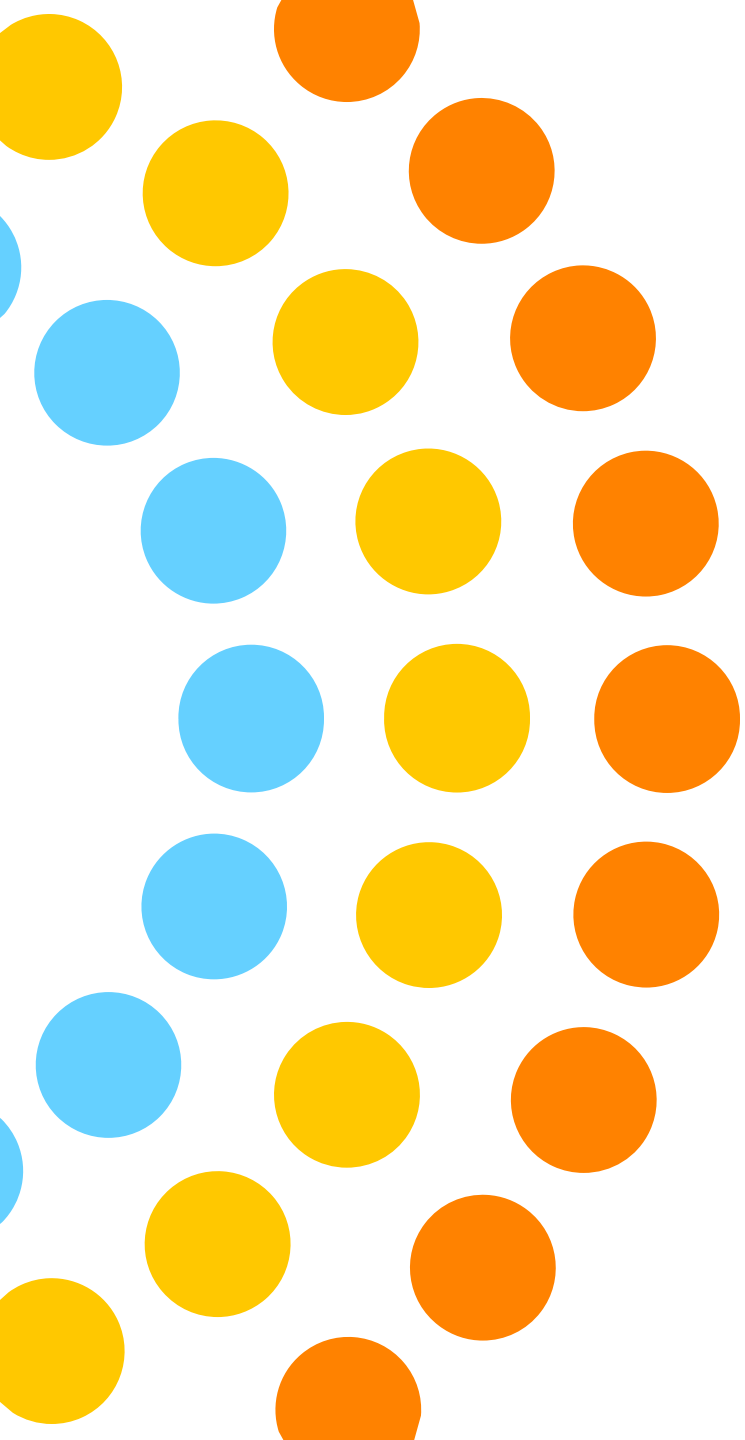
# Vulnerability Detection with CodeQL

Confoo 2025

DILITRUST



# Vulnerability Detection with CodeQL

## A Deep Dive into Code Analysis for Secure Software

Alexis Agahi

DILITRUST

# Needs

Security compliance requirements
ISO27001 & OWASP recommandations

How can we enforce security in our codebase?




DiliTrust

# How?

Code **reviews**

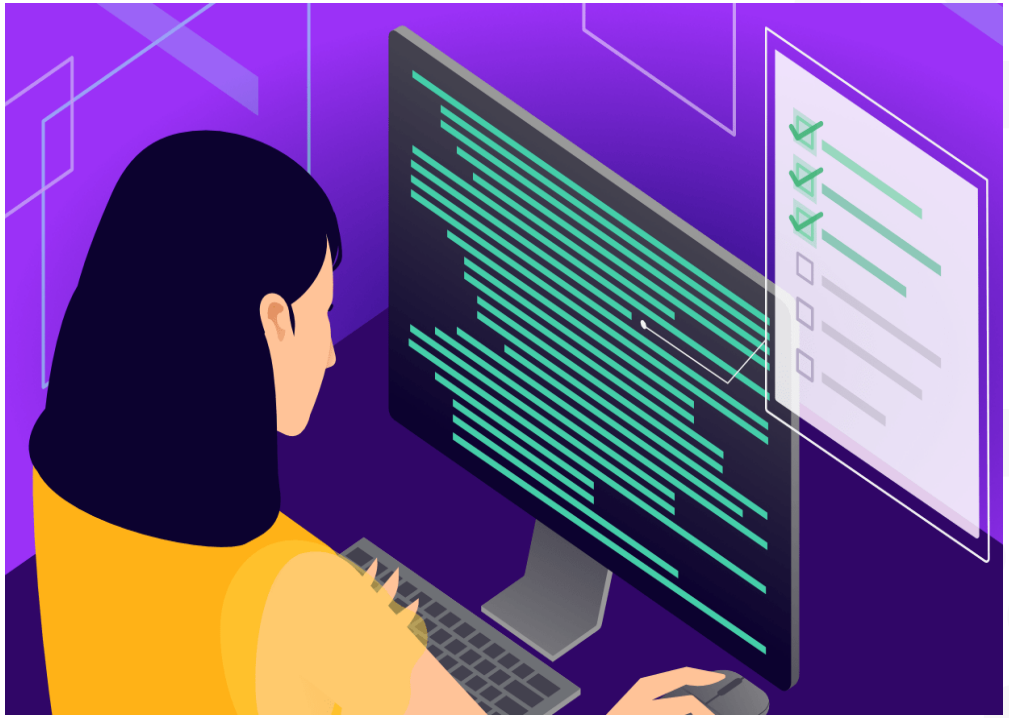
Code **audit**, **bug bounty** and **pentest** campaigns

Code Analysis

Static Code Security Analysis (**SAST**)

Dynamic Code Security Analysis (**DAST**)

Hybrid Security Analysis (combining SAST and DAST)



# Static Code Security Analysis (SAST)

**Lexical Analysis** - Detecting password="12345" in the code.

**Syntax & Semantic Analysis** - Example: identifies insecure coding patterns.

**Data Flow Analysis** - User input reaching eval() without sanitization.

**Control Flow Analysis** - Identifying unreachable authentication checks.

**Taint Analysis** - Detecting SQL injection or XSS vulnerabilities.

**Pattern-Based Analysis (Rule-Based)** - Identifying strcpy() (unsafe function) in C code.

**SAST Tools** - CodeQL, Semgrep, Bearer, Joern.

# Dynamic Code Security Analysis (DAST)

**Fuzz Testing (Fuzzing)** - Crashing a web server by sending unexpected input.

**Runtime Taint Analysis** - Detecting unsanitized input reaching exec().

**Memory Analysis** - Identifying heap overflow exploits.

**Execution Tracing & Profiling** - Detecting unauthorized API calls.

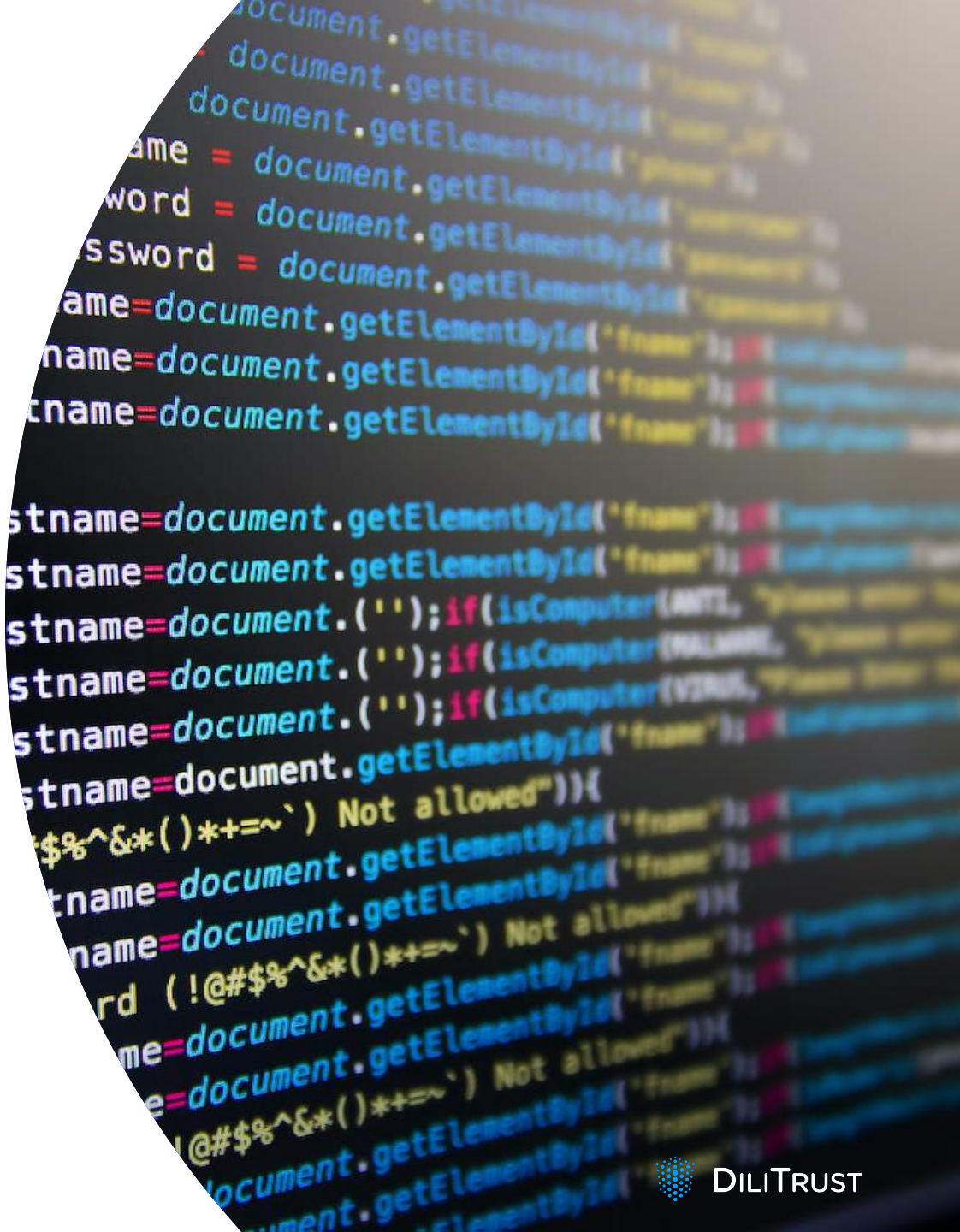
**DAST Tools**

Burp Suite - Web application penetration testing

OWASP ZAP - Automated security testing for web apps

Valgrind / AddressSanitizer (ASan) - Memory safety analysis

# CodeQL




DiliTrust

# CodeQL

Data Flow - Taint Analysis - Control Flow Analysis

Custom Query Language (QL) – Lets you write custom security rules to detect specific vulnerabilities

Supports Multiple Languages: C, C++, C#, Java, JavaScript, TypeScript, Python, Ruby, Go


Free for OpenSource Projects

GitHub Advanced Security (GHAS) is required to use CodeQL in private repositories.

# Why CodeQL for Security?

Finds security vulnerabilities at scale

Used by GitHub Advanced Security & OpenSource Security Coalition

Database of pre-build vulnerability queries (more than 1200) and community pack

https://github.com/GitHubSecurityLab/CodeQL-Community-Packs/


DILITRUST

# CodeQL Architecture

**CodeQL Extractor Parses the Source Code**

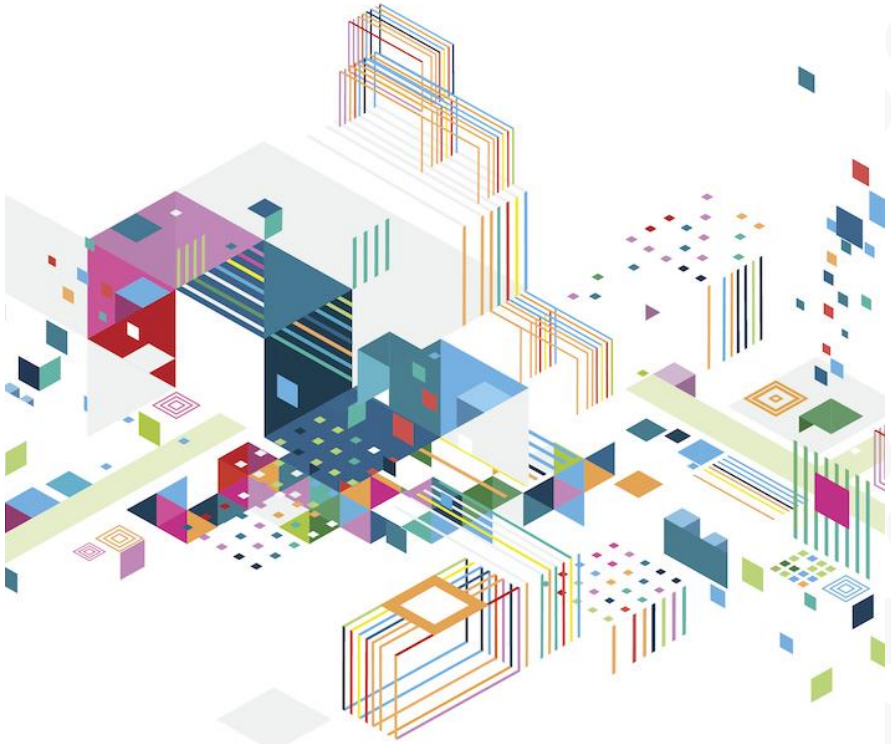
Abstract Syntax Tree (**AST**) and a Control Flow Graph (**CFG**).

**Converts Code into a Relational Database**

The extracted code is stored as a CodeQL database consisting of multiple tables.

**Runs CodeQL Queries on the Database**

Security rules are written in QL (Query Language) and executed against the database.



# Understanding CodeQL

Querying code as a database

Having this JavaScript code

```
console.log("Any logs");
const PASSWORD = "password123";
console.log("More logs");
```

This CodeQL will detect hardcoded password :)

```
import javascript

from Literal l
where l.getValue() = "password123"
select l, "Hardcoded password detected!"
```

# Abstract Syntax Tree (AST)

## Lexical Analysis

CodeQL first tokenizes the JavaScript source code into individual components (tokens).

Tokens are basic building blocks like keywords, identifiers, literals, and operators.

```
function greet(name) {
    eval("console.log(" + name + ")");
}
```

| Token | Type |
|---|---|
| function | Keyword |
| greet | Identifier |
| ( | Symbol |
| name | Identifier |
| ) | Symbol |
| { | Symbol |
| eval | Identifier |
| ( | Symbol |
| "console.log(" | String |
| + | Operator |
| name | Identifier |
| + | Operator |
| ")" | String |
| ); | Symbol |
| } | Symbol |

# Abstract Syntax Tree (AST)

## Parsing (Building the AST)

CodeQL parses the tokens into a tree structure (**AST**).

Each node in the **AST** represents a JavaScript construct (e.g., function, variable, expression).

The **AST** defines parent-child relationships between different code elements.

```
FunctionDeclaration
├── Identifier (name: "greet")
├── Parameters
│   └── Identifier (name: "name")
└── BlockStatement
    └── ExpressionStatement
        └── CallExpression
            ├── MemberExpression
            ├── Identifier (name: "eval")
            └── Arguments
                └── BinaryExpression
                    ├── Literal (value: "console.log(")
                    ├── Identifier (name: "name")
                    ├── Literal (value: ")")
```

# Abstract Syntax Tree (AST)

## Storing AST in a CodeQL Database

CodeQL extracts the AST into a relational database.

Each JavaScript element (functions, variables, expressions, etc.) becomes a table in the database.

| Table Name | Description |
|---|---|
| **functions** | Stores function declarations |
| **calls** | Stores function calls |
| **expressions** | Stores expressions (e.g., binary, logical) |
| **identifiers** | Stores variable and function names |
| **literals** | Stores string and number literals |

```
function greet(name) {
    eval("console.log(" + name + ")");
}
```

```
import javascript

from CallExpr call
where call.getCalleeName() = "eval"
select call, "Avoid using eval() due to security
risks."
```

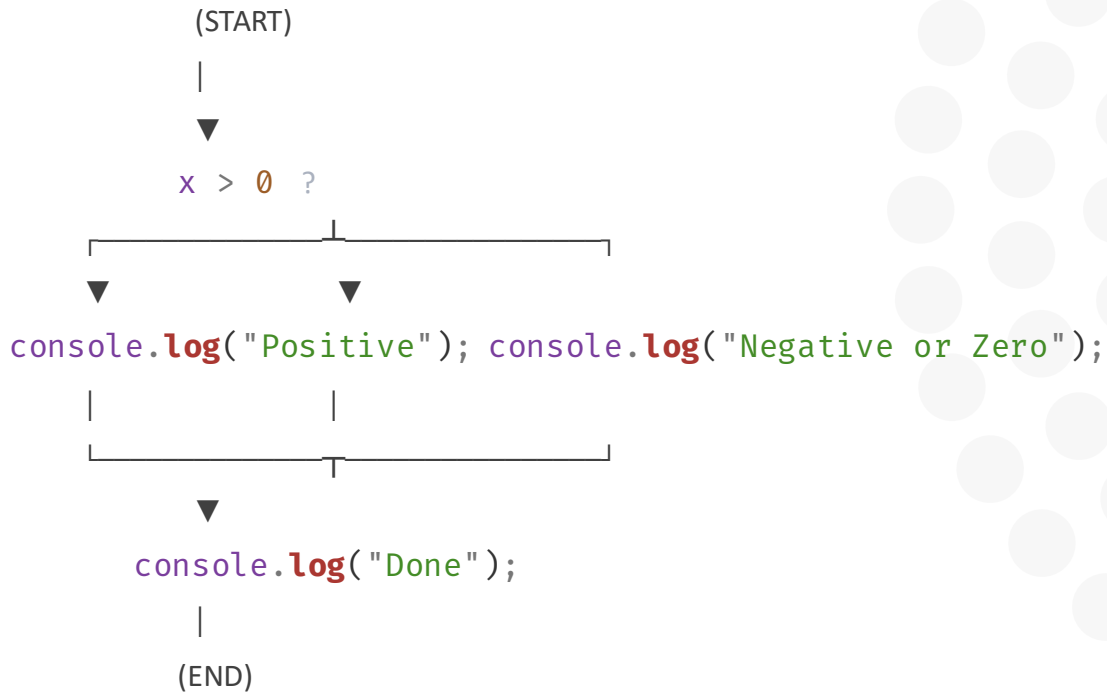
This query scans the AST for eval() calls and reports them as a security risk.

# Control Flow Graph (CFG)

Control Flow Graph (CFG) is a graph-based representation of the execution flow of a program.

```
function check(x) {
    if (x > 0) {
        console.log("Positive");
    } else {
        console.log("Negative or Zero");
    }
    console.log("Done");
}
```

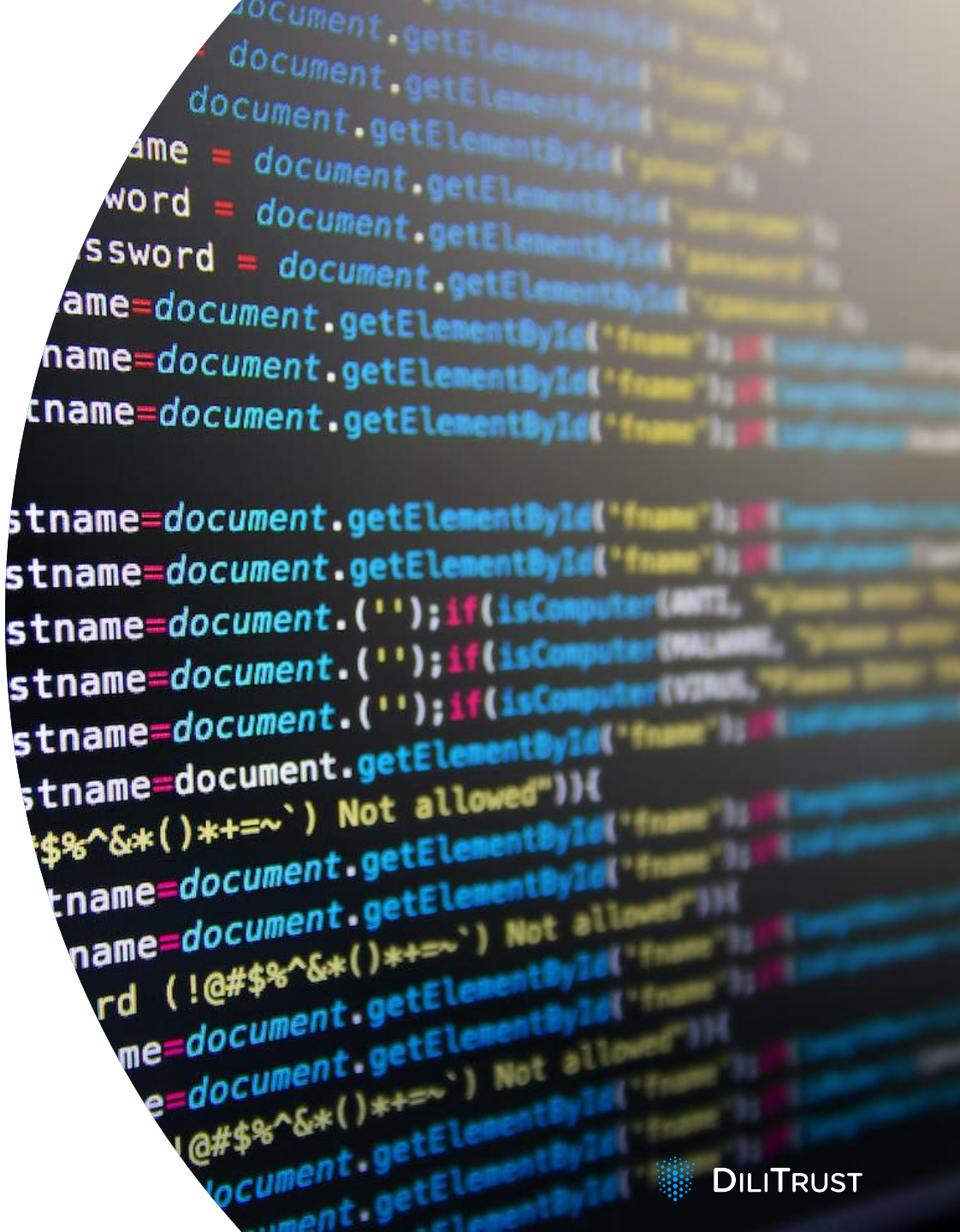
```
import javascript

from BlockStmt deadCode
where not exists(deadCode.getAPredecessor())
select deadCode, "This block is unreachable."
```






DiliTrust

# Hands On




DiliTrust

# Install and run CodeQL

CLI Demo

## Installing CodeQL CLI & VS Code Extension

```
brew install codeql
```

## Setting up a CodeQL database

```
codeql database create confoo25 --language=javascript --source-root=./src
```

## Running queries

```
codeql query run --database= confoo25 my-query.ql
```

## Running community specific queries

```
codeql database analyze confoo25 --download codeql/javascript-queries  --format=sarif-latest --output=results.sarif
```

# Integrating CodeQL in CI

## Jenkins Integration

```
pipeline {
    agent any
    stages {
        stage('Checkout Code') {
            steps {
                checkout scm
            }
        }
        stage('CodeQL Analysis') {
            steps {
                sh '''
                    codeql database create codeql-db --language=java --source-root=.
                    codeql analyze codeql-db --format=sarif --output=codeql-results.sarif
                '''
            }
        }
        stage('Archive Results') {
            steps {
                archiveArtifacts artifacts: 'codeql-results.sarif', fingerprint: true
            }
        }
    }
}
```

# CodeQL Query Basics

## CodeQL syntax overview

```
import javascript // Import the JavaScript CodeQL library

from Function f // Define a variable (f) representing a function
where f.getName() = "myFunction" // Filter functions by name
select f, "This is a function named myFunction."
```

```
import javascript

from Variable v // Define a variable (v) representing a variable
where v.isGlobal() // Filter variable by scope (Global)
select v, "Global variable found: " + v.getName()
```

### Demo: Finding unused functions

Javascript

```
function usedFunction() {
     console.log("This function is used.");
}
function unusedFunction() {
     console.log("This function is never called.");
}
usedFunction(); // Only this function is called
```

CodeQL

```
import javascript

from Function f
where not exists(f.getCallSignature())
select f, "This function is never called: " + f.getName()
```

# Detecting Common Vulnerabilities

## Cross-Site Scripting (XSS)

```
<html><body>
<script>
// ⚠️ Get 'username' parameter from the URL (User-controlled input)
const urlParams = new URLSearchParams(window.location.search);
const username = urlParams.get("username");

if (username) { // ⚠️ Dangerous: Directly inserting user input into innerHTML
    document.getElementById("greeting").innerHTML = "<h1>Welcome, " + username + "!</h1>";
}
</script>
<div id="greeting"></div> <!-- User input inserted here -->
</body></html>
```

```
import javascript
from CallExpr assignment, Expr userInput
where assignment.getCalleeName() = "innerHTML" and // Target innerHTML assignment
      userInput = assignment.getArgument(0) and // Check what is being assigned
      userInput instanceof VariableAccess // Ensure it comes from a variable (e.g., URL input)
select assignment, "🚨 Possible XSS: User-controlled input is assigned to innerHTML."
```

# Detecting Common Vulnerabilities

## SQL Injection

```
const express = require('express');
const mysql = require('mysql2');
const app = express();
const db = mysql.createConnection({ host: "localhost", user: "user", password: "password", database: "db", insecureAuth: true });

app.get('/user', (req, res) => {
    const userId = req.query.id; // ⚠️ User-controlled input
    const query = "SELECT * FROM users WHERE id = " + userId; // ⚠️ Direct concatenation
    db.query(query, (err, result) => { if (err) throw err; res.send(result); });
});
app.listen(3000);
```

```
import javascript

from CallExpr queryCall, Expr queryParam, Expr query
where
    queryCall.getCalleeName() = "query" and
    queryParam = queryCall.getArgument(0) and
    query = queryParam.(VarRef).getVariable().getAnAssignedExpr() and
    query instanceof BinaryExpr
select queryCall, "🚨 Possible SQL Injection: User input is directly concatenated into a SQL query."
```

# Detecting Common Vulnerabilities

## SQL Injection

```
const express = require('express');
const mysql = require('mysql2');
const app = express();
const db = mysql.createConnection({ host: "localhost", user: "user", password: "password", database: "db", insecureAuth: true });

app.get('/user', (req, res) => {
    const userId = parseInt(req.query.id); // ✅ Input is converted to an integer (safe)
    const query = "SELECT * FROM users WHERE id = " + userId; // ⚠️ Direct concatenation
    db.query(query, (err, result) => { if (err) throw err; res.send(result); });
});
app.listen(3000);
```

# Tainted Flow Analysis

## Taint Propagation and Flow State

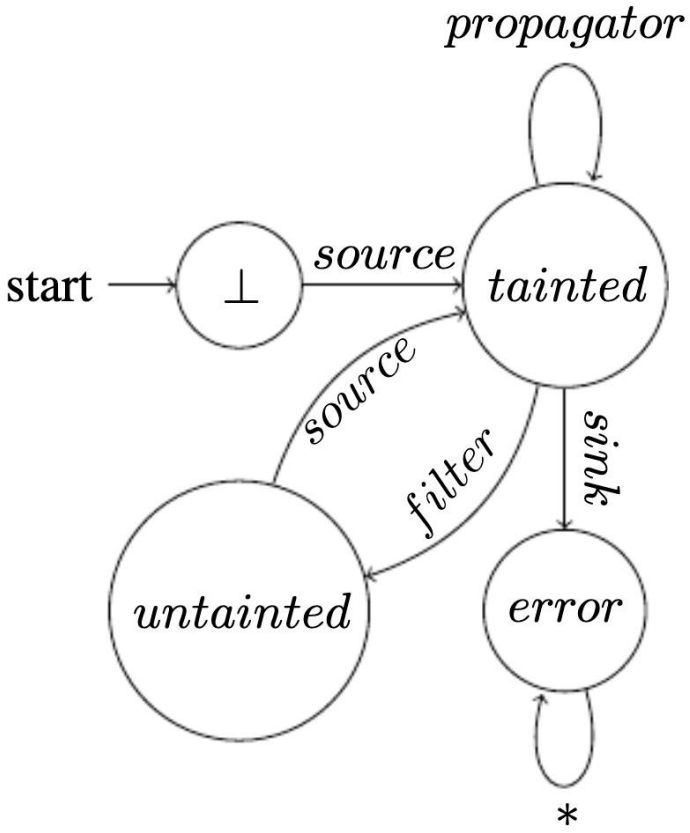
Certain data **sources** must be treated as potentially dangerous until proven safe.

The system tracks these **tainted** values through various program operations

**Source** identification: Entry points where untrusted data enters the system (example: user inputs, network requests, file reads)

**Sink** detection: Critical locations where tainted data could cause harm (example: database queries, system commands, HTML output)

**Propagation**: How **taint** spreads through operations (example: string concatenation, object property assignment, and function calls) or get **sanitized**






DILITRUST

# Tainted Flow Analysis

## CodeQL Example

```
function buildStructure(input) {
     return input + 1;
}

void processStructure(s) {
    sink(s); // ⚠️ sink function that could cause harm
}

void getAndProcess() {
    const userInput = userInput(); // ⚠️ source provides tainted value
    const structure = buildStructure(userInput);
    // ⚠️ structure may be tainted by userInput
    processStructure(structure);
}
```

```
import javascript
import DataFlow
import DataFlow::PathGraph

module Config implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    exists(CallExpr ce |
      ce.getCalleeName() = "userInput" and
      source.asExpr() = ce
    )
  }
  predicate isSink(DataFlow::Node sink) {
    exists(CallExpr ce |
      ce.getCalleeName() = "sink" and
      sink.asExpr() = ce.getArgument(0)
    )
  }
}

module Flow = DataFlow::Global<Config>;
from Flow::PathNode source, Flow::PathNode sink
where Flow::flowPath(source, sink)
select sink.getNode(), source, sink, "🚨 Direct flow from source to sink!"
```

# Tainted Flow Analysis

## CodeQL Example

```
function buildStructure(input) {
    return sanitize(input + 1); // ✅ sanitize the state
}
void processStructure(structure) {
    sink(structure); // sink function that could cause harm
}
void getAndProcess() {
    const userInput = userInput();
    // ⚠️ source provides tainted value
    const structure = buildStructure(userInput); // ✅ sanitized
    processStructure(structure);
}
```

```
from CustomTaintTracking cfg, DataFlow::Node source, DataFlow::Node
sink
where cfg.hasFlow(source, sink)
select sink, "🚨 Sanitized flow from user input to sink!"
```

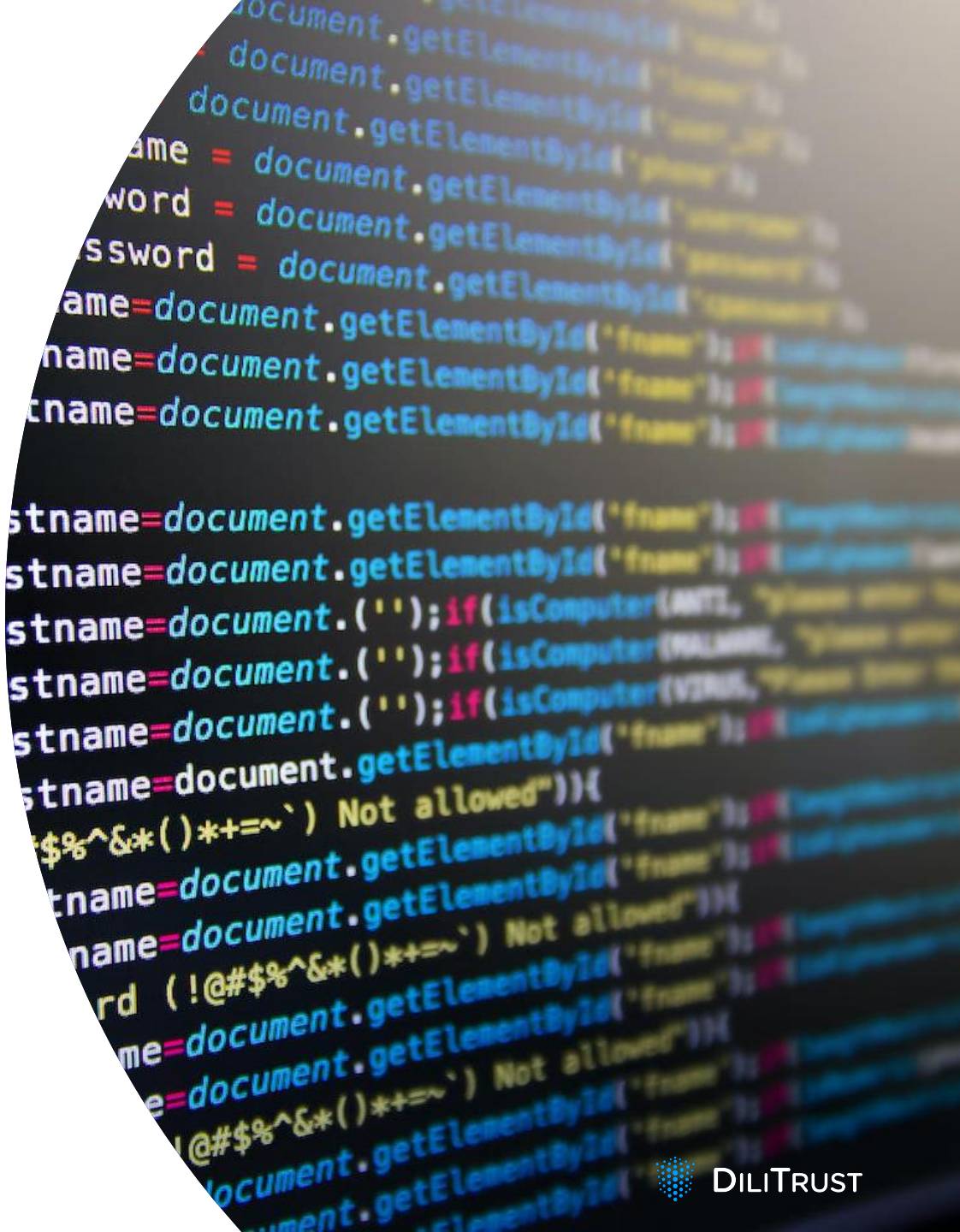
```
import javascript
import semmle.javascript.dataflow.TaintTracking

class CustomTaintTracking extends TaintTracking::Configuration {
  CustomTaintTracking() { this = "CustomTaintTracking" }

  override predicate isSource(DataFlow::Node source) {
    exists(CallExpr ce |
      ce.getCalleeName() = "userInput" and
      source.asExpr() = ce
    )
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(CallExpr ce |
      ce.getCalleeName() = "sink" and
      sink.asExpr() = ce.getArgument(0)
    )
  }

  override predicate isSanitizer(DataFlow::Node sanitizer) {
    exists(CallExpr ce |
      ce.getCalleeName() = "sanitize" and
      sanitizer.asExpr() = ce.getArgument(0)
    )
  }
}
```

# Limitations

## False Positives and False Negative

| Cause | False Positive? | False Negative? | Example |
|---|---|---|---|
| Static Analysis Limitations | ✅ Yes | ✅ Yes | Cannot see actual runtime behavior (e.g., dynamic function calls) |
| Overly General Query Rules | ✅ Yes | ❌ No | Flags too many cases as vulnerable (e.g., all SQL string concatenations) |
| Lack of Context Awareness | ✅ Yes | ✅ Yes | Doesn't always recognize sanitization functions (e.g., parseInt()) |
| Missed Data Flow Tracking | ❌ No | ✅ Yes | Fails to track tainted data through function calls or libraries |
| Too Strict Query Conditions | ❌ No | ✅ Yes | Only detects vulnerabilities when they exactly match predefined patterns |

# CodeQL Alternatives



# OpenSource SAST

**Semgrep** relies on syntactic pattern matching with only limited, experimental taint analysis features. Offers an open-source community edition (LGPL 2) and is backed by **Semgrep Inc** with commercial solution.

https://semgrep.dev/

**Bearer** proposes Intraprocedural data flow analysis. Only Bearer CLI is open source  (Elastic License 2.0). Backed by **Cycode** and offers with commercial solution.

https://github.com/Bearer

**Joern** is fully open-source solution (Apache 2) based on a Code Property Graph (CPG) and Scala DSL query language. Used by academic researcher and community driven.

https://github.com/joernio/joern

# LLM Alternatives - ChatGPT

### Security Issues Identified:

1. **SQL Injection (Critical)**
   - The following line constructs an SQL query by directly concatenating user input:

```
js                                                    Copy  Edit

let query = "SELECT * FROM users WHERE id = " + userId;
```

   - If an attacker sends `?id=1 OR 1=1`, the query becomes:

```
sql                                                   Copy  Edit

SELECT * FROM users WHERE id = 1 OR 1=1;
```

   - **Fix:** Use parameterized queries or prepared statements to prevent SQL injection:

```
js                                                    Copy  Edit

let query = "SELECT * FROM users WHERE id = ?";
db.query(query, [userId], (err, result) => { if (err) throw err; res.send(resu
```

# LLM Alternatives - ChatGPT

**Secure Code Implementation:**

```js
const express = require('express');
const mysql = require('mysql');
const helmet = require('helmet');
const rateLimit = require('express-rate-limit');
require('dotenv').config();

const app = express();
app.use(helmet());

const limiter = rateLimit({ windowMs: 15 * 60 * 1000, max: 100 });
app.use(limiter);

const db = mysql.createConnection({
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  password: process.env.DB_PASS,
  database: process.env.DB_NAME
});
```

```
app.get('/user', (req, res) => {
  let userId = parseInt(req.query.id, 10);
  if (isNaN(userId)) return res.status(400).send("Invalid user ID");

  let query = "SELECT * FROM users WHERE id = ?";
  db.query(query, [userId], (err, result) => {
    if (err) return res.status(500).send("Internal Server Error");
    res.send(result);
  });
});

app.listen(3000, () => console.log("Server running on port 3000"));
```

# LLM Alternatives

| | SAST | LLM | Hybrid Approach |
|---|---|---|---|
| OWASP Top 10 Coverage | 92% | 78% | 95% |
| Business Logic Flaws | 18% | 67% | 72% |
| False Positive Rate | 34% | 41% | 22% |
| Novel Vuln Discovery | 12% | 54% | 48% |

IRIS, a neuro-symbolic approach that systematically combines LLMs with static analysis to perform whole-repository reasoning for security vulnerability detection

Table 1: Overall performance comparison of CodeQL vs IRIS on Detection Rate (↑), Average FDR (↓), and Average F1 (↑). We present results of IRIS with different LLMs including OpenAI GPT-4 and GPT-3.5, Llama-3 (L3) 8B and 70B, and DeepSeekCoder (DSC) 7B.

| | Method | #Detected (/120) | Detection Rate (%) | Avg FDR (%) | Avg F1 Score |
|---|---|---|---|---|---|
| | CodeQL | 27 | 22.50 | 90.03 | 0.076 |
| IRIS + | GPT-4 | **55** (↑ 28) | **45.83** (↑ 23.33) | **84.82** (↓ 5.21) | **0.177** (↑ 0.101) |
| | GPT-3.5 | 47 (↑ 20) | 39.17 (↑ 16.67) | 90.42 (↑ 0.39) | 0.096 (↑ 0.020) |
| | L3 8B | 41 (↑ 14) | 34.17 (↑ 11.67) | 95.55 (↑ 5.52) | 0.058 (↓ 0.018) |
| | L3 70B | 54 (↑ 27) | 45.00 (↑ 22.50) | 90.96 (↑ 0.93) | 0.113 (↑ 0.037) |
| | DSC 7B | 52 (↑ 25) | 43.33 (↑ 20.83) | 95.40 (↑ 5.37) | 0.062 (↓ 0.014) |

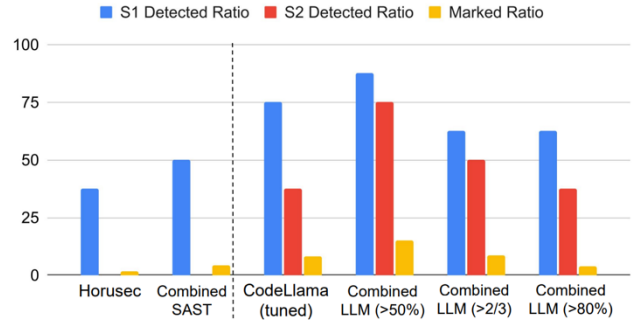


Figure 5: Combinations of SAST Tools or LLMs for Java

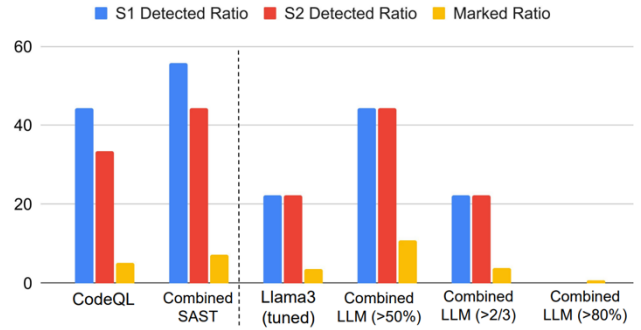
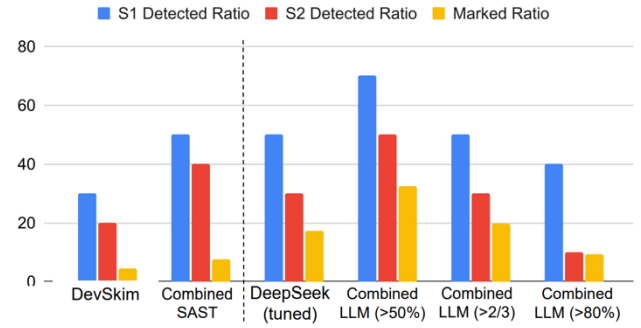


Figure 6: Combinations of SAST Tools or LLMs for C



igure 7: Combinations of SAST Tools or LLMs for Python

LLM-Assisted Static Analysis for Detecting Security Vulnerabilities: https://arxiv.org/pdf/2405.17238v2
Comparison of Static Application Security Testing Tools and Large Language Models for Repo-level Vulnerability Detection: https://arxiv.org/pdf/2407.16235



DILITRUST

# LLM Limitations

LLM Limitations

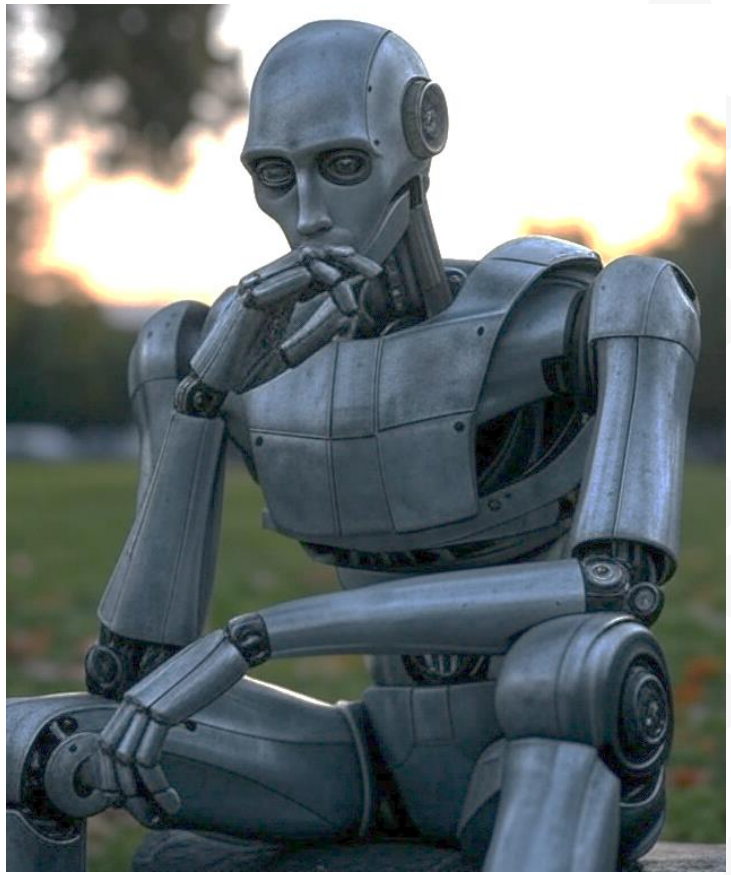
**Cost**: Analyzing extensive codebases demands significant computational power.

**False Positives and Negatives**: LLMs can generate many false alarms or miss clear vulnerabilities when analyzing code.

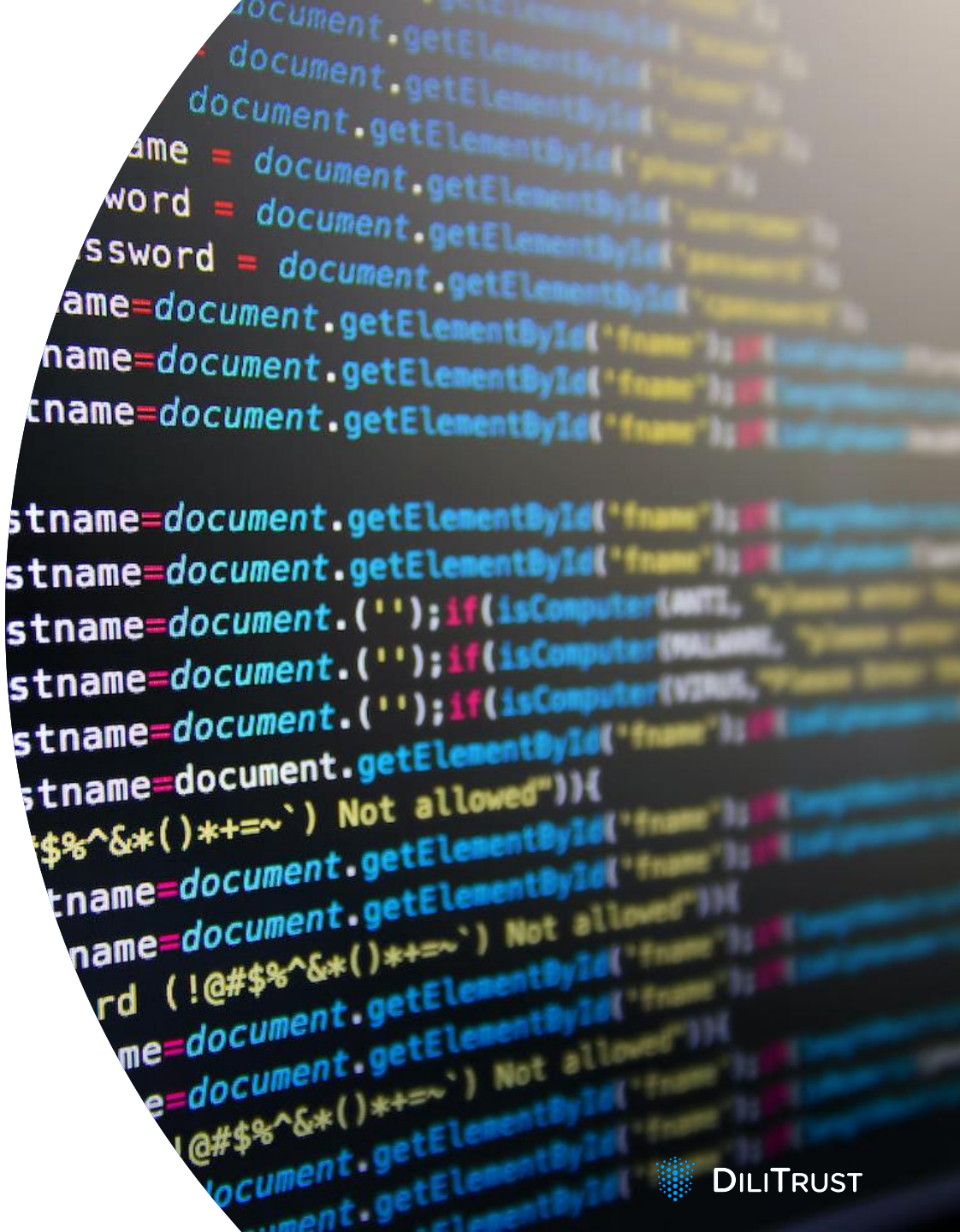
**Contextual Understanding**: Despite their ability to grasp semantic context, LLMs struggle with complex reasoning over entire codebases.

**Lack of Domain Expertise**: LLMs may not possess the same level of domain expertise as human developers, potentially leading to code that is not optimized for security, performance, or maintainability.

**Evolving Nature of Security Threats**: As security threats constantly evolve, LLMs need to be regularly updated.



# Conclusion




DiliTrust

# Conclusion

CodeQL is a **powerful tool**, backed by an active community of security experts.

Mastering CodeQL requires a **significant time** investment in learning query writing.

SAST generates numerous **false positives** and primarily detects **well-known vulnerabilities**.

While **LLMs show great promise**, they are neither cost-effective nor fast, particularly for large codebases and CI workflows.

Combining **SAST with LLMs** might be a middle ground but somehow feels like convoluted solution.

DiliTrust

# Thank you

dilitrust.com