Of course. Understanding the difference between static and dynamic analysis is fundamental to automated vulnerability discovery. They are two distinct philosophies for examining code, each with its own strengths and weaknesses.

Here is a detailed breakdown with examples.

The Fundamental Difference

- **Static Analysis** looks at the application's code and structure *without executing it*. It's like reading a car's blueprint to find design flaws.
- **Dynamic Analysis** tests the application *while it is running*. It's like test-driving the car to see if it crashes or behaves unexpectedly under stress.

Static Analysis (SAST - Static Application Security Testing)

What It Is:
SAST tools analyze an application's source code, bytecode, or compiled binaries in a non-running state. They model the application to find coding patterns and data flows that could lead to security vulnerabilities.
How It Works:

1. **Code Modeling:** The tool first parses the code to build a model, often an Abstract Syntax Tree (AST) or a Code Property Graph (CPG). This model represents the code's structure, components, and the relationships between them.
2. **Pattern Matching:** It can scan for simple, known-bad patterns, like the use of a dangerous function (e.g., eval() in PHP or JavaScript).
3. **Taint Analysis:** More advanced tools perform static taint analysis. They identify "sources" (user-controlled input like $_GET['id']) and "sinks" (dangerous functions like mysqli_query()). They then analyze the code to see if there is any possible path where data from a source can reach a sink without being properly sanitized.

Pros	Cons
Complete Code Coverage: In theory, it can analyze 100% of the codebase, including obscure error-handling routines and corners of the application that are difficult to reach during runtime testing.	**High Rate of False Positives:** This is the biggest weakness. Because it doesn't execute the code, it can't know the application's state or context. It may flag a "vulnerable" path that is impossible to trigger in a real-world scenario, leading to a lot of noise.
Finds Flaws Early: It can be integrated	**Struggles with Runtime Complexity:** It

directly into a developer's workflow and CI/CD pipelines, finding potential bugs before the application is even deployed.	has a very hard time with issues that only appear at runtime, such as misconfigurations on the server, vulnerabilities in third-party libraries, or complex logic in dynamically-typed languages like PHP and Python.
Pinpoints Exact Location: Since it works with the source code, it can usually report the exact file and line number of a potential vulnerability, making it easier for developers to fix.	**Can't Understand Business Logic:** It doesn't understand the developer's intent and may flag code that is technically risky but perfectly safe within the context of the application's business logic.

Example:
CodeQL (from GitHub/Microsoft) is a leading SAST tool. It treats an application's source code as a database. Security researchers write special queries to find vulnerable patterns. For instance, a query could be: "Find all paths where data from an HTTP request can reach a SQL query execution function without first passing through a sanitization function."

Dynamic Analysis (DAST - Dynamic Application Security Testing)

What It Is:
DAST tools interact with a running application to find vulnerabilities. They treat the application like a black box (or grey box if instrumented) and probe it for weaknesses from the outside.
How It Works:

1. **Discovery/Crawling:** The tool first crawls the running application to discover all accessible pages, API endpoints, and input parameters.
2. **Attack/Fuzzing:** It then sends a barrage of malicious or unexpected payloads to these inputs. This can include SQL injection strings, cross-site scripting payloads, or, in the case of SSRFuzz, specially crafted URLs.
3. **Observation:** The tool monitors the application's response to these attacks. It looks for evidence of a vulnerability, such as:
    - Application crashes or error messages.
    - Unexpected data being returned.
    - Callbacks to an external server (Out-of-Band or OAST, which is what detector.py does).
    - Delays in response time (for time-based attacks).

Pros	**Cons**

Low Rate of False Positives: If a DAST tool finds a vulnerability, it's almost certainly real because it was actually exploited and observed in a running environment.	**Limited Code Coverage:** It can only test the parts of the application that it can discover and access through crawling. If a feature is hidden behind a complex workflow or is not linked anywhere, the tool will likely miss it completely.
Finds Runtime & Configuration Issues: It is excellent at finding vulnerabilities that only exist because of the application's environment (e.g., server misconfigurations, issues in third-party components, flawed business logic).	**Requires a Running Application:** You must have a fully deployed and functional application environment to test, which means testing happens later in the development lifecycle.
Language Agnostic: Black-box DAST tools don't care if the application is written in PHP, Java, or Python. They only interact with it over standard protocols like HTTP.	**Difficult to Pinpoint Root Cause:** For a black-box tool, when a vulnerability is found (e.g., a SQL injection on a login form), it can be difficult to identify the exact file and line number in the source code that contains the flaw without manual investigation.

Example:
Burp Suite Professional's Scanner is a classic DAST tool. You point it at a web application, and it will crawl it and automatically send thousands of pre-built attack payloads to every input it finds, reporting any successful exploits. SSRFuzz from the paper is an advanced grey-box DAST tool because it combines external fuzzing with internal instrumentation (Taint Analysis) to get the best of both worlds.

Summary Table

Aspect	**Static Analysis (SAST)**	**Dynamic Analysis (DAST)**
Code Execution	Code is **not** executed (at rest).	Code **is** executed (in motion).
Primary Goal	Find potential flaws in the code itself.	Find exploitable vulnerabilities in the running application.

Code Coverage	High (can cover 100% of the codebase).	Lower (only covers what is discovered/executed).
False Positives	Can be very high.	Very low.
DevOps Stage	Early (CI/CD, pre-deployment).	Late (QA, post-deployment).
Root Cause	Easy to pinpoint (file and line number).	Can be difficult to pinpoint without instrumentation.

Modern approaches, like the one in the SSRFuzz paper, often create a **hybrid** system to get the benefits of both. SSRFuzz uses dynamic analysis (instrumentation and fuzzing), but it also performs a static-like analysis upfront to identify all the potential sink functions in the PHP language.