Static Application Security Testing (SAST)
Tools Evaluation Report

Comparative and Practical Study of SAST Tools

Submitted by: **Soham Mukherjee**

2. Executive Summary

Static Application Security Testing (SAST), often referred to as "white-box testing," is a testing methodology that analyzes source code to find security vulnerabilities that make an organization's applications susceptible to attack. SAST scans an application before the code is compiled, inspecting the code at rest.

In the modern DevSecOps landscape, SAST is critical because it allows for early detection of vulnerabilities (Shift-Left), reducing the cost and complexity of remediation. By integrating SAST into the CI/CD pipeline, organizations can enforce security standards and ensure that code committed to the repository is free from known vulnerabilities like SQL Injection, Cross-Site Scripting (XSS), and buffer overflows.

The purpose of this report is to evaluate leading SAST tools—ranging from enterprise-grade solutions to lightweight open-source scanners—to determine their efficacy, integration capabilities, and suitability for different development environments.

3. Introduction to SAST

Definition: SAST is a set of technologies designed to analyze application source code, byte code, and binaries for coding and design conditions that are indicative of security vulnerabilities.

How SAST Works:

- **Abstract Syntax Tree (AST):** Tools parse code into an AST to understand the structure and semantics of the program.
- **Taint Analysis:** Tracks data flow from untrusted sources (sinks) to sensitive areas of the application to identify injection flaws.
- **Pattern Matching:** Uses predefined rules to identify known insecure coding practices.

Role in SDLC: SAST is the cornerstone of the "Shift-Left" security philosophy. Unlike Dynamic Application Security Testing (DAST), which requires a running application, SAST can be performed as the developer writes code (via IDE plugins) or during the build process, enabling immediate feedback and faster iteration cycles.

4. SAST Tools Covered

This report provides a comprehensive analysis of the following tools, categorized by their market positioning and primary use cases:

- SonarQube (Code Quality & Security)
- Checkmarx SAST (Enterprise Security)
- Veracode Static Analysis (Cloud-Native Platform)

- Fortify Static Code Analyzer (Enterprise Depth)
- Synopsys Coverity (High-Accuracy Analysis)
- Klocwork (C/C++ & Embedded Systems)
- Semgrep (Lightweight & Customizable)
- CodeQL (Semantic Code Analysis)
- Bandit (Python Specific)
- Brakeman (Ruby on Rails Specific)
- ESLint Security Rules (JavaScript/TypeScript)

5. In-Depth Tool Evaluation

SonarQube

Overview & Purpose: A widely used open-source platform for continuous inspection of code quality, which includes basic to intermediate security scanning capabilities.

Detection Methodology: Combines pattern matching and data flow analysis to detect bugs, code smells, and security vulnerabilities.

Supported Languages: 29+ languages including Java, C#, Python, JavaScript, TypeScript, C++, and Go.

Rule Engine: Uses Quality Profiles. Users can activate/deactivate rules. Supports custom rules via plugins.

Integration: Excellent integration with Jenkins, Azure DevOps, GitLab CI. IDE integration via SonarLint.

Strengths: Great UI/UX, combines quality and security, massive community support, easy setup.

Limitations: Security analysis is not as deep as specialized SAST tools; higher false negative rate for complex taint analysis.

Ideal Use Case: General-purpose development teams needing a balance of code quality and security.

Linux / CLI Commands:

```
# Installation / Setup
docker run -d --name sonarqube -p 9000:9000 sonarqube

# Execution / Scan
sonar-scanner \
  -Dsonar.projectKey=my_project \
  -Dsonar.sources=. \
  -Dsonar.host.url=http://localhost:9000 \
  -Dsonar.login=my_token
```

Checkmarx SAST (CxSAST)

Overview & Purpose: An enterprise-grade SAST solution known for its high accuracy and ability to scan uncompiled code.

Detection Methodology: Deep data flow analysis (taint tracking) across the entire application logic.

Supported Languages: Extensive support for 25+ languages and frameworks, including mobile (iOS/Android) and legacy languages.

Rule Engine: Highly customizable CxQL (Checkmarx Query Language) allows security teams to write complex custom queries.

Integration: Integrates seamlessly with all major CI/CD pipelines, IDEs, and issue trackers (Jira).

Strengths: Scans uncompiled code, extremely powerful query language, low false positive rate with proper tuning.

Limitations: High licensing cost, steep learning curve for CxQL, resource-intensive scans.

Ideal Use Case: Large enterprises with strict compliance requirements and complex, multi-language codebases.

Linux / CLI Commands:

```
# Installation / Setup
# Assumes cx-cli is downloaded and in PATH
chmod +x cx-cli

# Execution / Scan
cx scan create \
  --project-name 'MyApp' \
  --sast-preset 'Checkmarx Default' \
  --source ./src \
  --report-pdf=report.pdf
```

Veracode Static Analysis

Overview & Purpose: A cloud-based SaaS platform that analyzes binary code (compiled) rather than source code, offering a holistic view of the application.

Detection Methodology: Binary static analysis (patented technology) which creates a model of the application's control and data flow.

Supported Languages: Java, .NET, C/C++, JavaScript, Python, PHP, Ruby, Mobile binaries.

Rule Engine: Managed by Veracode; less user-customizable than source-based tools but highly curated for accuracy.

Integration: API-driven integrations with CI/CD, IDE plugins, and pipeline scan capabilities.

Strengths: No need to expose source code (binary scan), extremely low false positives, scalable SaaS model.

Limitations: Slower turnaround time due to cloud upload/scan process compared to local scanners; requires buildable artifacts.

Ideal Use Case: Organizations prioritizing vendor-managed security and third-party code verification.

Linux / CLI Commands:

```
# Installation / Setup
curl -O https://downloads.veracode.com/securityscan/pipeline-scan-LATEST.zip
unzip pipeline-scan-LATEST.zip

# Execution / Scan
java -jar pipeline-scan.jar \
  --file myapp.war \
  --project_name 'MyApp' \
  --fail_on_severity 'Very High, High'
```

Fortify Static Code Analyzer

Overview & Purpose: A veteran enterprise tool by OpenText (formerly Micro Focus) offering comprehensive vulnerability detection.

Detection Methodology: Uses multiple analyzers (data flow, semantic, structural, configuration) to pinpoint vulnerabilities.

Supported Languages: Broadest language support in the industry (27+), including obscure legacy languages.

Rule Engine: Highly configurable. Security teams can write custom rules to match specific internal frameworks.

Integration: Audit Workbench provides a rich interface for triage. Robust CI/CD and IDE plugins.

Strengths: Depth of analysis is unmatched; enterprise-grade reporting and compliance mapping.

Limitations: Can be slow on large codebases; complex setup and maintenance; high cost.

Ideal Use Case: Government, Defense, and Banking sectors where depth of analysis is paramount.

Linux / CLI Commands:

```
# Installation / Setup
./Fortify_SCA_and_Apps_Linux.run # Enterprise installer

# Execution / Scan
# 1. Clean
sourceanalyzer -b mybuild -clean
# 2. Translate
sourceanalyzer -b mybuild javac *.java
# 3. Scan
```

```
sourceanalyzer -b mybuild -scan -f results.fpr
```

Synopsys Coverity

Overview & Purpose: Renowned for its precision in identifying critical quality and security defects, particularly in C/C++.

Detection Methodology: Deep semantic analysis and interprocedural data flow analysis.

Supported Languages: Strong focus on C/C++, C#, Java, JavaScript, Python, Ruby.

Rule Engine: Advanced configuration options for aggressive or conservative analysis modes.

Integration: Integrates with build systems (Make, CMake, Maven, Gradle) and CI servers.

Strengths: Lowest false positive rate in the industry for C/C++; creates a complete build map for accuracy.

Limitations: Initial configuration can be complex; primarily optimized for compiled languages.

Ideal Use Case: Embedded systems, automotive, and critical infrastructure software development.

Linux / CLI Commands:

```
# Installation / Setup
./cov-analysis-linux64.sh # Enterprise installer

# Execution / Scan
# Capture Build
cov-build --dir cov-int make
# Analyze
cov-analyze --dir cov-int
# Commit Defects
cov-commit-defects --dir cov-int --stream my_stream --url http://coverity_server:8080
```

Klocwork

Overview & Purpose: A static analysis tool optimized for DevOps, specializing in C, C++, C#, and Java.

Detection Methodology: Differential analysis engine that allows for instant analysis of changed files only.

Supported Languages: C, C++, C#, Java, JavaScript, Python.

Rule Engine: Complies with MISRA, AUTOSAR, CERT, and CWE standards out of the box.

Integration: Designed for continuous integration; supports incremental analysis to speed up pipeline builds.

Strengths: Fast incremental scans; excellent for compliance (MISRA/AUTOSAR) in embedded development.

Limitations: Less focus on web application vulnerabilities compared to Checkmarx or Veracode.

Ideal Use Case: Embedded software, IoT, and automotive industries requiring rigorous compliance.

Linux / CLI Commands:

```
# Installation / Setup
./kw-server-installer.sh

# Execution / Scan
kwcheck create -u user -b build_spec
kwcheck run
```

Semgrep

Overview & Purpose: A modern, lightweight, open-source static analysis engine designed to be fast and developer-friendly.

Detection Methodology: Syntactic pattern matching that looks like source code. It runs offline and on uncompiled code.

Supported Languages: Go, Java, JavaScript, JSON, Python, Ruby, TypeScript, Terraform, and more.

Rule Engine: Rules look like the code you are writing. Extremely easy to write custom rules in YAML.

Integration: Runs in CI/CD in seconds; pre-commit hooks; integrates with GitHub Actions natively.

Strengths: Blazing fast, open-source core, highly accessible rule syntax, developer-loved.

Limitations: Data flow analysis is less mature than enterprise tools (though improving with the Pro engine).

Ideal Use Case: Modern DevSecOps teams, startups, and CI/CD pipelines requiring speed.

Linux / CLI Commands:

```
# Installation / Setup
pip install semgrep
```

```
# Execution / Scan
semgrep scan --config=auto . --json > report.json
```

GitHub CodeQL

Overview & Purpose: A semantic code analysis engine developed by GitHub to query code as if it were data.

Detection Methodology: Treats code as a relational database. Vulnerabilities are found by writing queries against the database.

Supported Languages: C/C++, C#, Go, Java, JavaScript, TypeScript, Python, Ruby, Swift.

Rule Engine: Extremely powerful query language (QL). Community-driven queries are constantly updated.

Integration: Native integration with GitHub Advanced Security; runs via GitHub Actions.

Strengths: Deep semantic understanding; free for open source; immense library of community queries.

Limitations: Steep learning curve for QL; requires compilation for compiled languages; enterprise features cost money.

Ideal Use Case: Teams hosted on GitHub and security researchers looking for deep logic flaws.

Linux / CLI Commands:

```
# Installation / Setup
gh extension install github/gh-codeql # via GitHub CLI

# Execution / Scan
# 1. Create Database
codeql database create ./qldb --language=python
# 2. Analyze
codeql database analyze ./qldb --format=sarif-latest --output=results.sarif
```

Bandit

Overview & Purpose: A tool designed specifically to find common security issues in Python code.

Detection Methodology: AST-based static analysis that processes each file to build an AST and runs plugins against it.

Supported Languages: Python only.

Rule Engine: Configurable through profiles and exclusion lists (baseline).

Integration: Pip installable; integrates easily into tox, pre-commit, and CI pipelines.

Strengths: Lightweight, Python-native, easy to use, free and open source.

Limitations: Limited to Python; relatively simple analysis (no deep taint tracking).

Ideal Use Case: Python projects needing a quick, baseline security check.

Linux / CLI Commands:

```
# Installation / Setup
pip install bandit

# Execution / Scan
bandit -r ./src -f json -o output.json
```

Brakeman

Overview & Purpose: A static analysis tool which checks Ruby on Rails applications for security vulnerabilities.

Detection Methodology: Scans the Rails application flow to identify dangerous method calls and mass assignment issues.

Supported Languages: Ruby on Rails.

Rule Engine: Specific checks for Rails versions; allows ignoring false positives via configuration files.

Integration: Fast enough to run on every commit; zero configuration required for standard Rails apps.

Strengths: Zero-configuration; specialized knowledge of Rails internals; very fast.

Limitations: Limited to Ruby on Rails; cannot scan generic Ruby scripts effectively.

Ideal Use Case: Any Ruby on Rails web application.

Linux / CLI Commands:

```
# Installation / Setup
gem install brakeman

# Execution / Scan
brakeman -o report.html
```

ESLint (Security Plugins)

Overview & Purpose: A pluggable linting utility for JavaScript and JSX, extended with security plugins.

Detection Methodology: AST-based pattern matching to identify problematic patterns.

Supported Languages: JavaScript, TypeScript.

Rule Engine: Fully customizable via .eslintrc files. Huge ecosystem of plugins.

Integration: Ubiquitous in JS ecosystem; integrates with every IDE and build system.

Strengths: Already part of the workflow for most JS devs; zero friction adoption.

Limitations: Not a dedicated security tool; misses complex data flow vulnerabilities.

Ideal Use Case: Frontend and Node.js teams ensuring basic hygiene and code quality.

Linux / CLI Commands:

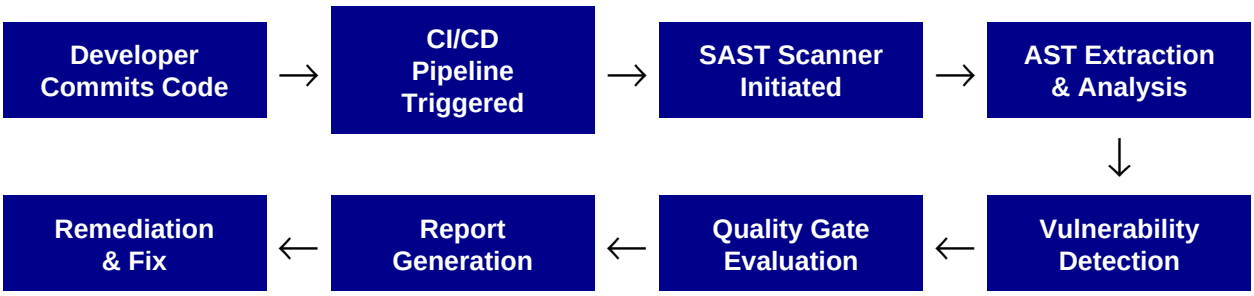
```
# Installation / Setup
npm install eslint eslint-plugin-security --save-dev

# Execution / Scan
npx eslint . --ext .js,.jsx
```

6. SAST Workflow (Textual Representation)

The following steps outline a standard automated SAST workflow in a DevSecOps pipeline:

1. **Step 1: Code Commit**
   Developer pushes code changes to the Version Control System (e.g., Git).
2. **Step 2: CI Trigger**
   The CI/CD server (Jenkins, GitLab CI, etc.) detects the commit and initiates the build pipeline.
3. **Step 3: SAST Scan Initiation**
   The SAST tool is invoked. It checks out the source code (or build artifacts).
4. **Step 4: Analysis & Rule Matching**
   The tool builds an Abstract Syntax Tree (AST) or Control Flow Graph (CFG) and applies security rules/queries against the model.
5. **Step 5: Vulnerability Detection**
   Potential vulnerabilities are flagged (e.g., SQLi, XSS, Hardcoded Secrets).
6. **Step 6: Gate Evaluation**
   The pipeline evaluates the results against a quality gate (e.g., 'Fail build if High Severity > 0').
7. **Step 7: Reporting**
   A report is generated (PDF/JSON/SARIF) and uploaded to the dashboard or sent to the developer.
8. **Step 8: Remediation**
   The developer reviews the findings, fixes the valid issues, and marks false positives.

Visual Process Flow:



7. Comparative Analysis

Enterprise vs. Developer-Centric:

Enterprise tools (Fortify, Checkmarx) offer superior depth, compliance reporting, and language support but come with high costs and slower scan times. Developer-centric tools (Semgrep, Snyk, SonarQube) prioritize speed, IDE integration, and usability, often trading off some depth of analysis for immediate feedback.

Accuracy vs. Speed:

Tools performing deep interprocedural taint analysis (Coverity, Veracode) require more time and computational resources, making them better suited for nightly builds. Syntactic tools (Semgrep, ESLint) are instantaneous and fit for pre-commit hooks.

8. Challenges & Limitations

False Positives: The most significant challenge in SAST. Tools often flag secure code as vulnerable because they lack runtime context. This causes 'alert fatigue' among developers.

Lack of Runtime Context: SAST cannot detect configuration issues, authentication logic flaws, or issues that only manifest when the application is running (covered by DAST).

Performance Overhead: Deep scans on monolithic codebases can take hours, creating bottlenecks in the CI/CD pipeline.

Rule Maintenance: Custom rules are often necessary for proprietary frameworks, requiring specialized knowledge to maintain.

9. Conclusion

No single SAST tool is a silver bullet. The optimal strategy often involves a layered approach: utilizing lightweight, fast scanners (like Semgrep or SonarQube) for immediate developer feedback within the IDE and PR checks, while reserving deep, enterprise-grade scanners (like Checkmarx or Veracode) for nightly builds or release gates.

For a successful DevSecOps implementation, the focus must shift from simply "finding bugs" to "fixing bugs" by integrating these tools seamlessly into the developer's existing workflow and minimizing false positives.

Thank you