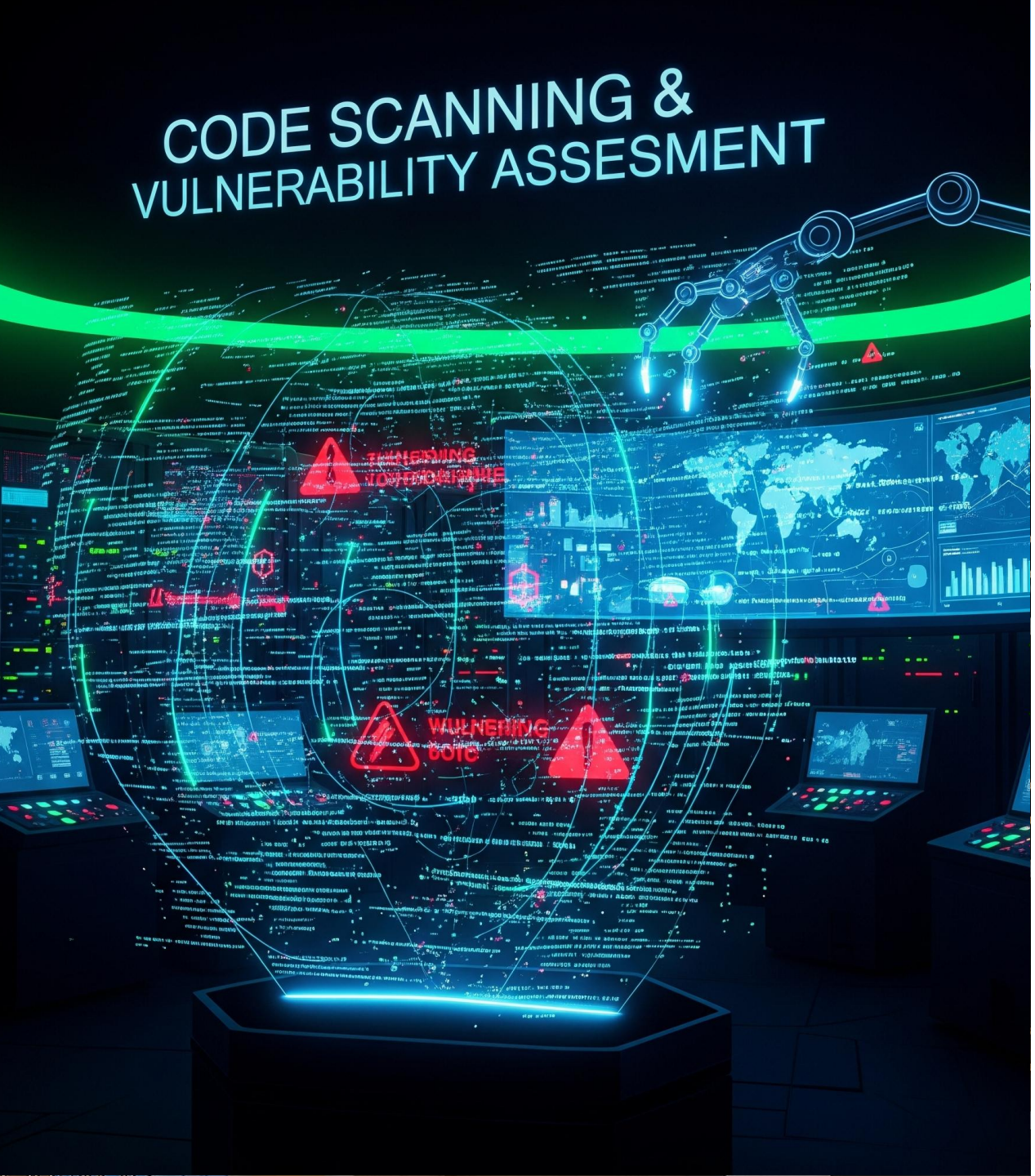


Fundamentals of Secure Software Engineering

Winter term 2025/26

Code Scanning

Dr. Sergej Dechand / Dr. Christian Tiefenau

Outline today

🐛 Vulnerability of the day

- Cache Poisoning

🔍 Code Scanning / Static Analysis (Continued)

- Code Property Graphs
- Taint Analysis

⚙️ Dynamic Analysis

- Black-box Testing
- Testing Phases

⚔️ Vulnerability (Attack) of the day

Cache Poisoning

VOTD | Cache Poisoning: Actually an attack technique

Common Attack Pattern Enumeration and Classification

CAPEC-141: Cache Poisoning

Attack Pattern ID: 141
Abstraction: Standard

Status: Draft
Completeness: Complete

Presentation Filter: Basic

Summary

An attacker exploits the functionality of cache technologies to cause specific data to be cached that aids the attackers' objectives. This describes any attack whereby an attacker places incorrect or harmful material in cache. The targeted cache can be an application's cache (e.g. a web browser cache) or a public cache (e.g. a DNS or ARP cache). Until the cache is refreshed, most applications or clients will treat the corrupted cache value as valid. This can lead to a wide range of exploits including redirecting web browsers towards sites that install malware and repeatedly incorrect calculations based on the incorrect value.

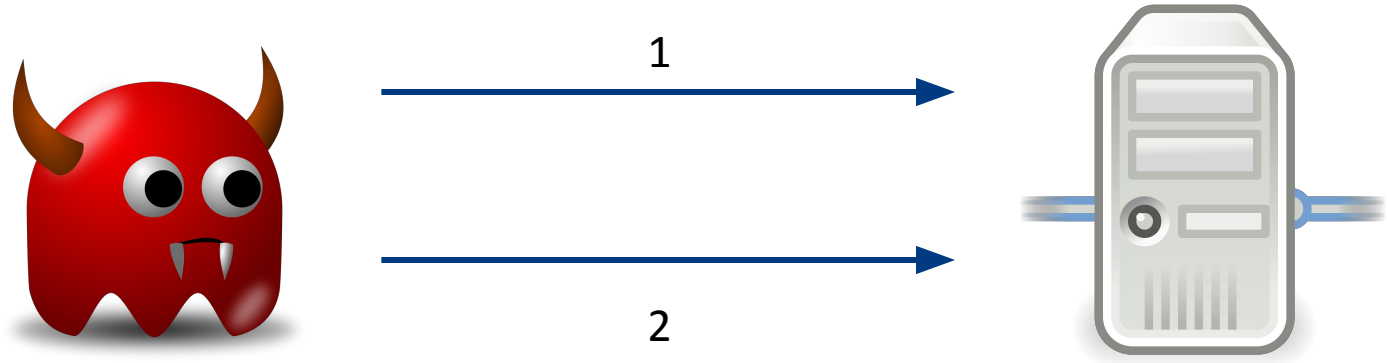
Attack Prerequisites

- The attacker must be able to modify the value stored in a cache to match a desired value.
- The targeted application must not be able to detect the illicit modification of the cache and must trust the cache value in its calculations.

http://capec.mitre.org/data/definitions/141.html

Example | BIND DNS Cache Poisoning Attack

The mechanism of a DNS Cache Poisoning attack



1. **Attacker continuously issues DNS queries to DNS server**
   - Server will delegate up, asking its parent for the record
2. **Attacker also sends forged, incorrect responses to server**
   - Problem: authenticity of response checked by a too small nonce (16 bit)
3. **Once the correct nonce is (coincidentally) sent to the server, server will cache bogus record**
4. **Henceforth queries for that domain will yield the bogus response**

State-of-the-art solution: DNSSec

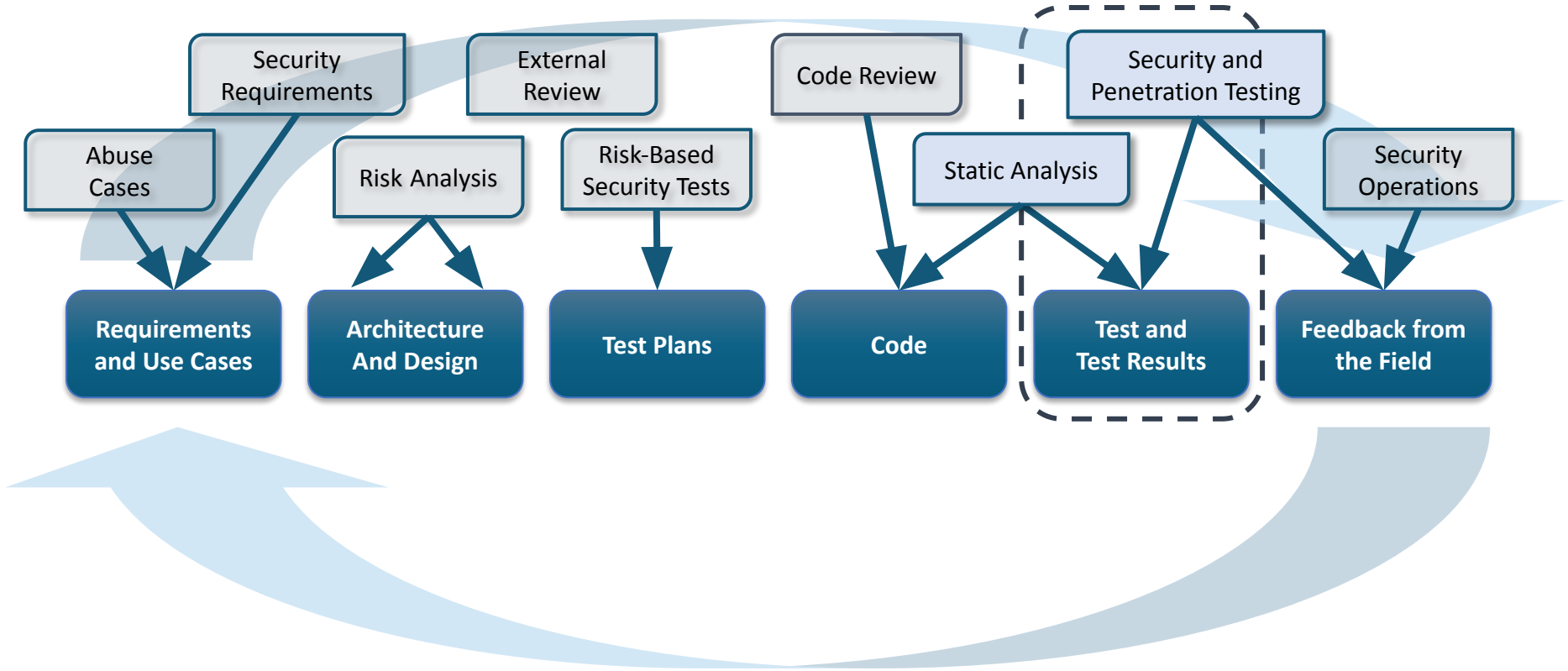
https://www.invicti.com/learn/mitm-dns-spoofing-dns-cache-poisoning/

Avoiding cache poisoning

- If possible, don’t allow users much control over caches to begin with
- As always, input validation helps, but it should be complemented with other countermeasures.

Today's Lecture

Shifting further to Security Testing



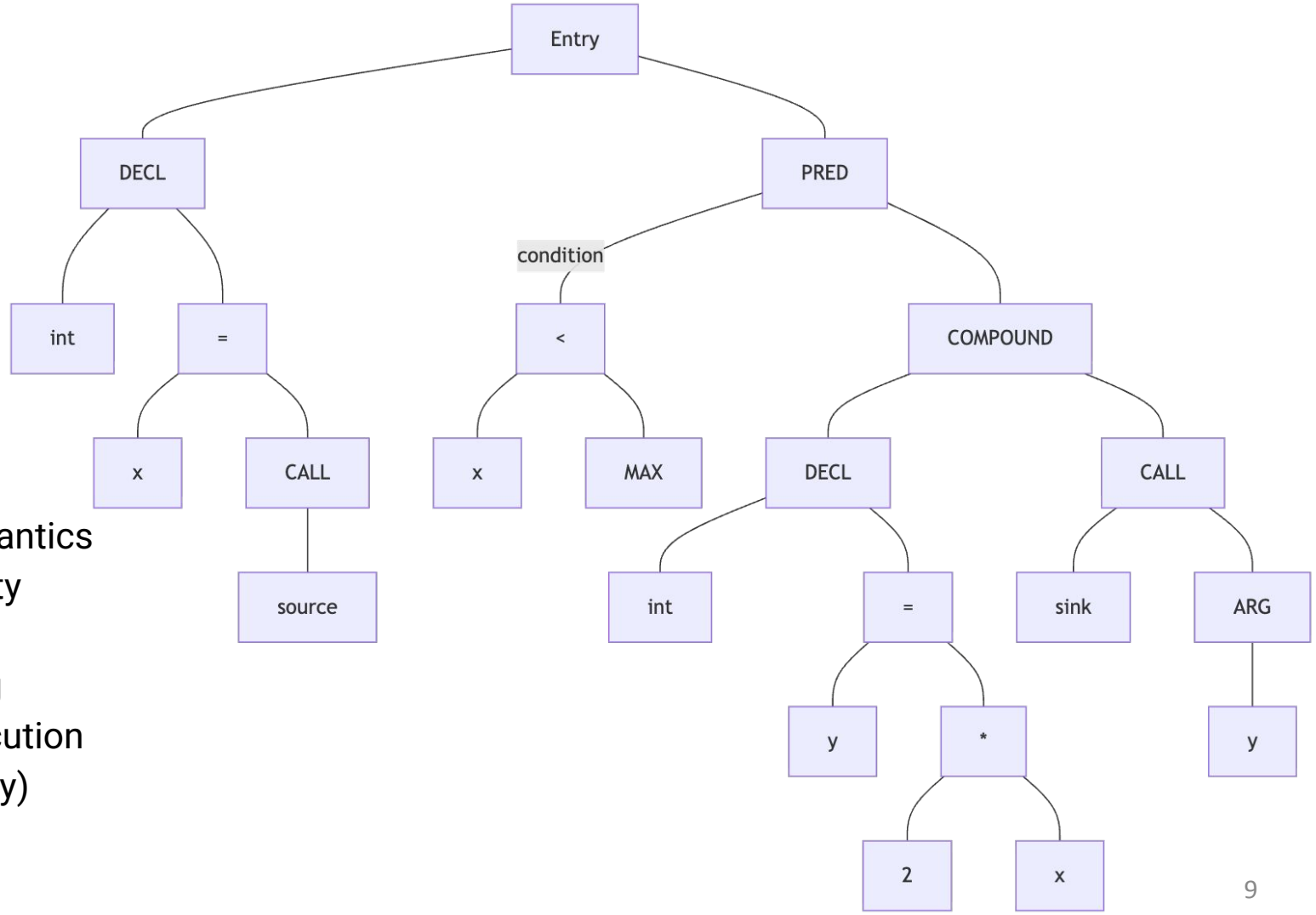
🔍 Static Analysis (Continued)

Static Analysis

Abstract Syntax Tree (ASTs)

The foundation for program analysis: capturing code structure for automated reasoning

```
void foo() {
  int x = source();
  if (x < MAX) {
    int y = 2 * x;
    sink(y);
  }
}
```

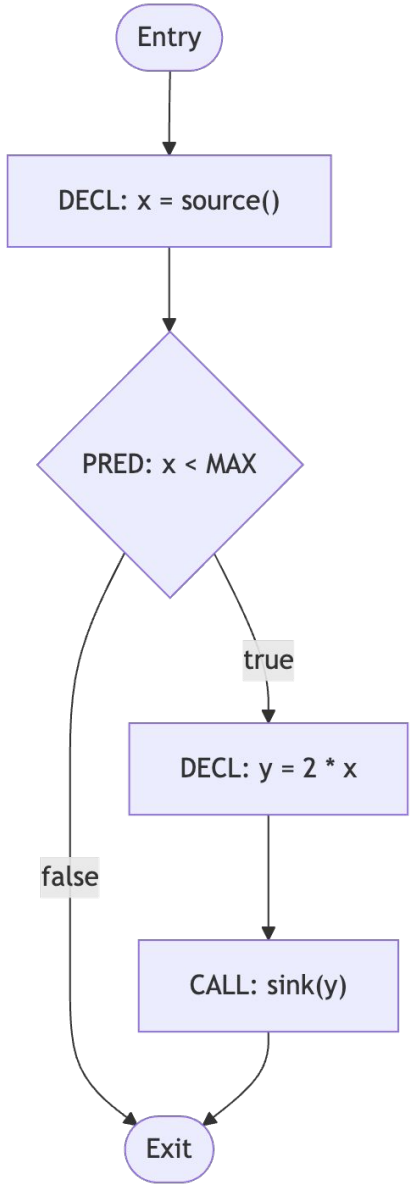


- Describes program structure, types, and semantics
- Enables pattern matching for bug/vulnerability detection
- Language-agnostic representation for tooling
- **Limitation:** AST shows structure but not execution order → we can't tell reachable paths to sink(y)

Control Flow Graphs (CFGs)

Beyond syntax: extracting and exposing execution paths for analysis

```
void foo() {
  int x = source();
  if (x < MAX) {
    int y = 2 * x;
    sink(y);
  }
}
```

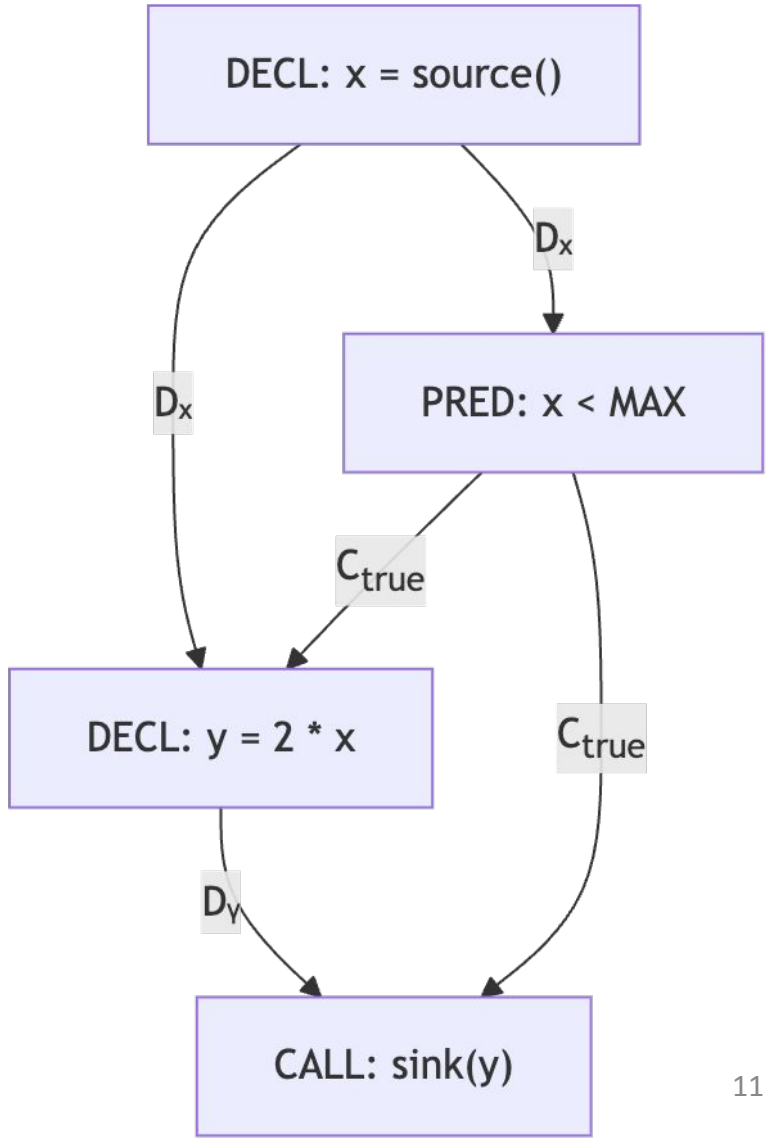


- AST misses which statements can follow which, which paths are reachable, and how control flows through branches and loops
- AST can be used to build a CFG
- Edges (arrows) show all possible execution sequences from entry to exit
  - Can sink() be reached?
  - Can both branches execute?
- True/false edges capture guards that determine which code executes
- **Limitation:** CFG doesn't provide data dependencies → we see sink(y) can execute, but not that y is derived from x which came from source()

Program Dependence Graphs (PDGs)

Tracking data and control dependencies: what influences what

```
void foo() {
  int x = source();
  if (x < MAX) {
    int y = 2 * x;
    sink(y);
  }
}
```



- CFG shows possible execution order, not data flow: Missing which variables affect which computations and which statements depend on conditions
- Data edges (D_x, D_y): Track value propagation: x flows from source() to condition and y computation
- Control edges (C_true): Statements executed only when condition holds - y=2*x and sink(y) depend on x < MAX being true
- Follow data edges from untrusted sources to dangerous sinks
- Program slicing: Answer "what affects this sink?" (backward) or "what does this source affect?" (forward)

Code Property Graphs (CPGs)

From separate graphs to unified analysis: tools for practical vulnerability discovery

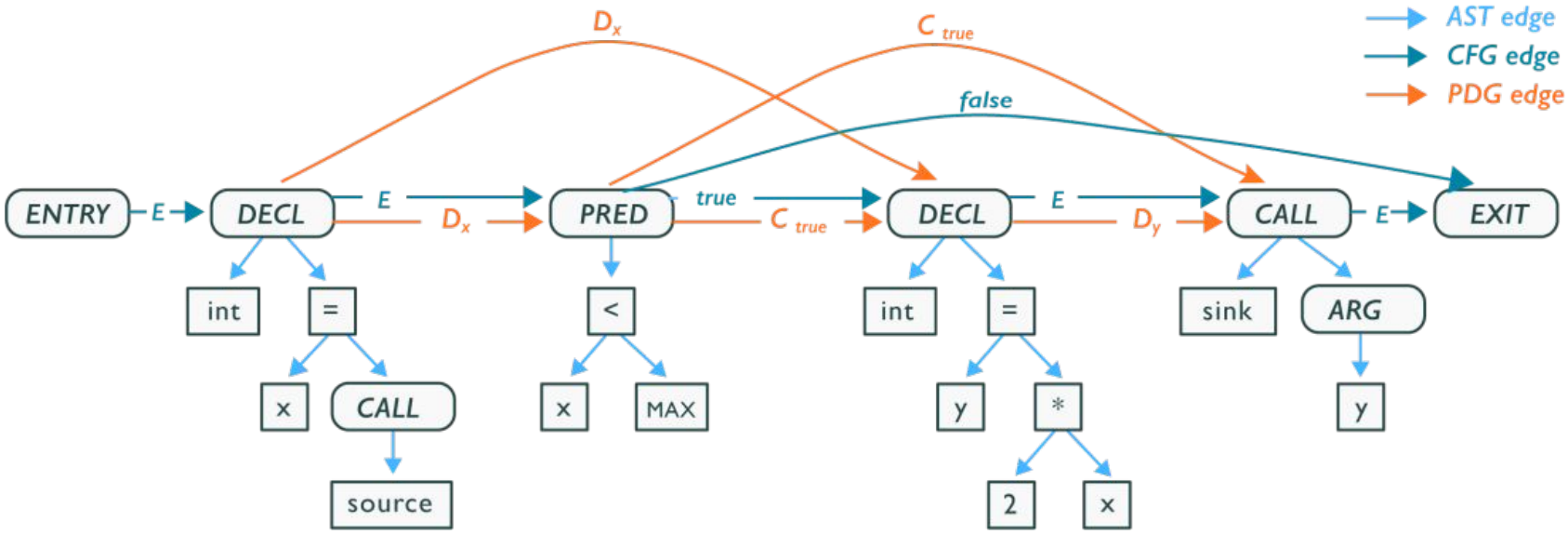
```
void foo() {
  int x = source();
  if (x < MAX) {
    int y = 2 * x;
    sink(y);
  }
}
```

Unified representation: All three graphs combined into single queryable structure

Goal: Query CPG to find source to sink flows traversing data edges along control paths containing taints

Different tools: Joern (CPG queries), Semgrep (pattern + dataflow), CodeQL (QL queries)

Production impact: These approaches discovered thousands of real vulnerabilities in open-source projects



Example Queries (Simplified)

Use markdown or query languages to find tainted data

Semgrep

```
rules:
  - id: sql-injection-concat
    languages: [java]
    severity: ERROR
    message: "SQLi via string concatenation"
    pattern: $STMT.executeQuery($Q + ...)
    pattern-not: $STMT.executeQuery("...")

  - id: sql-injection-taint
    mode: taint
    languages: [java]
    message: "Tainted SQL query"
    pattern-sources:
      - pattern: (HttpRequest
$R).getParameter(...)
    pattern-sinks:
      - pattern: (Stmt $S).executeQuery(...)
    pattern-sanitizers:
      - pattern: Encoder.encodeForSQL(...)
```

Joern

```
def source = cpg.call
  .methodFullName("HttpRequest.getParameter.*")

def sink = cpg.call
  .methodFullName("Statement.executeQuery.*")
  .argument(1)

def sanitizer = cpg.call.name("encodeForSQL")

sink.reachableBy(source)
    .whereNot(_.reachableBy(sanitizer))
    .l
```

Open Challenges & Current Research

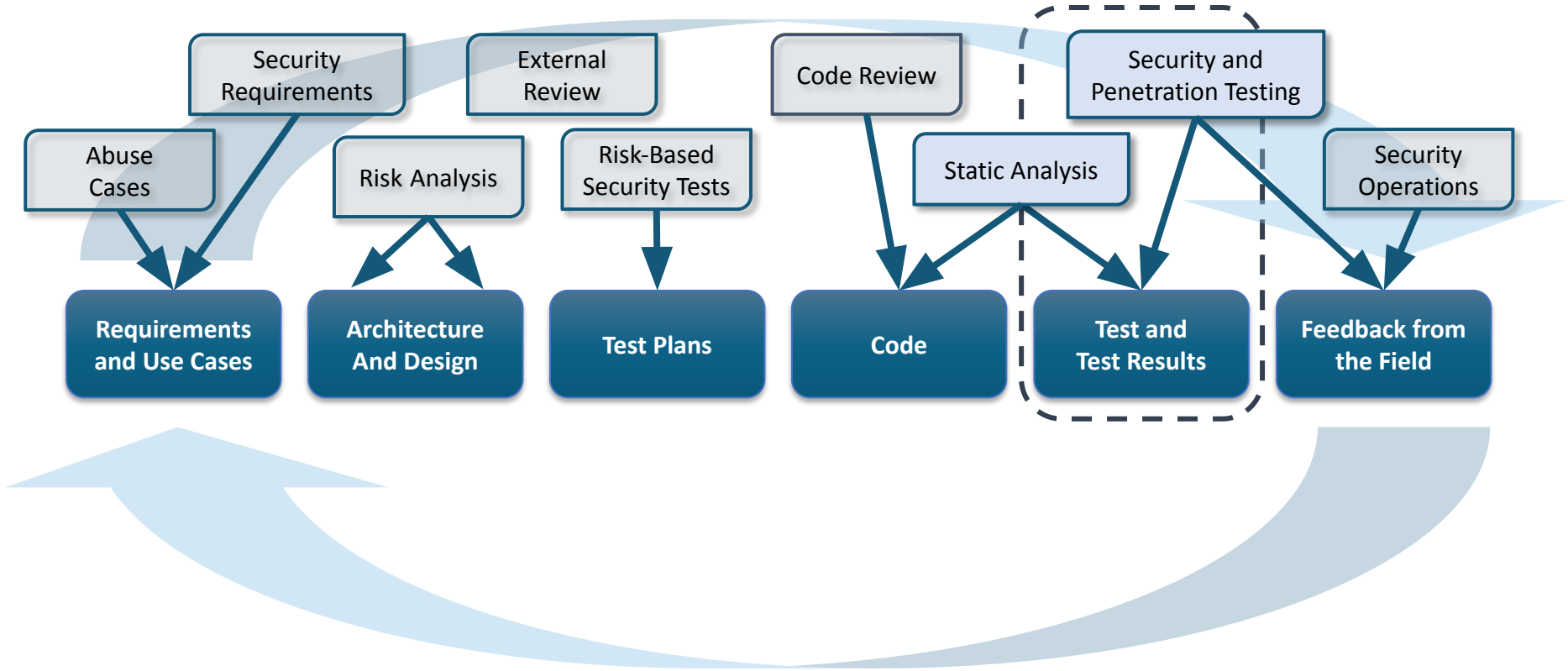
```
interface DataStore {
    void save(String data);
}

class SecureDB implements DataStore {
    void save(String data) { }
}

class InsecureDB implements DataStore {
    void save(String data) { /* string conc → SQLi */
}
}

void process(DataStore store, String userInput) {
    store.save(userInput);  // Which implementation?
}
```

Why this is hard

- Precise solution requires whole-program analysis → doesn't scale
- Frameworks use reflection, dependency injection, dynamic proxies → invisible to static analysis
- The more abstract the code, the worse static analysis performs

Today's Lecture

Shifting further to Security Testing



Dynamic Analysis

Dynamic Application Security Testing (DAST)

Dynamic Application Security Testing (DAST)

Attack a running system

📘 Definition

Dynamic Application Security Testing (DAST) is typically a black-box security testing method that attacks a running system ‘from outside’ and observes its behavior, just like a real attacker would.

✅ How it works (simplified)

- Attack interfaces and endpoints
  - Fuzz inputs by mutating benign test data and adding randomness
  - Apply attack heuristics
- Watch for unexpected behavior and crashes

🎯 Use Cases

- Later in the SDLC
- You need a running system (at least the modules under test)

🛠️ Tools

- OWASP ZAP (Open Source), Burp Suite (Pentesting Standard), Restler, Jazzer [1]

1. 🐝Dynamic Vulnerability Scanning Tools

(Blackbox) Dynamic Analysis / Fuzzing

Consisting of two components: input generation (Fuzzing) und bug detection

fuzz | verb.
/'fəz/
1. to make or become blurred

⌨️ Input Generation / Fuzzing

- Scripted test cases
- Fuzzing
  - Mutated inputs
  - Payload variations
  - Boundary value cases
- Authentication bypass attempts

Mutated Inputs →

← Feedback

🔎 Observe System Under Test

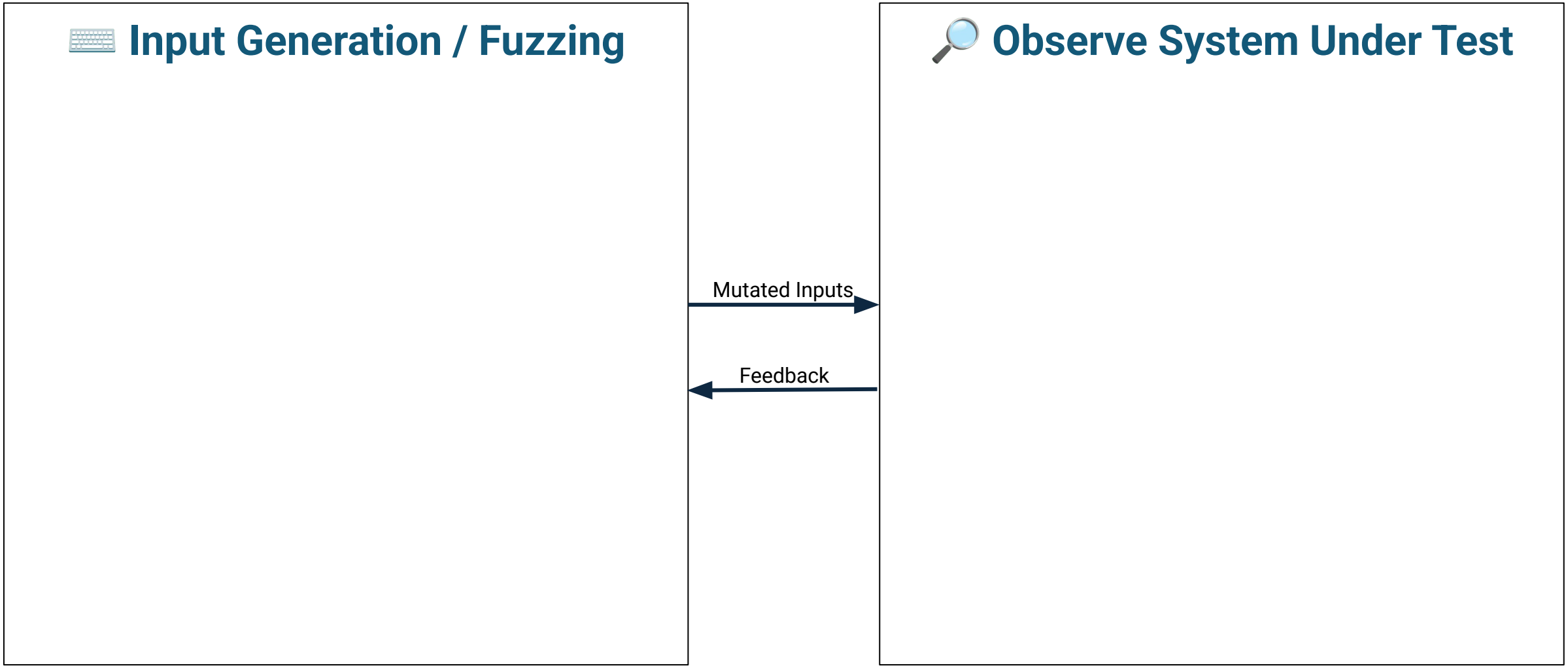
- Observe
  - Crashes
  - Timeouts
  - Error messages
  - Data flow anomalies
- Bug detectors

(Blackbox) Dynamic Analysis / Fuzzing

Consisting of two components: input generation (Fuzzing) und bug detection

fuzz | verb.
/'fəz/

1. to make or become blurred



(Blackbox) Dynamic Analysis / Fuzzing

Consisting of two components: input generation (Fuzzing) und bug detection

fuzz | verb.
/'fəz/
1. to make or become blurred

⌨️ Input Generation / Fuzzing

```
> GET /user/admin HTTP/1.1
```

Mutated Inputs →

← Feedback

🔎 Observe System Under Test

```
< HTTP/1.1 200
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Sun, 06 Apr 2025 11:04:20 GMT
<
< {"success":true,"users":["admin"]}
```

(Blackbox) Dynamic Analysis / Fuzzing

Consisting of two components: input generation (Fuzzing) und bug detection

fuzz | verb.
/'fəz/
1. to make or become blurred

⌨️ Input Generation / Fuzzing

```
> GET /user/123 HTTP/1.1
```

Mutated Inputs →

← Feedback

🔎 Observe System Under Test

```
< HTTP/1.1 200
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Sun, 06 Apr 2025 11:33:07 GMT
<
< {"success":true,"users":[]}
```

(Blackbox) Dynamic Analysis / Fuzzing

Consisting of two components: input generation (Fuzzing) und bug detection

fuzz | verb.
/'fəz/
1. to make or become blurred

⌨️ Input Generation / Fuzzing

Mutated Inputs →

← Feedback

🔎 Observe System Under Test

```
< HTTP/1.1 404
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Sun, 06 Apr 2025 12:06:52 GMT
<
<
{"timestamp":"2025-04-06T12:06:52.456",
"status":404,"error":"Not Found",
"path":"/123"
```

(Blackbox) Dynamic Analysis / Fuzzing

Consisting of two components: input generation (Fuzzing) und bug detection

fuzz | verb.
/'fəz/
1. to make or become blurred

⌨️ Input Generation / Fuzzing

```
> GET /user/Bzp%27%3B%20DROP%20TABLE%20users%3B%20--%20
```

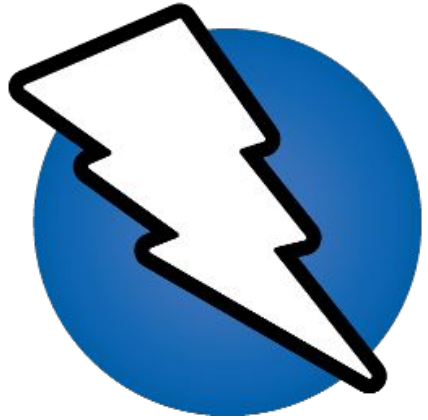
Mutated Inputs

Feedback

🔎 Observe System Under Test

DAST | In a Nutshell

Making requests and observing responses with OWASP ZAP



🎯 Target URL

`http://localhost:8080/user/admin`

💉 ZAP Payloads leading to SQL Injection

Request	Result	Comment
/user/admin	{"users":["admin"]}	Normal request
/user/admin1	{"users":[]}	Invalid user, handled cleanly
⋮	{"users":[]}	Lots of benign errors
/user/adm%'	"Syntax error in SQL statement "	Likely an SQL injection
/user/adm%';	"Syntax error in SQL statement "	Likely an SQL injection
/user/adm%';D	"Syntax error in SQL statement "	Likely an SQL injection
⋮	Errors	Multiple suspicious errors
/user/adm%';DROP TABLE users	"Syntax error in SQL statement	SQLi payload triggers DB error
/user/admin (again)	Table "USERS" not found	Confirms successful injection

DAST | Usage

Finds inputs that trigger the vulnerability (with almost no false positives)

📝 Note

In our example, we could not only find the vulnerability but also the concrete input (easier to exploit). Depending on logging settings, we lose the code location where the error occurs.

📋 Requirements

- Requires a running system
- Can only use documented API definitions
- Mainly used by penetration testers and attackers

🎯 Pentester's Favorite[1]

- Without source code and thus language-agnostic
- Significantly lower number of false positives with concrete inputs
- Checks the actual configuration in the production system
- Get the potential user input triggering the issue

[1] Most reported vulnerabilities are found with DAST (despite difficult attribution). This is not scientifically proven but is generally accepted by the security community.

DAST | Limitations

SAST can quickly find errors that might take a long time with DAST

📝 Note

Since DAST typically works without source code, assumptions must be made based on observations. In complex applications, good test cases and attack strategies are needed to quickly find vulnerabilities. Strategies are also needed to know when to stop.

```
public ResultSet LoginUser(String username, String password) throws SQLException {
    if (username.equals("Larry")) {
        // perform check confusing dynamic analysis
        // still vulnerable to SQL Injection
        String query = "SELECT * FROM users WHERE username = '"
            + username + "' AND password = '" + password + "'";
        return this.conn.createStatement().executeQuery(query);
    } else {
        return null;
    }
}
```

DAST | Summary

Despite its immense advantages, DAST cannot replace SAST

✅ Advantages

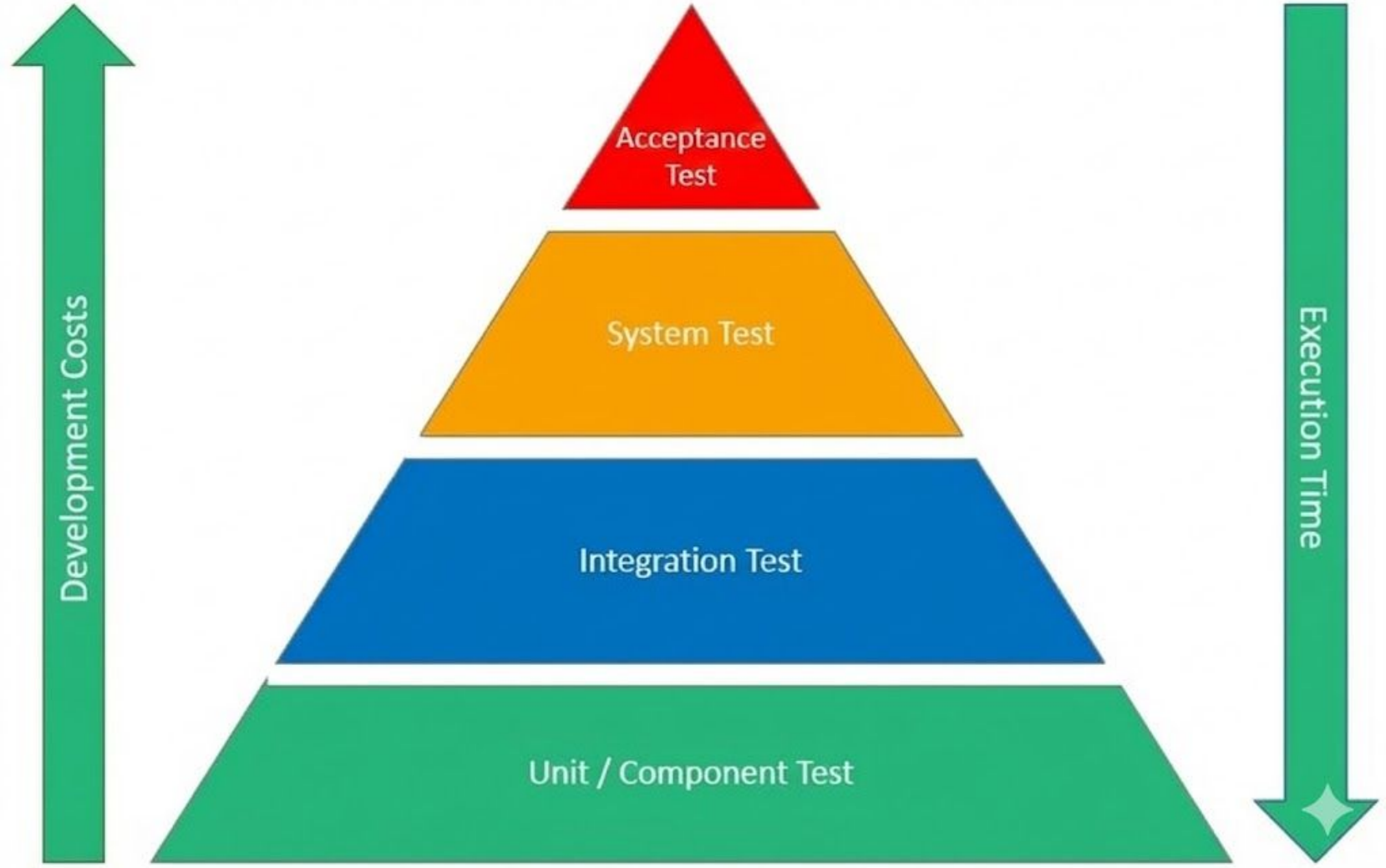
- No source code required / language-agnostic
- Finds vulnerabilities that SAST overlooks
- Provides the input that triggers the vulnerability

❌ Disadvantages

- Simple source code constructs can confuse DAST
- Difficulties with complex applications
- Requires good API documentation
- There is no defined end (when to stop testing)

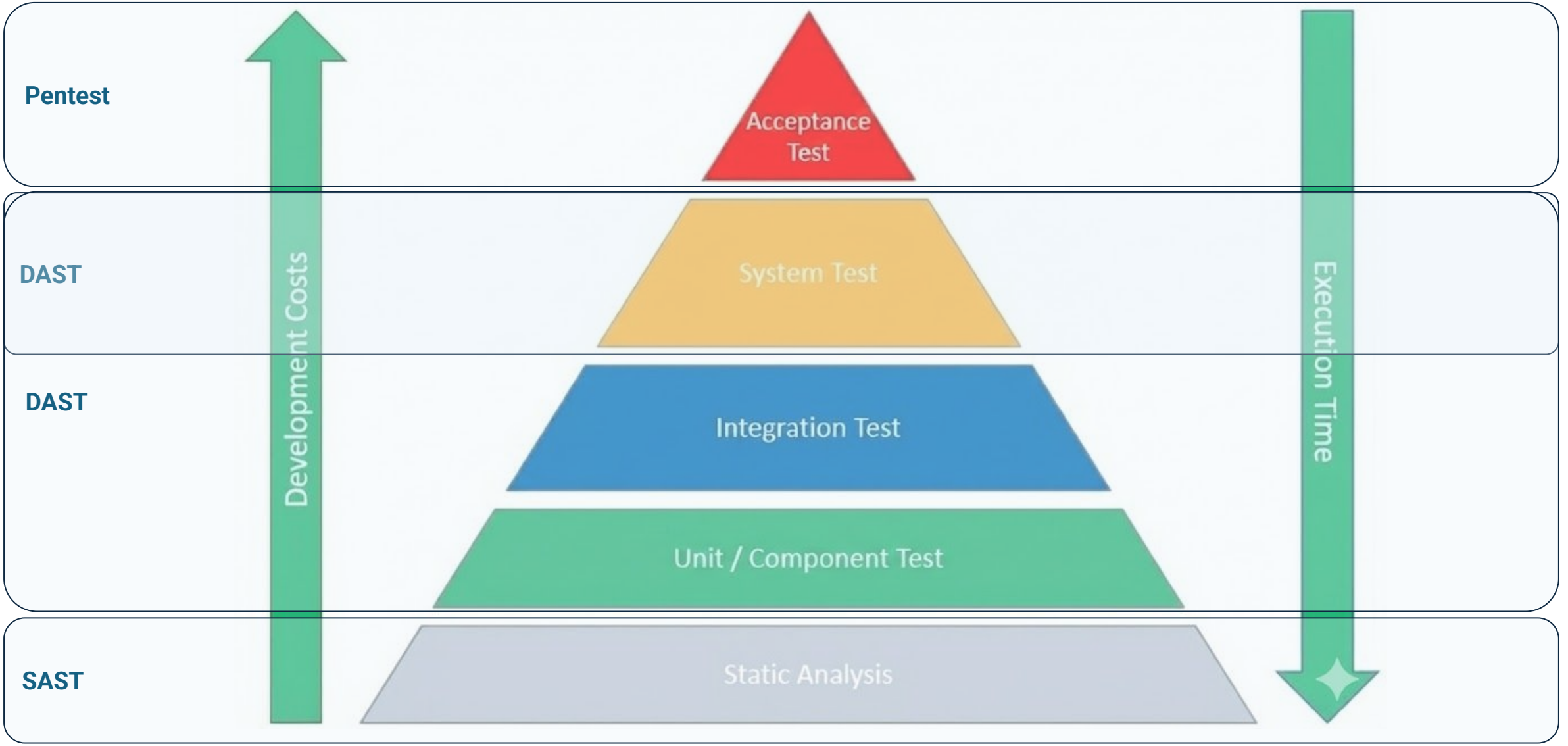
Recap | Remember the Test Pyramid?

Same concepts apply for security testing



Recap | Remember the Test Pyramid?

Same concepts apply for security testing



References & Further Readings

- Yamaguchi et al: *Modeling and Discovering Vulnerabilities with Code Property Graphs*, IEEE S&P, 2014
- Brian Chess and Jacob West, *Secure Programming with Static Analysis*, Addison-Wesley, 2007.
- Michael Howard, *A Process for Performing Security Code Reviews*.IEEE S&P, July 2006.
- Eoin Keary et. al., OWASP Code Review Guide 1.1, http://www.owasp.org/index.php/Category:OWASP_Code_Review_Project, 2008.
- Jason Cohen, Best Kept Secrets of Peer Code Review http://smartbear.com/SmartBear/media/pdfs/best-kept-secrets-of-peer-code-review.pdf, 2006.
- IEEE Standard for Software Reviews and Audits," in IEEE Std 1028-2008 , Aug. 2008. DOI: 10.1109/IEEESTD.2008.4601584
- Khedker, Uday et al., Data flow analysis: theory and practice, 2009

References & Further Readings

- Soot: www.soot-oss.org
- Phasar: www.phasar.org
- https://www.businessinsider.com/why-hacker-gang-lizard-squad-took-down-xbox-live-and-playstation-network-2014-12?r=US&IR=T