AST vs. CPG

The Context Divide: Why Traditional AST Falls Short

From Structure to Semantics

Contemporary application security testing has traditionally depended upon the **Abstract Syntax Tree (AST)** to examine code through parsing its structure. Although highly effective in discovering known vulnerabilities via pattern matching, AST-based tools are less proficient at comprehending how code behaves within the context of actual execution scenarios. With attackers focusing more on sophisticated logic, indirect data flows, and contextual triggers, the drawbacks of AST become ever more apparent.

Code Property Graphs (CPGs) enable a higher-level, more semantically dense method that integrates AST with control flow and data flow analysis. CPGs transcend surface syntax to richly comprehend a workload's behavior, intent, and logic. This enables tools like Qwiet AI to identify subtle, elusive vulnerabilities by inspecting what code appears and how it works in context.

This side-by-side comparison offers a clear, feature-to-feature contrast between AST-based tools like Checkmarx and Qwiet AI's CPG-based analysis. Whether you're a security engineer researching tooling or a decision-maker looking for a more forward-thinking application and code security solution.

A Side-by-Side Understanding

Feature	Checkmarx (AST-based)	Qwiet (CPG-based)
Structure Focused	✅ This demonstrates the original code structure.	✅ Encompasses AST but analyzes more than just the written code.
Control Flow Insight	❌No	✅Includes Control Flow Graph (CFG)
Data Flow / Dependencies	❌No	✅Includes Program Dependence Graph (PDG)
Contextual Understanding	⚠ Limited	✅ Deep, cross-functional, path-sensitive
Vulnerability Reachability	❌Pattern-only	✅ Path + flow-aware source-to-sink tracing
Analysis Method	Rule-based pattern matching	Graph traversal and semantic queries
AI/ML Potential	Narrow due to lack of context	High Graphs encode rich behavioral data

What AST Can and Can't Do

AST outlines the structure of how developers wrote the code, rather than how it behaves or reacts to changes. It can tell you:

- A function that was declared
- When it calls another function
- When variables were assigned

While AST is adept at syntax-aware **pattern matching**, its inability to comprehend the code's behavior is a significant limitation. I**t's like having a map of streets without cars, traffic lights, or directions - just street names**. It's the infrastructure without any of the utilities. This limitation underscores the need for a more advanced solution like the CPG.

Checkmarx uses this to match known destructive code patterns, which works well for well-understood, signature-style issues. But it **doesn't detect new**, unknown, or context-sensitive vulnerabilities.

What the CPG Adds: AST + CFG + PDG

Control Flow Graph (CFG)

- Shows execution paths (if/else, loops, calls)
- Understands the sequence in which code executes
- Detects unreachable code, execution dependencies

Program Dependence Graph (PDG)

- Shows how data moves between variables/functions
- Captures both control dependencies (e.g., "this happens if X") and data dependencies (e.g., "X depends on Y")
- Great for identifying side effects and vulnerable data propagation

Think of CPG as a GPS that not only shows the roads (AST), but also traffic (CFG), and what cars are carrying (PDG). This layered graph model **knows every piece of code's context, path, and purpose**.

Why CPG Enables True Semantic Analysis

With a CPG, Qwiet can perform **deep semantic queries**, such as:

- “Find all paths where user input reaches a file write function without sanitization.”
- “Show me where a tainted value reaches a system API.”
- “Track data from this variable across function calls and returns.”

This context is **not possible with just AST + pattern matching**.

Reachability Analysis & Source-to-Sink Flow

Qwiet’s CPG behaves like a **social network of your code**, where:

- Nodes = code entities (functions, parameters, calls)
- Edges = relationships (calls, uses, assigns, returns)

So Qwiet can trace how **tainted inputs (sources)** move through your code to **dangerous functions (sinks)** - this is **source-to-sink flow analysis**, essential for:

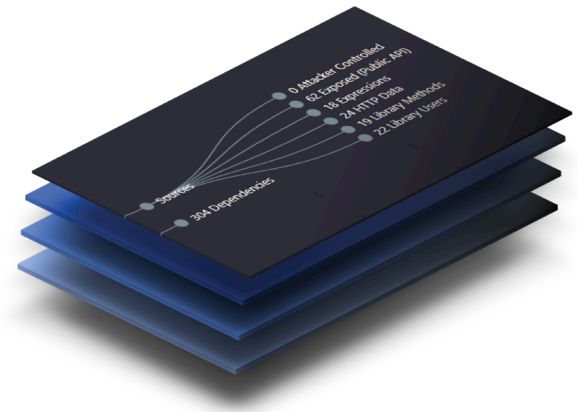
- SQL injection
- XSS
- Command injection
- Path traversal
- Business logic flaws

Qwiet's CPG can detect various vulnerabilities, including SQL injection, XSS, command injection, path traversal, and business logic flaws. It does this **without relying solely on known patterns**, meaning it can catch **zero-days and logic errors** that AST-based tools miss. This comprehensive coverage makes CPG a powerful tool for identifying and mitigating security risks in software applications.

Why It’s More Unique

Qwiet’s CPG is more technically sound and accurate because it:

- Models **real** program behavior, not just syntax.
- Enables graph-based AI/ML to **learn** how vulnerabilities appear in practice.
- **Modular and language-agnostic**, providing support for multiple languages by plugging in new parsers.



- AutoFix vulnerabilities can **be in context** because it understands the full behavior chain.

Conclusion

AST tools like Checkmarx tell you what the code looks like. *CPGs tell you what it does.* This fundamental difference makes Qwiet's CPG more powerful, context-aware, and future-proof, especially for catching advanced or unknown security issues. It's a tool that can keep up with the ever-evolving landscape of software security, and we are confident in its superiority.

Review our Solution Brief: Compliance and Security Benefits for Enterprises to learn more.

Ready to secure your codebase with Agentic AI?

Request a personalized demo to see how Qwiet AI delivers faster fixes, fewer false positives, and more intelligent security workflows powered by Code Property Graphs and AI AutoFix.

Request a Demo

qwiet AI