Code Property Graph Meets Typestate: A Scalable Framework to Behavioral Bug Detection

Xingjing Deng[1,*], Zhengyao Liu[1,*], Xitong Zhong[1], Shuo Hong[1], Yixin Yang[1]
Xiang Gao[1,#], Xuhui Yan[2], Hailong Sun[1,#]
[1] *Beihang University, Hangzhou Innovation Institute of Beihang University*
{xjdeng, zhengyaoliu, xitongzhong, hongshuo, yixinyang, xiang_gao, sunhl}@buaa.edu.cn
[2] *Huawei Company,* yanxuhui@huawei.com

***Abstract*—Behavioral bugs caused by incorrect state changes are particularly challenging to identify because they depend on specific code execution paths. While code property graph (CPG) combine multiple code views through abstract syntax trees (AST), their built-in redundancy from syntax details and fixed connection rules make them hard to scale—a major problem when analyzing large software systems. We introduce QVoG, a new framework that improves CPG by combining graph-based code analysis with state behavior checking. Our main innovation lies in simplifying the CPG at the statement level by consolidating control and data flows into meaningful code blocks and optimizing the edges. This approach reduces the graph size by more than 10 times compared to AST-based methods while maintaining accuracy. This lightweight design allows easy integration of state tracking, where we match object lifecycle rules to simplified CPG connections using replaceable patterns. The combination of streamlined graphs and state-aware analysis helps QVoG effectively find difficult-to-identify behavioral bugs, successfully detecting 25 issues (including 17 confirmed cases and 2 official CVE) in real-world projects. Importantly, QVoG analyzes raw source code without requiring compilation and supports projects exceeding 1 million lines of code.**

***Index Terms*—Code Property Graph; Typestate Analysis; Bug Detection;**

I. Introduction

Behavioral bugs, such as use-after-free, double-free, and resource leaks, pose a significant challenge in modern software systems. Unlike structural vulnerabilities that can often be identified through syntactic patterns, behavioral bugs arise due to incorrect state transitions during program execution. These issues frequently manifest in complex real-world scenarios where objects or resources are accessed in unintended states, leading to security vulnerabilities, crashes, or memory corruption. Many existing static analysis techniques, such as [1] [2] [3], primarily focus on code similarity, hence struggle to detect these issues because they fail to capture the temporal aspects of program execution. Additionally, while some existing approaches offer deeper insights into program behavior, they may rely on computationally intensive methods such as symbolic execution, constraint solving, and abstract interpretation [4] [5] [6]. They can be prohibitively expensive in terms of time and resources, making them impractical for large-scale software projects.

To address security and reliability concerns, graph-based code analysis has emerged as a powerful technique in static analysis. Code Property Graph (CPG) [7] unifies multiple program representations — such as Abstract Syntax Trees (AST), Control Flow Graph (CFG), and Program Dependency Graph (PDG) — into a single intermediate representation. This graph-based approach has proven highly effective for query-driven bug detection. By leveraging predefined graph traversal rules, CPG-based methods can identify structural vulnerabilities such as injection attacks, buffer overflows or insecure API usage patterns [8] [9] [10] [11].

However, despite their success in structural analysis, CPG face fundamental challenges in detecting behavioral vulnerabilities. The core issue lies in the fact that CPG primarily model program structure rather than execution behavior, making them ill-suited for reasoning about state changes and object lifecycles. This limitation leads to two key problems:

First, CPG lacks explicit state-tracking mechanisms, making it difficult to analyze resource management patterns within software. Many behavioral bugs arise from incorrect object usage sequences — such as improper memory deallocation, premature resource release, or unexpected state transitions in finite-state machines. Without a way to model object lifecycles and enforce typestate rules [12] [13], conventional CPG-based analysis cannot effectively detect these vulnerabilities. Moreover, modern software often involves complex interactions between objects across multiple functions or modules, where aliasing—where multiple references point to the same underlying object—plays a critical role. However, CPG frequently struggles to accurately track these alias relationships and inter-procedural dependencies, further limiting their ability to capture complex behavioral issues.

Second, CPG's scalability is severely hindered by graph complexity. Traditional CPG is constructed at the AST level, resulting in excessively large and redundant graph representations as the code base grows. Since AST-based structures retain fine-grained syntactic details, they introduce unnecessary noise, increasing memory overhead and slowing down query execution. This high computational cost makes it impractical to apply CPG-based analysis to large-scale software projects,

* Equal contribution, listed in alphabetical order
# Corresponding author

particularly those exceeding millions of lines of code.

Another significant challenge is generalization across different programming languages. While many bug detection techniques aim to support multiple languages, variations in syntax, semantics, and memory management models necessitate language-specific adaptations. Most existing CPG-based analysis frameworks rely on language-dependent features, requiring developers to manually redefine analysis rules for each new language. The lack of a unified cross-language representation hinders the portability and extensibility of existing detection methods.

These challenges underscore the need for a more expressive, scalable, and language-agnostic program representation — one that not only captures structural relationships but also models object states over time while remaining efficient for large-scale software analysis.

To address these limitations, we introduce QVoG, a query-based static analysis framework that enhances CPG with graph simplification and typestate analysis. Specifically, QVoG streamlines CPG representations by reducing redundancy in AST nodes and eliminating unnecessary dependencies, thereby improving efficiency. Additionally, it integrates state-tracking mechanisms by introducing refined data-flow edges and alias analysis, enabling precise reasoning about state transitions, object lifecycles, and resource management patterns. By combining structural analysis with behavioral modeling, QVoG effectively bridges the gap between structural and behavioral bug detection, leading to more accurate and efficient static program analysis. Specifically, QVoG incorporates the following key innovations:

- **Simplified CPG Structure.** To alleviate the excessive complexity and redundancy in the traditional CPG representations like those in Joern [14], we simplify the CPG by constructing it at the statement level while integrating CFG, Data Flow Graph (DFG), and Call Graph (CG) into a unified representation. This reduces graph bloat, improving both efficiency and scalability.
- **Scalable and Adaptable Detection.** To minimize the difference between languages, we adopt a generalized detection algorithm inspired by LLVM's [15] design philosophy, where all attributes inherit from a common Value-based structure. This abstraction enables the transformation of language-specific data into a unified format, allowing for cross-language compatibility.
- **Behavioral Consistency Enforcement.** To detect behavioral bugs, we leverage typestate analysis to construct a formal model of expected object or API usage patterns. Once the corresponding Deterministic Finite Automaton (DFA) is established, we transform the analysis into a Domain-Specific Language (DSL) query, enabling efficient and automated detection of violations.

We evaluate QVoG across three key dimensions: graph size and query efficiency, behavioral bug detection performance, and real-world applicability. Our results demonstrate that the optimized CPG representation significantly reduces graph size,

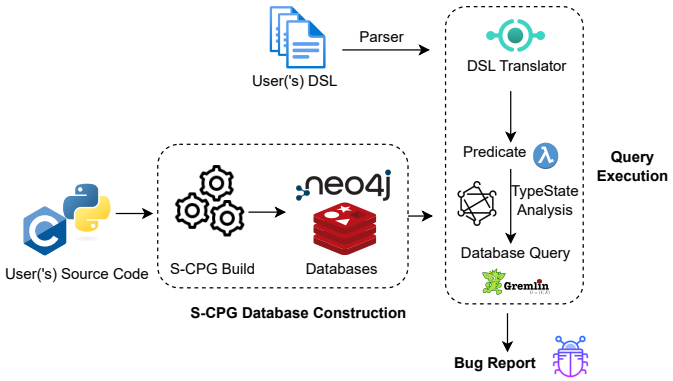

Fig. 1: Workflow of QVoG

containing only one-tenth the number of nodes and one-twentieth the number of edges compared to Joern. Additionally, QVoG outperforms Joern in computational efficiency, using nearly half the CPU time. In behavioral bug detection, QVoG achieves a precision of 84.21% and a recall of 89.54%, showcasing its effectiveness in identifying complex issues. Furthermore, in real-world applications, QVoG has reported 25 security issues, with 17 confirmed by developers and 2 assigned CVE numbers, highlighting its practical impact and reliability. QVoG is publicly available as of now [16]–[19].

II. Methodology

To address the aforementioned challenges, we propose QVoG, a static analyzer based on simplified CPG, referred to as S-CPG in subsequent chapters. QVoG is designed to be a multi-language framework for detecting behavioral bugs using typestate modeling. This section first explains detailed description of the overall design of the framework. Then we will introduce the functionality of its individual modules, the detection algorithm, the S-CPG, and how users can interact with the system.

A. Workflow of QVoG

As shown in Figure 1, to support behavioral bug detection, we employ a deterministic finite automaton (DFA) to model program state transformations. DFA can be represented using a specially-designed DSL. To support bug detection, QVoG builds S-CPG using static analysis tools tailored to different languages, and stores the constructed S-CPG into a hybrid database. Then, the behavioral bug detection is performed by executing the queries modeled by DFA on S-CPG. Those phases are decoupled to enable greater flexibility and modularity.

To optimize performance, QVoG employs a hybrid storage strategy: S-CPG is stored in a graph database, while function summaries and other metadata are cached in a separate database to optimize query performance by minimizing redundant computations. By structuring the workflow in a modular and efficient manner, QVoG ensures scalable and adaptable static analysis for detecting software bugs.

B. Predicate-based Behavioral Bug Detection

To address behavioral bugs such as use-after-free (UAF), which often arise from incorrect state transitions of variables leading to unintended or unsafe program behavior, we employ a DFA to model and manage these state transformations. The DFA serves as a structured framework to define and enforce valid state transitions, ensuring that variables are used in a safe and predictable manner. To realize the functionality of the automaton, we define two pivotal components: FilterPredicate and FlowPredicate. These predicates act as foundational mechanisms for filtering and simulating state transitions within the automaton. Before delving into the specifics of these predicates, it is essential to establish the key detection principles underpinning QVoG. The predicate-based query library provides a systematic and language-agnostic approach to analyzing program behavior. By transforming program data into the engine's internal data structures, the library enables efficient filtering and processing of this data. Following this overview, we will formally define the automaton and predicates, providing a rigorous foundation for their application in detecting and mitigating behavioral bugs.

Definition 1 (Typestate Automaton)**.** A variable's typestate is modeled as a *deterministic finite automaton* (DFA) $M = (S, \Sigma, \delta, s_0, F)$, where:

- S is a finite set of states (e.g., $\{\texttt{INIT}, \texttt{ALLOC}, \texttt{FREED}\}$).
- Σ is a set of program operations (e.g., $\{\texttt{alloc}, \texttt{free}, \texttt{use}\}$).
- $\delta : S \times \Sigma \to S$ defines state transitions.
- $s_0 \in S$ is the initial state.
- $F \subseteq S$ denotes safe terminal states.

Definition 2 (FilterPredicate)**.** The FILTERPREDICATE $\phi(n)$ selects AST nodes n that correspond to critical operations (e.g., allocation/free) σ for state tracking:

$$\phi(n, \sigma) := \begin{cases} 1 & \text{if } \exists \sigma \in \Sigma,\ n \text{ is operated by operation } \sigma, \\ 0 & \text{otherwise.} \end{cases}$$

which operates purely on syntax, independent of the current state s.

Definition 3 (FlowPredicate)**.** The FLOWPREDICATE $\psi(u, v)$ verifies whether two nodes u, v in the program graph (S-CPG) can participate in a valid state transition sequence:

$$\psi(u, v) := \begin{cases} 1 & \text{if } \exists \text{ path } u \rightsquigarrow v \text{ with operations } \sigma_1, ..., \sigma_k \text{ s.t.} \\ & \quad \delta^*(s_i, \sigma_1 ... \sigma_k) = s_j \text{ in automaton } M, \\ 0 & \text{otherwise.} \end{cases}$$

where δ^* extends δ to path sequences. This predicate requires analyzing both graph connectivity and state transition logic.

Consider the UAF problem as an example. The source $\rightarrow$ sink model provides a structured approach for identifying potential vulnerabilities. In this model, a free(v) operation acts as the source, while a use(v) operation serves as the sink, indicating a potential UAF violation. However, this simple model is insufficient on its own. In practice, certain operations, such as realloc(v), can safely reallocate memory, effectively mitigating the risk of UAF vulnerabilities. We classify such operations as barriers, as they prevent unsafe access to freed memory. By leveraging the barrier mechanism, we can significantly reduce the number of meaningless paths that would otherwise need to be considered. This approach enhances the efficiency of the analysis, avoiding the need for SMT solving, which might still yield imprecise results. A UAF violation is flagged if execution follows a source $\rightarrow$ sink pattern without encountering a barrier. Conversely, a source $\rightarrow$ barrier $\rightarrow$ sink pattern ensures proper memory management and prevents UAF issues. This requirement is formally expressed as a Linear Temporal Logic (LTL) constraint. This invariant ensures that after a free(v) operation, use(v) is prohibited until a realloc(v) occurs.:

$$G\left(\texttt{free}(v) \rightarrow X\left(\neg \texttt{use}(v)\, \mathcal{U}\, \texttt{realloc}(v)\right)\right)$$

where

- G (Globally) ensures the condition holds at all times
- X (Next) enforces the condition in the next state
- U (Until) requires the condition to hold until another condition is met

Algorithm 1: UAF Detection via FilterPredicate and FlowPredicate

```
Input: S-CPG G;
Deterministic Finite Automaton (DFA) M;
Variable v
Output: Violation flag (boolean)
S_free ← FilterPredicate(G, M, FREED, v);
S_use ← FilterPredicate(G, M, USE, v);
S_realloc ← FilterPredicate(G, M, REALLOC, v);
foreach u ∈ S_free do
    foreach w ∈ S_use do
        if FlowPredicate(G, M, u, w) then
            if ∄ r ∈
              S_realloc such that FlowPredicate(G, M, u, r)∧
              FlowPredicate(G, M, r, w) then
                return VIOLATION;
            end
        end
    end
end
return SAFE;
```

Detecting behavioral bugs involves two key processes: node filtering (via FilterPredicate) and relationship evaluation (via FlowPredicate). Integrating these predicates into typestate analysis enables precise tracking of state transitions and efficient detection of memory safety violations. The FilterPredicate identifies critical nodes (e.g., free(v), use(v)) within the AST, while the FlowPredicate validates the logical flow between nodes in the program graph. Specifically, it can trace the data flow to determine whether the source and sink operate on the same variable. Additionally, during DFG traversal, we

directly construct alias mappings for the source variable using AssignStatement, improving detection accuracy by capturing indirect references. After establishing data flow relationships, we can further leverage control flow and the call graph to verify whether the sink is actually reachable from the source when necessary. By combining these predicates with the LTL invariant, we ensure memory safety by verifying that all execution paths conform to the UAF constraint. The formal algorithm for UAF detection is provided in Algorithm 1.

Algorithm 2: Optimized Data FlowPredicate via DFG

Input: Graph: DFG; Node u, w;
Output: Data Flow existence between u and w
$N_{\text{reachable}} \leftarrow$ sort(ReachableNodes(DFG, u));
foreach $r \in N_{reachable}$ **do**
  if $r == w$ **then**
    return *TRUE*;
  end
  if *CacheContains*(r) **then**
    continue;
  end
  CacheAdd(r);
  if *FlowPredicate*(DFG, r, w) **then**
    return *TRUE*;
  end
end
return *FALSE*;

Behavioral bugs often require path-sensitive algorithms to accurately track variable states across all possible execution paths. However, the path-sensitive nature of these algorithms introduces significant computational overhead, making them time-consuming and resource-intensive. To address this challenge, we propose a cache-based FlowPredicate algorithm for different types of edges, as detailed in Algorithms 2. The cache-based DFG FlowPredicate optimization leverages caching to enhance efficiency by reducing redundant computations. Given two nodes u and w, the goal of data FlowPredicate checks there u and w have data flow relation. To optimize this process, a caching strategy is employed. Specifically, reachable nodes of node u are sorted by line number, and each node is checked for prior visitation. If a node has already been visited, further dataflow traversal for this predicate is discontinued from that node, thereby preventing redundant evaluations and improving performance. The core idea of our approach is to reduce redundancy by leveraging caching mechanisms and node filtering, thereby improving efficiency without compromising accuracy. The CFG-CG cache strategy follows a similar rationale. If a node can only transition along a single edge, it can be cached at the destination node after the first traversal is completed. This approach minimizes redundant computations and enhances efficiency by ensuring that subsequent traversals can directly utilize cached results instead of re-evaluating the same paths.

C. Simplified Code Property Graph Representation

After constructing the abstract DFA for a specific problem, we further design the S-CPG to align with the appropriate analysis mechanisms. As proposed by Yamaguchi [7], the CPG represents source code structure and semantics. However, conventional CPGs, like those in Joern, store AST nodes with numerous edge types, resulting in excessive complexity. For example, the code in Listing 1 generates around 100 nodes and 460 edges, increasing traversal overhead and hindering analysis performance. To address this, we propose optimizations to streamline the CPG, reducing its complexity while ensuring it remains effective for detecting behavioral bugs.

Listing 1: Example C program to extract CPG

```
1  #include <stdio.h>
2  int main(void) {
3      int a = 2;
4      int b = a * a;
5      if (b > a) {
6          b = b - a;
7      }
8      printf("a + b = %d\n", a + b);
9      return 0;
10 }
```

Specifically, we propose a simplified CPG that stores the code at the level of statement. As shown in Figure 2, instead of storing AST nodes directly in the graph database, we make them attributes of the statement to which they belong, as there is no need to traverse over the original AST structure. This conclusion is drawn from actual observations. Behavioral bug detection using graph queries typically [20] depends on path-based problem detection. Given this characteristic, it is often unnecessary to expand all paths at every AST node, allowing us to omit a significant amount of redundant information, thereby improving analysis efficiency and readability. When detailed statement information is required, QVoG can directly delve into the node's attribute to retrieve its specific information, and then perform a more precise analysis to ensure detection accuracy and effectiveness. After our optimization, the graph for Listing 1 consists of only 10 nodes and 15 edges.

More specifically, S-CPG can be defined as $G = (V, E)$, where V represents a statement in the source code whose attributes are shown in the following list:

- $file$: The full path of the file this statement belongs to.
- $lineno$: The line number of this statement in the file.
- $code$: The original code of the statement.
- ast: The minimum AST in JSON format.
- $properties$: (optional) Other properties, such as function name if the statement includes a function call.

Moreover, even if we are building the graph over statement granularity, a single statement may still contain multiple expressions. Taking a *for-loop* as an example, QVoG will expand the AST structure, and build initialization, condition, and step expression separately.

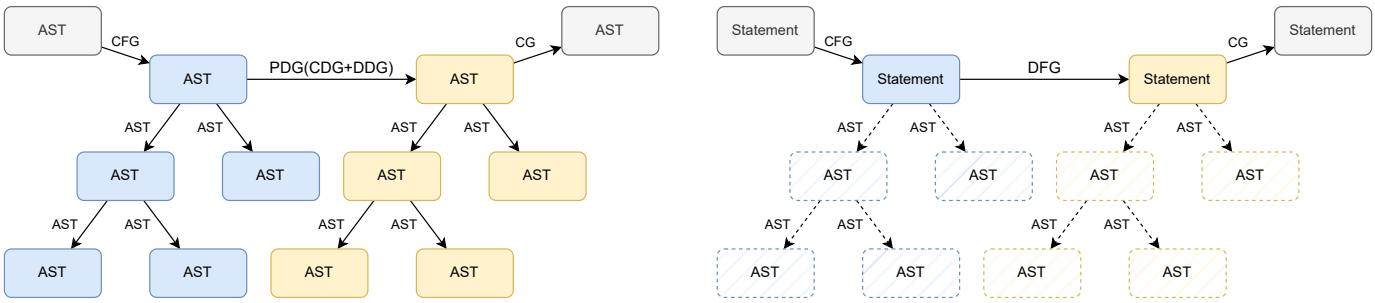

Fig. 2: Comparison of CPG before and after optimization

E represents the flow between statements. Eq. 1 to Eq. 4 presents the formal definition of E. In our implementation, QVoG mainly combines CFG, DFG, and CG to build the S-CPG.

$$E = E_{\text{cfg}} \cup E_{\text{dfg}} \cup E_{\text{cg}} \tag{1}$$
$$E_{\text{cfg}} \subseteq V \times \{\text{true}, \text{false}, \epsilon\} \times V \tag{2}$$
$$E_{\text{dfg}} \subseteq V \times \mathcal{V} \times \{\text{def-use}, \text{decl-use}\} \times V \tag{3}$$
$$E_{\text{cg}} \subseteq V \times \mathcal{F} \times 2^{\mathcal{P}} \times V \tag{4}$$

where:

- $\mathcal{V}$: Set of program variables (local/global)
- $\mathcal{F}$: Set of function identifiers
- $\mathcal{P}$: Parameter mapping $\{(param_1, actual_1), \dots\}$

Definition 4 (Control Flow Edge (via (2)))**.** For an edge $(u, l, v) \in E_{\text{cfg}}$, label l is defined as:

$$l = \begin{cases} \text{true}, & \text{if cond}(u) \wedge \text{true\_br}(v), \\ \text{false}, & \text{if cond}(u) \wedge \text{false\_br}(v), \\ \epsilon, & \text{otherwise}, \end{cases} \tag{5}$$

where:

$$\begin{aligned} \text{cond}(u) &:= u \text{ is a conditional statement}, \\ \text{true\_br}(v) &:= v \text{ is the true branch target}, \\ \text{false\_br}(v) &:= v \text{ is the false branch target}. \end{aligned}$$

Definition 5 (Data Flow Edge (via (3)))**.** A data flow edge is defined as follows:

1) **Def-Use Edge**:

$$\begin{aligned} (u, x, \text{def-use}, v) \in E_{\text{dfg}} \iff & \text{def}(u, x) \wedge \text{use}(v, x) \\ & \wedge \neg \exists w \in \text{path}(u, v), \\ & \text{def}(w, x). \end{aligned} \tag{6}$$

where:

- $\text{def}(n, x)$: Boolean predicate that is true if node n defines (i.e., assigns a new value to) variable x.
- $\text{use}(n, x)$: Boolean predicate that is true if node n uses (i.e., reads the value of) variable x.
- $\text{path}(u, v)$: Represents all nodes on any path from node u to node v in the control flow graph, exclusive of u and v.

2) **Decl-Use Edge**:

$$(u, x, \text{decl-use}, v) \in E_{\text{dfg}} \iff \text{decl}(u, x) \wedge \text{use}(v, x). \tag{7}$$

where (in addition to above):

- $\text{decl}(n, x)$: Boolean predicate that is true if node n declares (i.e., introduces into scope) variable x.

Definition 6 (Call Graph Edge (via (4)))**.** An edge $(u, f, \mathcal{P}, v) \in E_{\text{cg}}$ exists if:

$$\begin{aligned} v &= \text{entry}(f), \\ \forall (p_i, x_i) \in \mathcal{P},\ & p_i \in \text{param}(v), x_i \in \text{actual}(u). \end{aligned}$$

where:

- $\text{entry}(f)$: Function that returns the unique entry node of function f.
- $\text{param}(v)$: Function that returns the set of formal parameters passed to a function call occurring at node v.
- $\text{actual}(u)$: Function that returns the set of actual arguments passed in a function call at node u.

Traditionally, CPG incorporates a Program Dependence Graph (PDG) [21], which is a composition of the Control Dependence Graph (CDG) and the Data Dependence Graph (DDG). Deviating from traditional CPGs, our methodology first eliminates the CDG component.. In practical use, the CDG is primarily employed to identify constraints at specific program points for SMT solving. However, in real-world scenarios, these constraints are often complex and computationally expensive to resolve, making the approach inefficient. Therefore, for behavioral bug detection, we adopt an alternative algorithmic mechanism, predicate-based query, as described in II-B. Additionally, we simplify the DDG by leveraging the DFG. Since the DDG essentially represents a CFG-emulated version of the DFG, and the CFG-emulated structure can be reconstructed through CFG traversal, we can eliminate redundancy by caching the result after the first use.

Since we use typestate techniques to model program behavior, each statement effectively represents a variable's state transition, enabling precise tracking of changes throughout

program execution. To further improve alias analysis and better support typestate analysis, we refine data flow edges by classifying them into two distinct categories. The first category consists of def-use edges, which capture the relationship between a variable's definition and its subsequent usage within the same function. This includes function parameters, actual arguments, and return values. These edges help resolve issues related to variable scoping and ensure that the correct definition is linked to its usage. The second category is decl(are)-use edges, which are critical for tracking variables across different functions. These edges are particularly useful in cases where data flow is implicit and does not form an explicit, direct connection. For example, when a function initializes a pointer variable and passes it to another function, the data flow between these functions may not be immediately apparent in a straightforward def-use relationship. By introducing declare-use edges, we can more accurately capture such implicit flows, thereby improving the precision of our analysis and ensuring a comprehensive understanding of the program's behavior.

D. Behavioral Bug Detection by Querying S-CPG

Given a program p under analysis, to validate where p satisfies the property defined in DFA, we propose a query-based engine to query S-CPG. Moreover, to enable users to define DFA easily, we design a user-defined DSL to define expected program behavior. The overall design of the query execution engine is shown in Figure 3. On one hand, it serves as the interface for DSL parsing, responsible for receiving, parsing, and processing query requests from the DSL. On the other hand, it acts as the interface for database interaction, translating parsed queries into executable operations for the database and efficiently retrieving and returning the corresponding data. It consists of three parts: language-independent code representation, predicate-based query library discussed in II-B, and DSL translator. During the execution of a query, the engine first fetches the S-CPG information from the database. With the help of the database adapter, the S-CPG is then converted into a language-independent representation. It erases the differences between programming languages with a consistent query interface. Based on this, a set of Fluent APIs is provided to support query operations on the S-CPG. Patterns written in DSL are then translated into query APIs and then executed on the database. Next, we will briefly describe the capabilities of each module.

1) Language-Independent Code Representation: To mask differences between languages, an intermediate code representation is often used. It makes the analysis tool more flexible and scalable for new programming languages [22]. The language-independent code representation is designed based on our S-CPG. The ideas are inspired by LLVM IR and *cpg* [23]. This representation is used for a unified AST structure to represent detailed information of statements. A strong type system is used to ensure a well-formatted code structure and provide the interface with combined language features. Nevertheless, not all differences between programming languages, especially AST structures, can be avoided. For example, the

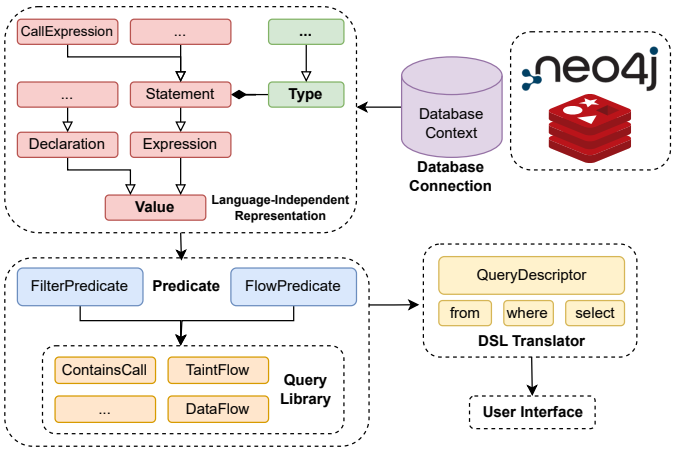

Fig. 3: Query Execution Design

Listing 2: BNF grammar of the DSL

```
1  Q       := from <decl> {, <decl>}
2             [where [not] <FlowPredicate>
3                    {and|or [not] <FlowPredicate>}]
4             select <expr> { , <expr> }
5  <decl> := type name | <FilterPredicate> name
6  <expr> := name | string
```

syntax of function calls in C is different from that in Python, which results in different AST node types and structures. To address this issue, an adapter is implemented to convert the AST of different programming languages into the language-independent code representation defined above. The adapter is designed to be extensible so that users can easily add support for new programming languages. Using the *ast* property stored in S-CPG, we can re-construct the AST and convert it into our language-independent representation.

2) DSL Translator: Once the underlying discrimination mechanisms are fully prepared, we have built a DSL on top of them to provide users with a more convenient interface. The DSL is designed to offer a more intuitive and efficient way to perform queries and operations, allowing users to express complex data filtering and flow analysis requirements through a concise syntax without needing to deeply understand the underlying data structures and processing logic.

This DSL is tightly integrated with II-B, where the `from` clause is utilized for FilterPredicate and the `where` clause for FlowPredicate. This integration enhances the flexibility and precision of node selection and relationship evaluation, enabling more effective and targeted analysis. Specifically, the `from` clause specifies one or more sets of nodes in the graph as the query context, allowing users to define the type of nodes to query using `decl` and optionally add predicates for more precise filtering. The `where` clause applies one or more predicates to refine the context further. For instance, when checking path-based vulnerabilities, the DSL leverages flow information to determine whether a path exists between specified nodes, while also supporting additional predicates

for node filtering. Finally, the `select` clause specifies the nodes to return in the query result. Designed to be simple and intuitive, the query operators are inspired by SQL in syntax but differ in their underlying mechanism, as detailed in the extended Backus–Naur form grammar shown in Listing 2. This combination of familiar syntax and powerful integration with predicates makes the DSL both accessible and robust for querying graph-based data.

III. Implementation

In this section, we provide implementation details of our approach. It involves the S-CPG extraction, the generality of typestate modeling, and optimization. In particular, we have applied multiple optimization strategies to enhance the query performance.

A. S-CPG Extraction

One of the primary focuses of our approach is on extracting S-CPG information. While several analysis tools are available for different programming languages, most only parse the source code and produce an AST. We have extended these tools to extract S-CPG information. Currently, QVoG supports C and Python. The analysis tools used for each language are listed in Table I.

TABLE I: Analysis tool used for S-CPG extraction

Programming Language	Analysis Tool Used
C	Eclipse CDT [24]
Python	Scalpel [25]

As the S-CPG extraction is decoupled, this process can be more flexible, utilizing various techniques as long as the output CPG format meets our definition. This flexibility enhances the analysis and opens the door for potential technological upgrades and functional extensions in the future. The extracted S-CPG will then be stored in a graph database. To ensure maximum portability, we use the standard interface provided by Apache TinkerPop framework. Based on this, we choose Neo4j as our graph database for its performance and reliability [26]. Finally, we embed Gremlin as the graph traversal language to store or retrieve graph information.

B. Generality of Typestate Modeling

Once we successfully design a corresponding automaton model, this approach exhibits strong adaptability and generalizability. For instance, as shown in Figure 4, the model is primarily designed for memory management operations, but it is not limited to this domain. The same methodology can be extended to broader resource management tasks. In other words, any operation that involves strict sequential usage constraints or resources that cannot be reused can be systematically modeled and analyzed using this automaton-based approach. Our method is not confined to memory management, it can be extended to file operations, synchronization mechanisms

https://neo4j.com/

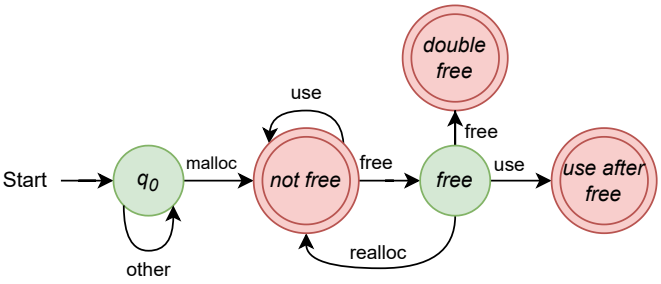

Fig. 4: Memory related detection Automaton

TABLE II: Benchmark Statistics

Benchmark	Project	#Cases	LOC
Juliet	CWE-401	56	0.1k
	CWE-415	39	0.1k
	CWE-416	40	0.1k
Real-word Projects	ubertooth	-	3.7k
	swupdate	-	90k
	barebox	-	385k
	hev-socks5-server	-	2.6k
	SKRTOS_sparrow	-	12k
	teddycloud	-	68k
	RefindPlus	-	118k
	XiUOS	-	175k
	scip	-	1050k
	pastas	-	24k
	PostgreSQL	-	1834k

(such as locks), and other processes governed by topological order constraints. To date, we have successfully applied this approach to memory management, lock synchronization, and file access, as well as to topological order-related issues, such as code injection vulnerabilities. These implementations demonstrate the effectiveness and applicability of this method, providing a general theoretical framework for static analysis that can be flexibly adapted to various types of software security problems.

IV. Evaluation

To validate the effectiveness of our approach, we evaluate our tool to answer the following research questions (RQs).

- **RQ1**: What is the performance of QVoG in terms of time cost?
- **RQ2**: How effective is QVoG compared to existing tools on benchmark datasets?
- **RQ3**: How well does QVoG perform in detecting issues in real-world projects?

Benchmark. To evaluate the effectiveness of QVoG, we use two datasets as shown in Table II. The first one is obtained from the widely recognized Juliet test suites [27]. It contains test cases in various CWEs based on the template and is designed to evaluate the effectiveness of static analysis tools in identifying different security vulnerabilities. We conducted

TABLE III: Graph Size and Query Time Comparison Across Detection Tools

Project	LOC	Source Count	QVoG		Joern	
			Graph Size	Query Time	Graph Size	Query Time
CWE-401	0.1k	3	32V + 63E	2s / 8s	194V + 983E	26s / 9s
ubertooth	3.7k	8	1.2kV + 3.5kE	11s / 12s	98kV + 1107kE	36s / 14s
swupdate	90k	151	38kV + 142kE	51s / 107s	*263kV + 2133kE*	96s / 30s
barebox	385k	329	348kV + 1153kE	155s / 314s	*2375kV + 26374kE*	*213s / 84s*
PostgreSQL	1834k	238	583kV + 1904kE	225s / 460s	Time Out	Time Out

a detailed case study for three common types of bugs, all of them are memory-related.

- **CWE-401 Memory Leak** The program fails to properly release allocated memory, leading to continuous memory consumption during its execution. This is a type of bug of forgetting to do something on a resource.
- **CWE-415 Double Free** The program attempts to free the same memory block twice, which can lead to program crashes or security vulnerabilities. This is a typical bug of unintentionally repeating operations on a resource.
- **CWE-416 Use After Free** The program continues to use a memory block after it has been freed, which can result in unpredictable behavior or crashes. This is a typical bug where a resource is not checked before being operated on.

We selected these CWEs due to their prevalence in real-world vulnerabilities and their generalizable root causes, such as improper allocation, deallocation, or resource management, which can extend to broader CWE categories.

The second dataset is built from real-world well-maintained open-source projects written in C and Python. These projects are famous or recently active in the developer community, with stars ranging from a few hundred to several thousand, reflecting their popularity and widespread use.

All experiments were conducted on a Windows 10 PC with a 2.6GHz Intel Core i7-9570H CPU and 16 GB of memory.

A. RQ1: What is the performance of QVoG in terms of time cost?

As project scale increases, source code analysis becomes increasingly challenging due to growing complexity and resource demands. To evaluate efficiency, we compare QVoG with Joern, a widely used static analysis tool based on CPG. However, detecting all possible paths is computationally infeasible in practice. Therefore, we adopt a similar approach to Joern by setting the "sliceDepth" hyperparameter (Joern's default is 20, while ours is 30). To evaluate QVoG and Joern in projects with different scales, we have selected one project from each magnitude order of 0.1k, 1k, 10k, 100k and 1000k and set the limit of time to 1 hour.

Table III presents the results of graph size and Memory Leak query execution time, evaluated using the RQ2 baseline dataset and RQ3 real-world scenarios. Italics indicate an error in the build process. Our findings reveal that QVoG significantly reduces graph complexity, with the number of nodes approximately one-third of the Lines Of Code (LOC) and the number of edges roughly equal to the LOC. In contrast, Joern exhibits substantial graph expansion, with the number of nodes reaching from tens of times to hundreds of times of the LOC and the number of edges scaling similarly. Moreover, since Joern is unable to export the CPG within an hour, we are unable to determine its exact graph size. This difference underscores the efficiency of our approach in maintaining a compact and manageable graph structure, which directly enhances the query performance.

In terms of query execution time, we report results in the format of "CPU Time / Real Time". The CPU Time results show that QVoG is at least one-third faster than Joern. However, in terms of Total Time, QVoG is slower due to IO overhead, while Joern benefits from in-memory database and multi-core optimizations, enabling it to achieve faster overall execution. Additionally, we measured the "database" construction time for QVoG, which completes building PostgreSQL in 15 minutes, while Joern requires 7 minutes with 1 GB more memory. This confirms Joern's reliance on an in-memory database. We believe that QVoG can be significantly faster if it also relies on in-memory database, which is left as future work. However, Joern does not perform all computations upfront. Certain overlays, such as the dataflow overlay, which is very time-consuming, may be computed when the export command is executed. In general, even when traversing 30 statement edges instead of 20 AST structures, our approach remains significantly faster. This performance advantage underscores the validity of our methodology, as the reduced graph complexity and enhanced CPU efficiency are consistent with our core design objectives.

B. RQ2: How effective is QVoG compared to existing tools on benchmark datasets?

To evaluate the effectiveness of our approach, we compare QVoG with several state-of-the-art static analysis tools, including Joern (2.0.161) [14], CodeQL (2.17.2) [28], Facebook Infer (1.2.0) [29], Clang Static Analyzer (16.04) [30], and SVF Saber (3.0) [31]. These tools represent a diverse range of techniques and capabilities in static code analysis, providing a comprehensive baseline for comparison. We evaluate these tools on Juliet dataset, since it has ground truth vulnerabilities. To detect these issues, QVoG leverages the automaton illustrated in Figure 4, which systematically models the life-

TABLE IV: Verification on Juliet Test Suites

Tool / CWE Type	QVoG		Joern		CodeQL		Infer		Clang SA		SVF Saber	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
CWE-401 "must"	92.50%	66.07%	77.08%	66.07%	**96.96%**	57.14%	0.00%	0.00%	93.02%	76.87%	49.54%	**96.42%**
CWE-401 "maybe"	61.53%	**100.00%**	-	-	**92.00%**	41.07%	-	-	-	-	-	-
CWE-415	85.36%	**92.10%**	**100.00%**	13.15%	**100.00%**	23.68%	**100.00%**	86.64%	**100.00%**	55.26%	96.96%	84.21%
CWE-416	97.43%	**100.00%**	91.67%	57.89%	0.00%	0.00%	0.00%	0.00%	**100.00%**	47.36%	-	-

cycle states and transitions of memory-related bugs, such as allocation, deallocation, and usage patterns.

The results of this extended evaluation are presented in Table IV, where we measure performance using the Precision/Recall Rate. We choose these metrics to provide a comprehensive evaluation of our system's detection capabilities. Precision gauges the purity of our reported findings, ensuring a low rate of false alarms, whereas Recall assesses the system's ability to identify all relevant instances, highlighting its coverage of actual vulnerabilities. Defined as follows:

$$\textbf{Precision Rate} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \times 100\%$$

$$\textbf{Recall Rate} = \frac{\text{True Positives}}{\text{True Positives + False Negatives}} \times 100\%$$

Since CodeQL's CWE-401 includes two query rules, we report two sets of results "must" and "maybe". The "must" results include only high-confidence detected bugs, while the "maybe" results capture all detected issues, including potential false positives. As Joern's query library currently lacks queries for CWE-401 and CWE-415, we developed these queries ourselves. Additionally, SVF Saber does not support the detection of CWE-416, so no results were generated for this bug.

Regarding the experimental data, for CWE-401, QVoG produces 100% recall on the "maybe" setting, outperforming all other tools. While CodeQL has the highest precision (96.96%), it misses almost half the vulnerabilities. For CWE-415, we are still doing well regarding the recall rate (92.10%), while the highest recall of other tools is 86.64%. Infer, CodeQL and Clang SA achieve 100% precision, which is a little higher than ours. Finally, for CWE-416, QVoG outperforms all the other tools in terms of both precision and recall. The superior performance of QVoG is primarily due to the comprehensive graph information, enabling precise control flow analysis and detailed data flow tracking across functions and files. This effectiveness is also closely tied to the robust implementation strategies, which further enhance its ability to handle complex code structures with accuracy.

C. RQ3: How well does QVoG perform in detecting issues in real-world projects?

To highlight the capability of detecting issues across multiple files, we further evaluate QVoG on open-source projects.

At the time of this writing, our analysis has uncovered a total of 25 issues. The list of buggy projects are shown in Table V Among them, 17 issues have been confirmed and fixed by developers based on our reports. Furthermore, our findings were deemed significant enough to be assigned 2 CVE identifier, underscoring the impact and real-world relevance of our approach in identifying security vulnerabilities.

TABLE V: Detected Issues and Developer Confirms

Project	**Detected Issues**	**Developer Confirm**
ubertooth	ML (2)	ML (0)
swupdate	ML (3), DF (1)	ML (3), DF (0)
barebox	ML (2), NPD (1)	ML (0), NPD (0)
hev-socks5-server	NPD (2), ML (3)	NPD (2), ML (3)
SKRTOS_sparrow	NPD (1)	NPD (1)
teddycloud	ML (2)	ML (2)
RefindPlus	NPD (2)	NPD (2)
XiUOS	ML (2)	ML (0)
scip	ML (1)	ML (1)
pastas	ML (3)	ML (3)
PostgreSQL	-	-

NPD: Null Pointer Dereference; ML: Memory Leak; DF: Double Free

Among the identified issues, the majority stem from memory leaks, which can lead to excessive resource consumption and degraded system performance. Additionally, we observed a significant number of null pointer dereference issues and file descriptor leaks, both of which can cause unexpected program crashes or resource exhaustion. In the examples above, we present two types of leak issues: an intra-procedure leak in listing 3 and an inter-procedure leak in listing 4. For clarity, we have made the the necessary simplifications in both examples. In listing 3, the allocation of resource occurs on line `2`. While the resource is properly freed on lines `13` and `16`, it is not released on the execution path leading to line `10`, resulting in a memory leak. In Listing 4, a procedure call is made on line `6`, and within this procedure, a file is opened on line `22`. If execution proceeds normally, the control is returned to the caller. However, if an error occurs in the procedure at line `12`, the previously allocated resource remains unreleased, leading to a file descriptor leak.

The key strength of QVoG lies in its path-sensitive analysis, enabling it to accurately track execution paths across different branches. Moreover, our tool effectively models both inter-procedure data flow and control flow, allowing for precise detection of leaks that span multiple procedures.

D. Discussion

In this section, we discuss the limitation of QVoG and the threats that may affect the validity of our evaluation.

Listing 3: Intra Procedure Memory Leak in swupdate

```
int parse_json(...)
    string = (char *)malloc(size+1);
    if (!string)
        return -ENOMEM;
    ...
    cfg = json_tokener_parse(string);
    if (!cfg) {
        if (...) {
            ...
            return -ENOMEM;
        }
        free(string);
        return -1;
    }
    free(string);
    return ret;
}
```

Listing 4: Inter Procedure File Descriptor Leak in hev-socks5-server

```
static int main_inner (void)
{
    int res;
    char* log_file
    log_file = ...
    res = logger_init(log_file, ...);
    if (res < 0)
        return -2;
    ...
    res = proxy_init (...);
    if (res < 0)
        return -4;
    run ();
    fini ();
}

int logger_init (char *path, ...)
{
    if ...
        ...
    else
        fd = open (path, ...);

    if (fd < 0)
        return -1;
    return 0;
}
```

Limitations. Many behavioral bug patterns require well-defined typestate models for accurate detection. Typestate analysis is essential for tracking the correct sequence of states a resource or object undergoes during its lifecycle. Without a precise and comprehensive typestate model, static analysis tools may miss important transitions or misidentify valid patterns as errors. However, the development of such models is hindered by several factors. First, incomplete documentation is a pervasive issue, as not all software libraries or APIs provide exhaustive details regarding their expected behaviors and state transitions. Second, the continuous evolution of software practices, where new patterns of usage emerge and existing ones evolve, complicates the task of maintaining up-to-date and accurate typestate specifications.

Threats to Validity. In practical detection, syntactic limitations present notable challenges. Our tool, like many static analysis approaches, inherently depends on syntactic representations of code to model program behavior. However, it is impossible to fully capture every syntactic pattern or language feature across diverse software environments. This inherent dependency means that our modeling may miss certain syntactic constructs or fail to represent them adequately. As a result, the construction of S-CPG, which relies on code structure and behavior, may be incomplete or imprecise. Such gaps can cause inaccuracies, including false negatives or positives, as the tool may not fully capture the program's underlying logic.

V. Related Works

In this section, we review existing research on code representation techniques, focusing on graph-based approaches and their applications in static analysis.

A. Code Representation for Static Analysis

Code representations [32], such as metric-based, token-based, and graph-based representations, provide varying levels of abstraction. Metric-based techniques extract software complexity and structural metrics [33] [34], but they fail to capture semantic relationships. Token-based representations, such as Scandariato's text-mining approach [35] and CP-Miner's token-pattern matching [36], offer syntactic insights but lack deeper structural understanding.

Nowadays, most code representation techniques revolve around graphs, as graph-based structures enable various secondary analyses, including code slicing, dependency tracking, and model training. AST, CFG, and PDG have been widely adopted for vulnerability detection [37] [38] [39] [40], where AST preserves syntactic structure, CFG models control flow, and PDG captures data dependencies [41] [42]. However, these representations are often used in isolation, resulting in fragmented information and increased complexity in security analysis. To address this, Yamaguchi et al. [7] introduced the CPG, which unifies AST, CFG, and PDG into a single representation. Despite its advantages, existing CPG-based techniques still generate highly complex graphs, which can impact scalability and query performance. Our work builds upon this foundation by introducing a more streamlined CPG structure, effectively reducing graph complexity while preserving key program properties.

B. Graph-Based bug Detection

Graph-based static analysis has become a powerful approach for bug detection, utilizing the inherent structure of code. These methods can be broadly categorized into traditional static program analysis and deep learning-driven approaches.

Traditional static analysis constructs graph-based representations at the intermediate representation (IR) level, such as LLVM IR [43], to model control and data flow [44] [45] [46]. While these methods offer precise bug detection, they produce large and computationally expensive graphs, making scalability a challenge. Techniques like symbolic execution

and SMT solving [47] [48] [49] help prune execution paths but significantly increase analysis time.

Deep learning-based approaches use graph representations to detect bugs, including training graph neural networks (GNNs), fine-tuning models like GraphCodeBERT, and prompting large language models (LLMs) [50] [51] [52] [53] [54]. While GNNs capture code dependencies, they struggle with memory constraints as graph size increases. Similarly, LLMs excel at semantic analysis but face token length limitations, leading to loss of critical context in large codebases.

To address these issues, we construct S-CPG directly from raw source code and use predicate-based method to effectively eliminate irrelevant branches, improving efficiency.

VI. Conclusion

Detecting behavioral bugs caused by incorrect state transitions remains a challenging problem, as these issues often depend on specific execution paths. Traditional CPG, while effective in integrating multiple code representations, suffer from excessive redundancy and rigid connection rules, limiting their scalability in large-scale software analysis. To address these challenges, we proposed QVoG, a scalable framework that enhances CPG through graph simplification and typestate analysis. By refining node granularity and selectively adding or removing edges, QVoG restructures CPG at the statement level, significantly reducing graph size while preserving essential control and data flow relationships. This reduction improves efficiency without compromising accuracy. Additionally, our framework incorporates state-aware analysis, enabling precise detection of object lifecycle violations through pattern-based rule matching. QVoG 's lightweight design allows it to analyze raw source code without requiring compilation, making it practical for large projects exceeding one million lines of code. In our evaluation on real-world software, QVoG effectively identified 25 behavioral bugs, including 17 confirmed cases and 2 CVEs. These results demonstrate that combining graph simplification with typestate analysis provides an efficient and scalable solution for detecting complex behavioral bugs in modern software systems.

VII. ACKNOWLEDGMENT

This work was supported by the National Key R&D Program of China No 2024YFB4506200, Aeronautical Science Foundation of China with Grant No 20240058051002, and National Natural Science Foundation of China under Grant No 62202026.

References

[1] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd annual conference on computer security applications*, pages 201–213, 2016.

[2] Junwei Zhang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li. Vulnerability detection by learning from syntax-based execution paths of code. *IEEE Transactions on Software Engineering*, 49(8):4196–4212, 2023.

[3] Hao Sun, Lei Cui, Lun Li, Zhenquan Ding, Zhiyu Hao, Jiancong Cui, and Peng Liu. Vdsimilar: Vulnerability detection based on code similarity of vulnerabilities and patches. *Computers & Security*, 110:102417, 2021.

[4] Sicheng Luo, Hui Xu, Yanxiang Bi, Xin Wang, and Yangfan Zhou. Boosting symbolic execution via constraint solving time prediction (experience paper). In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 336–347, 2021.

[5] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1232–1243, 2014.

[6] Kasper Luckow, Rody Kersten, and Corina Pasareanu. Complexity vulnerability analysis using symbolic execution. *Software Testing, Verification and Reliability*, 30(7-8):e1716, 2020.

[7] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *IEEE Symposium on Security and Privacy*. IEEE, May 2014.

[8] Fangcheng Qiu, Zhongxin Liu, Xing Hu, Xin Xia, Gang Chen, and Xinyu Wang. Vulnerability detection via multiple-graph-based code representation. *IEEE Transactions on Software Engineering*, 2024.

[9] Weining Zheng, Yuan Jiang, and Xiaohong Su. Vu1spg: Vulnerability detection based on slice property graph representation learning. In *IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 457–467. IEEE, 2021.

[10] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. Mvd: memory-related vulnerability detection based on flow-sensitive graph neural networks. In *Proceedings of the 44th international conference on software engineering*, pages 1456–1468, 2022.

[11] Peng Wu, Liangze Yin, Xiang Du, Liyuan Jia, and Wei Dong. Graph-based vulnerability detection via extracting features from sliced code. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 38–45. IEEE, 2020.

[12] Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):1–34, 2008.

[13] Eric Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 5–14, 2010.

[14] T. J. Team. Joern. Online, 2024. Available: https://joern.io.

[15] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[16] QVoG Team. C to s-cpg. Online, 2025. Available: https://doi.org/10.5281/zenodo.16395947.

[17] QVoG Team. Python to s-cpg. Online, 2025. Available: https://doi.org/10.5281/zenodo.16396285.

[18] QVoG Team. Query execution engine of qvog. Online, 2025. Available: https://doi.org/10.5281/zenodo.16396307.

[19] QVoG Team. Query of qvog. Online, 2025. Available: https://doi.org/10.5281/zenodo.16396335.

[20] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. Path-sensitive and alias-aware typestate analysis for detecting os bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–872, 2022.

[21] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[22] Konrad Weiss and Christian Banse. A language-independent analysis platform for source code. March 2022. arXiv:2203.08424 [cs.CR].

[23] Alexander Küchler and Christian Banse. Representing llvm-ir in a code property graph. In *25th Information Security Conference*, ISC. Springer, 2022.

[24] Danila Piatov, Andrea Janes, Alberto Sillitti, and Giancarlo Succi. Using the eclipse c/c++ development tooling as a robust, fully functional, actively maintained, open source c++ parser. *OSS*, 378:399, 2012.

[25] Li Li, Jiawei Wang, and Haowei Quan. Scalpel: The python static analysis framework. February 2022. arXiv:2202.11840 [cs.SE].

[26] Rahmatian Jayanty Sholichah, Mahmud Imrona, and Andry Alamsyah. Performance analysis of neo4j and mysql databases using public policies decision making data. In *2020 7th International Conference on Infor-*

mation Technology, Computer, and Electrical Engineering (ICITACEE). IEEE, September 2020.

[27] Paul E Black and Paul E Black. *Juliet 1.3 test suite: Changes from 1.2*. US Department of Commerce, National Institute of Standards and Technology, 2018.

[28] Dongjun Youn, Sungho Lee, and Sukyoung Ryu. Declarative static analysis for multilingual programs using codeql. *Software: Practice and Experience*, 53(7):1472–1495, 2023.

[29] Dominik Harmim, Vladimır Marcin, and Ondrej Pavela. Scalable static analysis using facebook infer. *I, VI-B*, 2019.

[30] Clang project Team. Clang static analyzer. Online, 2024. Available: https://clang-analyzer.llvm.org/.

[31] Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 254–264, 2012.

[32] Yixin Yang, Ming Wen, Xiang Gao, Yuting Zhang, and Sun Hailong. Reducing false positives of static bug detectors through code representation learning. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2024.

[33] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.

[34] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540, 2007.

[35] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, 2014.

[36] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, 32(3):176–192, 2006.

[37] Viet Hung Nguyen and Le Minh Sang Tran. Predicting vulnerable software components with dependency graphs. In *Proceedings of the 6th international workshop on security measurements and metrics*, pages 1–8, 2010.

[38] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12. ACM, December 2012.

[39] Nam H Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, pages 447–456, 2010.

[40] Jingyue Li and Michael D Ernst. Cbcd: Cloned buggy code detector. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 310–320. IEEE, 2012.

[41] Vincent Hendryanto Halim and Yudistira Dwi Wardhana Asnar. Static code analyzer for detecting web application vulnerability using control flow graphs. In *2019 International Conference on Data and Software Engineering (ICoDSE)*. IEEE, November 2019.

[42] Samuel Bates and Susan Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 384–396, 1993.

[43] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122, 2014.

[44] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.

[45] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 480–491, 2007.

[46] Jisheng Zhao, Michael G Burke, and Vivek Sarkar. Parallel sparse flow-sensitive points-to analysis. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 59–70, 2018.

[47] Maria Christakis, Peter Müller, and Valentin Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering*, pages 144–155, 2016.

[48] Silvio Ranise and Cesare Tinelli. Satisfiability modulo theories. *Trends and Controversies-IEEE Intelligent Systems Magazine*, 21(6):71–81, 2006.

[49] Dong Chen, Yang Zhang, Liang Cheng, Yi Deng, and Xiaoshan Sun. Heuristic path pruning algorithm based on error handling pattern recognition in detecting vulnerability. In *2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*, pages 95–100. IEEE, 2013.

[50] Xin-Cheng Wen, Cuiyun Gao, Shuzheng Gao, Yang Xiao, and Michael R Lyu. Scale: Constructing structured natural language comment trees for software vulnerability detection. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 235–247, 2024.

[51] Yutao Hu, Suyuan Wang, Wenke Li, Junru Peng, Yueming Wu, Deqing Zou, and Hai Jin. Interpreters for gnn-based vulnerability detection: Are we there yet? In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1407–1419, 2023.

[52] Rongcun Wang, Senlei Xu, Yuan Tian, Xingyu Ji, Xiaobing Sun, and Shujuang Jiang. Scl-cvd: Supervised contrastive learning for code vulnerability detection via graphcodebert. *Computers & Security*, 145:103994, 2024.

[53] Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software*, 212:112031, 2024.

[54] Yixin Yang, Bowen Xu, Xiang Gao, and Hailong Sun. Context-enhanced vulnerability detection based on large language model. March 2025. arXiv:2504.16877 [cs.SE].