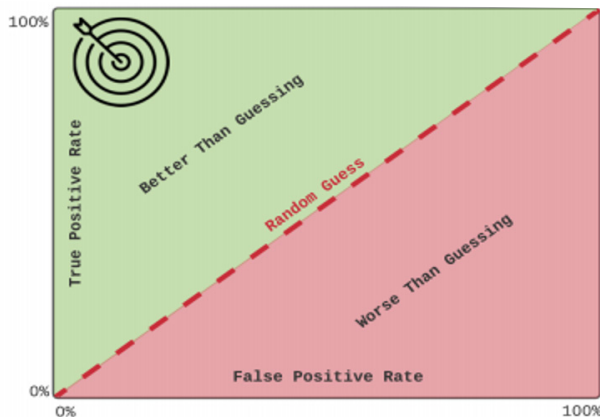
To gain a comparison around the performance and effectiveness of each vendor, the OWASP Benchmark for Security Automation should be utilized.

The OWASP Benchmark is an open and free Java test suite designed to facilitate comparisons of different static code analysis tools. The test suite measures the speed, coverage, and accuracy of vulnerability detection tools and services.

There are four possible test outcomes in the Benchmark:

1. Tool correctly identifies a real vulnerability (True Positive - TP)
2. Tool fails to identify a real vulnerability (False Negative - FN)
3. Tool correctly ignores a false alarm (True Negative - TN)
4. Tool fails to ignore a false alarm (False Positive - FP)

The diagram below represents how the OWASP Benchmark should be interpreted:



Static code analysis tools should strive to reach the target area in the illustration above, where the resulting vulnerability is in the 85-100% range on the true positive rate and 0-15% on the false positive rate.

As part of the analysis of potential SAST vendors, the OWASP Benchmark should be a key part of the decision-making process — even if Java is not the primary language that will utilize the SAST solution at implementation time. This is because the OWASP Benchmark is the most-thorough tool to compare/contrast static code analysis solutions. Keeping track of false positives should be included in the analysis for each vendor under review.

### STATIC CODE ANALYSIS SETUP AND PERFORMANCE EVALUATION

The setup and configuration for each vendor will vary depending on the design of their product. However, once the product itself has been setup and configured, the next step will be to allow the SAST product to access the source code that will be reviewed.

Before taking the time to implement a new solution into your CI/CD tooling, it is always a good idea to determine how long the analysis will take. Simply allowing the tool to scan the code manually will not only validate the functionality of the product with the supplied source code, but also provide basic information regarding the amount of time required to perform the analysis.

Most SAST tools will include full and partial scan functionality. It is a good idea to perform multiple iterations of all available scan modes for product comparison analysis.

### STATIC CODE ANALYSIS RESULTS

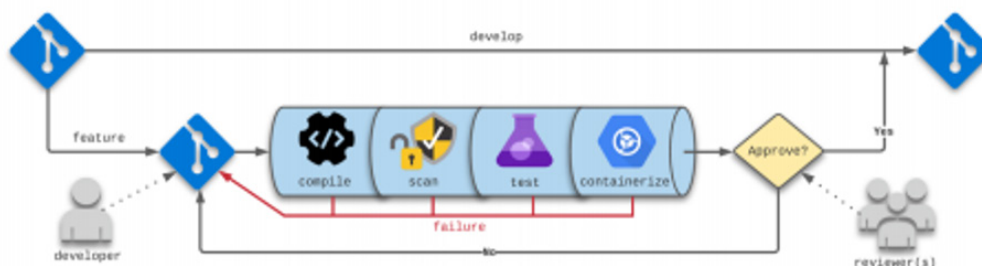
In addition to the OWASP Benchmark results, analysis of the SAST findings should be reviewed to gain an understanding of the gaps reported by all static code analysis tools under review. This would include factors like: overall performance, ease of use and product feature set – which may vary based upon each customer's needs. Maintaining an understanding on how the results differ can also become a factor in the decision-making process.

### STATIC CODE ANALYSIS CI/CD INTEGRATION

The following high-level illustration is intended to present the ideal design for SAST in a feature team's CI/CD pipeline (see image below).

The example above represents the following flow:

1. Developer creates new feature branch off the develop branch.
2. Developer makes changes according to acceptance criteria noted by the Product Owner.
3. Developer commits code and pushes changes to the origin/remote host.
4. The CI/CD solution executes the following pipeline:
  - The compile stage validates the branch can be compiled.
  - The scan stage executes the SAST solution to identify any vulnerabilities.
  - The test stage fires all unit and integration tests.
  - The containerize stage creates the expected container for future usage.
5. Any failures result in a failed build, which requires developer attention.
6. When ready and free of any build errors, the peer review aspect of the development lifecycle begins.



- If approved, the branch is merged into the develop branch.
- If issues exist, they will be addressed by the feature developer making the changes, which will return to step three (above).

When static code analysis tooling is included in the CI/CD pipeline, any issues with the analysis (as defined by the implementation) result in a failed build, which forces the feature developer(s) to address these issues before the changes can be peer reviewed.

## ADVANCED STATIC CODE ANALYSIS (SAST)

While this document is focused on getting started with static code analysis, there are a few advanced topics that evaluators may wish to include in the decision-making process:

- **Mute Unreachable Paths:** In cases where an open-source framework issue is identified, but not able to be immediately addressed, advanced SAST tooling can suggest manners in which the known-path can be blocked from use. This allows the framework to remain in place and keep the application secure.
- **SaaS Models Available:** With a service-based approach for static code analysis, there are no additional hardware needs or licensing costs required to implement the SAST solution.
- **API & Flow Customization:** The static code analysis tool provides an API into the results found. This allows direct access to the data for situations where the default UI is not preferred, allowing customization to static code analysis remediation flows.

## CONCLUSION

Static code analysis is a vital requirement for all teams producing features and functionality for customer-facing products, services, and APIs. At the minimum, SAST solutions should be part of the development lifecycle, participating in the CI/CD pipeline and utilized as part of the peer review process.

While quality solutions require a cost investment, analysis of the cost to attempt manual checks during peer-reviews should be taken into account.

Most likely, the cost to maintain static code analysis will be easy to justify. When seeking a static code analysis solution in your organization, the following questions should be considered and ranked for each vendor under review:

- Is the user-interface for the solution easy to use and effective?
- Does your organization adhere to any compliance regulations?
- Does the solution support CI/CD integration?
- Does the solution's scan time introduce any challenges in the development lifecycle?
- What is the vendor's OWASP Benchmark for Security Automation Score?
- What is the core design behind the SAST scan engine?
- How are false positives remediated?
- What advanced features are required by your organization?



### WRITTEN BY JOHN VESTER,

SOFTWARE ARCHITECT, CLEANSLATE TECHNOLOGY GROUP  
 Freelance Writer and Member of DZone Core

Information Technology professional with 25+ years expertise in application development, project management, enterprise integration and team management. Currently focusing on enterprise architecture/application design/continuous delivery utilizing object-oriented programming languages and frameworks. Prior expertise building Java-based APIs against React and Angular client frameworks. Integration architecture and design. CRM design and customization. Additional experience using both C# (.NET Framework) and J2EE (Spring, plus various other frameworks).



DZone, a Devada Media Property, is the resource software developers, engineers, and architects turn to time and again to learn new skills, solve software development problems, and share their expertise. Every day, hundreds of thousands of developers come to DZone to read about the latest technologies, methodologies, and best practices. That makes DZone the ideal place for developer marketers to build product and brand awareness and drive sales. DZone clients include some of the most innovative technology and tech-enabled companies in the world including Red Hat, Cloud Elements, Sensu, and Sauce Labs.

Devada, Inc.  
 600 Park Offices Drive  
 Suite 150  
 Research Triangle Park, NC 27709

888.678.0399 | 919.678.0300

Copyright © 2021 Devada, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means of electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.