Efficient Bug Detection by Inferring Implicit API Contract of Pointer State Transition

Xingjing Deng, Yunlong Ma, Xiang Gao*, Hailong Sun*
Beihang University, Hangzhou Innovation Institute of Beihang University
{xjdeng, yunlong_ma, xiang_gao, sunhl}@buaa.edu.cn

***Abstract*—Modern C functions heavily rely on implicit API contracts—assumptions about pointer validity that are contingent on execution paths. Violations of these unstated rules, such as dereferencing NULL pointers after failed operations, remain a primary source of reliability bugs. Unfortunately, traditional whole-program analyses and contract-mining techniques fail to effectively capture such implicit semantics. We present CONTRACT, a compositional framework that automatically infers and checks contracts governing pointer state transitions. CONTRACT models function behaviors as state transition systems over abstract pointer resources, synthesizing precise state preconditions and postconditions. This approach enables efficient inter-procedural consistency checking without redundant path exploration. Built atop Clang and applied to diverse real-world C software, CONTRACT uncovered 209 previously unrecognized pointer-semantic inconsistencies. To date, developers have confirmed 127 reports, with 28 fixed and 2 CVEs assigned.**

***Index Terms*—pointer state transition, implicit contracts inference, consistency checking**

I. INTRODUCTION

A substantial portion of bugs in C software not only stem from trivial implementation errors, but also from the misuse of Application Programming Interfaces (APIs) [1]. This phenomenon arises from a semantic gap between a function's expected usage—its implicit contract—and its actual invocation. Empirical studies corroborate this gap: Gu [2] found that over 83% of API misuse bugs are contractual misunderstandings related to pre-conditions, post-conditions, and state management.

In the context of C, these contractual obligations are inextricably linked to memory management. Industry reports from Microsoft and Google show that approximately 70% of all high-severity security bugs are due to memory safety errors [3] [4]. While not all of these issues stem from API misuse, a substantial subset arises from violations of implicit pointer-related contracts — the unstated assumptions about validity, aliasing, and lifetime. For example, `fclose(file)` assumes its argument is a valid and open file pointer; `realloc(p, n)` promises to return a new valid buffer or NULL without freeing `p`. However, enforcing these obligations is notoriously difficult since the C language lacks the formalism to express such constraints [5]. Consequently, these critical requirements persist as informal conventions. Developers must turn to documentation or comments, but these are notoriously unreliable, often being incomplete, outdated, or incorrect [6] [7], which places a high cost of study on programmers [8].

The lack of a reliable, machine-readable source of truth regarding the implicit usage contracts of APIs is a primary cause of many memory management-related bugs, which fuele the research into automated detection techniques. Broadly, existing approaches fall into two families: (1) static analysis that attempts to rediscover contracts from code, and (2) contract mining that infers contracts from external evidence.

Traditional static analysis [9] attempts to uncover these violations primarily through source-sink analysis, focusing on identifying specific pointer operations within the source code. However, these tools are constrained by an inherent trade-off between precision and scalability. Whole-program, path-sensitive analyzers [10] [11] [12] try to rediscover the contract at every call site, but quickly collapse under the state-space explosion. Conversely, more scalable tools avoid whole-program path sensitivity by relying on coarse summaries and sensitive function identification [13] [14] [15], restricting detailed analysis to a small set of "critical" APIs. However, these summaries are typically too coarse to capture the conditional nature of API contracts. This strategy merely shrinks the search space without addressing the fundamental deficit. In both cases, the root cause remains the same: the analyzer lacks a formal definition of the API's complex, conditional contracts.

Recognizing that static analysis alone cannot recover precise contracts without explicit guidance, a prominent line of research has focused on contract mining [16]—automatically learning API contracts from external evidence. However, this approach introduces its own set of significant limitations. Tools that mine API usage patterns [17] [18] [19] [20], whether from large client codebases or from historical bug-fixing patches, are fundamentally probabilistic. They assume that common or patched behavior represents correct usage, an assumption that does not always hold, and their effectiveness is contingent on the availability of a massive, relevant corpus. Tools that mine from API documentation and comments [21] [22] suffer from the notorious unreliability and ambiguity of natural language. Crucially, all these mining approaches often produce informal or semi-formal rules that are difficult to integrate.

We argue that the key to breaking this impasse is to shift from unreliable, external-evidence-based mining to a more robust paradigm: *formal semantic inference*, performed

*Corresponding authors.

directly on the function's implementation and augmented with targeted, external knowledge about common pointer and memory-management idioms.

To this end, we introduce the *Pointer State Transition Contracts (PSTC)*, a novel, formally structured representation of API contracts around pointer state for functions. The cornerstone of our work is the PSTC itself: a formal summary designed to capture the rich, conditional, and state-aware pointer semantics of C functions. Unlike traditional path-insensitive summaries, a PSTC is explicitly conditional, encoding a function's behavior as a set of `(Predicate, State)` pairs. This structure is expressive enough to model complex, real-world API contracts, such as the distinct outcomes of `realloc` on its success and failure paths.

To automatically generate PSTC for functions, we designed a novel, hybrid inference algorithm. It addresses key limitations of traditional contract mining. It first uses a Large Language Model (LLM) as a targeted semantic bootstrapper to infer initial PSTC for opaque library functions, injecting valuable domain-specific knowledge. It then uses a rigorous, bidirectional dataflow analysis, grounded in the function's own source code, to propagate and infer PSTC for all user-defined functions.

Finally, we present CONTRACT, an effective framework for compositional checking that leverages the inferred PSTC. By composing these precise, conditional semantics at call sites, CONTRACT can perform a state-aware analysis, enabling it to track pointer states across complex inter-procedural error-handling paths and detect deep, path-dependent bugs.

In summary, this paper makes the following contributions:

- **Pointer State Transition Contracts (PSTC): A Formal Contract Model.** We introduce the PSTC, a novel, formal representation for API contracts for functions. Its explicit, conditional `(Predicate, State)` structure is designed to capture the rich, path-dependent pointer state semantics of C functions.
- **A Hybrid Inference Algorithm for PSTC Generation.** We design a novel, two-stage algorithm that synergistically combines an LLM for bootstrapping external API semantics with a rigorous, bidirectional dataflow analysis to automatically infer the PSTC for a function.
- **An Effective Framework for PSTC-Guided Check.** We present CONTRACT, an effective framework that leverages the inferred PSTC to perform efficient and precise inter-procedural consistency checking.

We demonstrate CONTRACT's practical effectiveness by evaluating it on a diverse set of open-source projects. Our tool uncovered 209 previously unknown contract violations with a low false-positive rate, demonstrating the power of our semantic inference approach. CONTRACT is available now [23].

II. MOTIVATION

To exemplify the issues arising from missing implicit contracts, we showcase a real-world Null Pointer Dereference bug from the Linux Kernel's DRM subsystem [24] (Listing 1). This bug, while seemingly a simple "Missing Check", reveals a deeper problem rooted in implicit knowledge gap that affects both developers and automated tools.

Listing 1: Null Pointer Dereference bug in Linux Kernel

```
struct drm_display_mode *drm_cvt_mode(struct drm_device *
    dev, ...) {
    struct drm_display_mode *drm_mode;
    drm_mode = drm_mode_create(dev);
    if (!drm_mode)
        return NULL; // Potential NULL value return
    // ..
    return drm_mode;
}

void drm_mode_probed_add(struct drm_connector *connector,
    struct drm_display_mode *mode) {
    list_add_tail(&mode->head, ..); // Dereference
}
static int amdgpu_vkms_conn_get_modes(struct drm_connector
    *connector) {
    struct drm_device *dev = connector->dev;
    struct drm_display_mode *mode = NULL;
    // ..
    for (i = 0; i < ARRAY_SIZE(common_modes); i++) {
        mode = drm_cvt_mode(dev, ..);
+       if (!mode)
+           continue;
        drm_mode_probed_add(connector, mode);
    }
    // ..
}
```

A. The Bug: A Tale of Two Missing Contracts

The bug in `amdgpu_vkms_conn_get_modes` arises from the interaction of two functions: `drm_cvt_mode` and `drm_mode_probed_add`, each with its own unstated, implicit contract. The bug is caused by the violation of two distinct, unwritten contracts.

- The Producer's contract: The function `drm_cvt_mode` is a "producer" of a `drm_cvt_mode` pointer. Its contract includes a critical post-condition: on certain failure paths (memory allocation failure), it will return NULL.
- The Consumer's contract: `drm_mode_probed_add` is the "consumer" of this pointer. Its contract includes a strict pre-condition: the mode parameter must be a valid pointer, as it is dereferenced unconditionally (`&mode->head`).

B. The Core Challenge: The Invisibility of Implicit Contracts

The key reason sophisticated automated tools systematically fail to bridge this dual knowledge gap lies in the semantic invisibility of both contracts.

Fundamentally, since the static analyzer lacks the explicit knowledge that `drm_cvt_mode` may return `NULL`, it is forced to infer this behavior from scratch. For path-sensitive approaches, this inference is anguished by the kernel's complex encapsulation; the analysis attempts to penetrate the deep call chains but often out or loses precision amidst the heavy wrappers, failing to discover the specific path that returns `NULL`. Conversely, approaches based on sensitive function identification fail even earlier. Since `drm_cvt_mode` does not resemble standard security-critical APIs (like memory allocators), it is summarily excluded from the analysis scope. Ultimately, the existing widely used static analysis tools, such

as Infer [25], Clang Static Analyzer [26], fail to detect this bug because the crucial semantic contract is absent for them.

Contract mining from external evidence is fundamentally ill-suited for the rigorous Linux Kernel environment, where many core APIs are highly specialized and lack a statistically significant number of diverse "correct" usage examples to learn from. Mining client code or patches is probabilistic and fails on specialized kernel APIs that lack a sufficient corpus of "correct" examples; indeed, the success of recent tools like IMMI [27] relies on avoiding such fragile, pattern-mining assumptions. Mining from documentation is equally unreliable. This is empirically validated by GPTAid [28], which found that for 61.3% of the precise security rules it generated, the necessary information lacks explicit descriptions in the documentation. In short, both strategies fail to reliably produce the sound, formal contracts required for rigorous bug detection.

C. Our Approach: Making Both Contracts Explicit with PSTC

CONTRACT is designed from the ground up to make this invisible knowledge visible and checkable.

Inferring the Producer's Contract: Our inference algorithm uses forward analysis to analyze `drm_cvt_mode` once and generates a precise, conditional PSTC: `(retval NEQ NULL, retval ↦ VALID), (retval EQ NULL, retval ↦ NULL)` .

Inferring the Consumer's Contract: we also analyzes `drm_mode_probed_add`. Through its backward analysis, it sees the unconditional dereference `&mode->head` and infers a strict precondition: `(arg2 ↦ VALID)`.

Automated Contract Checking: When CONTRACT's checker analyzes the function, it has access to both rules from the PSTC database. At the call to `drm_mode_probed_add`, it sees that the mode variable is in a hybrid state, `VALID` or `NULL` state, but is being passed to a function that requires a `VALID` state. The checker immediately identifies this mismatch and reports the bug. By explicitizing the contract information of the function, we were able to detect this bug successfully.

III. METHODOLOGY

A. Overview

In this section, we present the design of CONTRACT, a framework for formal semantic inference and checking, to address the challenge of reasoning about implicit API pointer contracts. The core principle of CONTRACT is to transform the problem from a blind code exploration into a rigorous, semantics-driven process. It achieves this by first automatically inferring precise, machine-readable contracts—the Pointer State Transition Contracts (PSTC)—for functions, and then using these PSTCs to perform a global consistency check.

As illustrated in Figure 1, CONTRACT's architecture is organized around this central "infer-then-check" workflow, which consists of two main phases:

PSTC Inference. This is the core inference phase where CONTRACT derives the formal semantics for functions targeting at pointer. The process is a novel, hybrid approach that begins by using an LLM as a semantic bootstrapper to infer

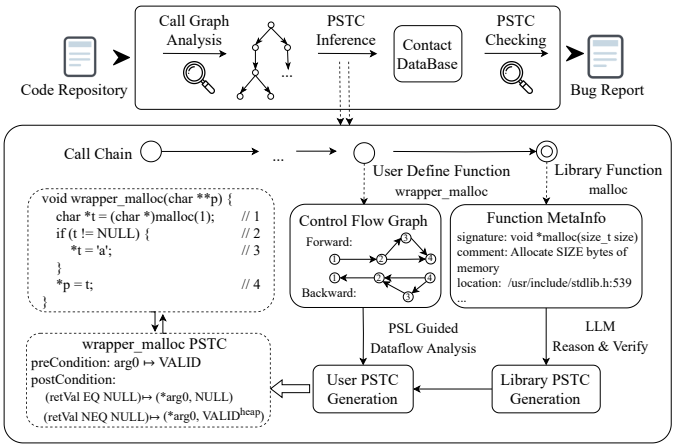

Fig. 1: Overview of CONTRACT

initial PSTCs for opaque library functions. With this external knowledge in place, a rigorous, bidirectional dataflow analysis is then performed on user-defined functions. A forward pass computes the function's conditional post-conditions by tracking its effects on pointer states, while a symmetric backward pass infers the minimal pre-conditions by propagating state requirements from sensitive operations.

PSTC Checking. This phase performs a comprehensive analysis that operates at two levels. At the intra-procedural level, the checker performs a path-sensitive analysis within the function, tracking the evolution of pointer states. At the inter-procedural level—specifically at function call sites—it avoids re-analyzing callee bodies. Instead, it retrieves the callee's pre-computed PSTC and performs two key actions: it checks that the caller's current state satisfies the callee's pre-conditions, and then updates the local state environment according to the callee's post-conditions.

B. Formal Definition of Pointer State Transition Contracts

Before detailing how PSTCs are inferred and checked, we first formally define their structure. A PSTC is designed to be a comprehensive, machine-readable contract of a function's pointer-related state. Conceptually, a PSTC describes transformations over the abstract states of pointers. Therefore, to define the PSTC, we must first establish the foundational vocabulary for these states: our Pointer State Lattice.

1) The Pointer State Lattice (PSL): The Semantic Foundation: To reason formally about program behavior, static analysis must approximate the program's concrete, and often infinite, execution states with a finite set of abstract states. This is the core principle of Abstract Interpretation [29], a foundational theory of program analysis. In this theory, the set of abstract states and their relationships are formally defined as a mathematical structure known as a lattice. For bug detection, the core mechanics rely on two fundamental lattice operations: join operator ($\sqcup$) and meet operator ($\sqcap$). The join operator defines how to conservatively merge abstract states from diverging control-flow paths (approximating the union of behaviors). Conversely, the meet operator calculates the intersection of

these states, identifying properties that must hold true across all paths.

Following this principle, we designed the Pointer State Lattice (PSL) as the specific abstract domain for our analysis. The PSL is a lightweight yet expressive lattice designed to capture the evolving safety and liveness semantics of pointers. It serves as the semantic vocabulary for our entire framework, allowing CONTRACT to track pointer states in a unified, mathematically-grounded manner.

a) The Four Core States: At its core, the PSL abstracts each pointer variable into one of four mutually exclusive states, summarized in Table I. These states represent the fundamental stages of a pointer's lifetime.

TABLE I: The Four Core Abstract States in the PSL.

State	Meaning
`UNINIT`	The pointer is uninitialized or holds a garbage value.
`DANGLING`	The pointer refers to a deallocated memory region.
`NULL`	The pointer explicitly holds the null value.
`VALID`	The pointer refers to a valid, accessible memory region.

Theoretical Foundation: A Risk-Monotonic Lattice. These states form a Risk-Monotonic Lattice defined by the partial order:

$$\texttt{VALID} \sqsubseteq \texttt{NULL} \sqsubseteq \texttt{DANGLING} \sqsubseteq \texttt{UNINIT}$$

Unlike traditional value-based lattices, here the relation $s_1 \sqsubseteq s_2$ denotes that state s_2 strictly subsumes the safety risks of s_1. To rigorously justify this, we define $\Phi(s)$ as the set of Forbidden Operations (operations that trigger a vulnerability) for a pointer in state s. The partial order is defined as the subset relation $\Phi(s_1) \subseteq \Phi(s_2)$.

1. **Risk Subsumption via Forbidden Operations**: We demonstrate the strict inclusion chain by verifying that for every pair $s_1 \sqsubseteq s_2$, state s_2 inherits all forbidden operations of s_1 and adds new ones, with no counter examples (no operation is safe in s_2 but unsafe in s_1).

- $\texttt{VALID} \sqsubseteq \texttt{NULL}$:
  Let $\Phi(\texttt{VALID}) = \emptyset$ (Safe for dereference and free).
  Let $\Phi(\texttt{NULL}) = \{\texttt{dereference}\}$ (Triggers CWE-476).
  Inclusion Logic: $\emptyset \subset \{\texttt{dereference}\}$. The transition introduces a restriction on access, without relaxing any existing rules.
- $\texttt{NULL} \sqsubseteq \texttt{DANGLING}$:
  Let $\Phi(\texttt{DANGLING}) = \{\texttt{dereference}, \texttt{free}\}$.
  Inclusion Logic: While accessing `NULL` typically crashes, accessing a `DANGLING` pointer constitutes a Use-After-Free (CWE-416). Crucially, calling `free(NULL)` is a safe no-op in C, whereas calling `free(DANGLING)` triggers a Double-Free (CWE-415). There is no operation that is safe on a `DANGLING` pointer but unsafe on `NULL`. Thus, the set of unsafe operations for `NULL` is a strict subset of those for `DANGLING`.
- $\texttt{DANGLING} \sqsubseteq \texttt{UNINIT}$:
  An `UNINIT` pointer (CWE-457) holds an indeterminate value, meaning it can non-deterministically alias any address (valid, null, or freed).
  Inclusion Logic: To be sound, the analyzer must assume the worst-case aliasing. Thus, any operation unsafe for `DANGLING` is potentially unsafe for `UNINIT` (inheriting CWE-415/416/476).

2. **Mathematical Soundness via Risk Sets:** Formally, defining the partial order as the inclusion of forbidden operation sets ($\Phi(s_1) \subseteq \Phi(s_2)$) inherently satisfies the lattice axioms. Since set inclusion is Reflexive, Antisymmetric, and Transitive, our domain constitutes a valid Finite Lattice. Consequently, the join ($\sqcup$) operation is mathematically well-defined as the union of forbidden operations ($\Phi(s_1) \cup \Phi(s_2)$), ensuring the analysis conservatively captures all potential violations in merging.

Conservative Abstraction. This ordering is crucial for data-flow analysis. When control-flow paths merge, we use the lattice's *join* operator ($\sqcup$) to conservatively combine pointer states. Based on our definition, the join computes the Least Upper Bound, effectively selecting the Worst-Case Safety Assumption. For example, if a pointer `p` is `VALID` on one path and `NULL` on another:

$$\texttt{VALID} \sqcup \texttt{NULL} = \texttt{NULL}$$

The result is `NULL` because the analyzer must conservatively assume the risk of `dereference` exists, enforcing the stricter safety checks associated with the `NULL` state. Conversely, the meet operation ($\sqcap$) computes the greatest lower bound, representing the intersection of safety properties.

b) Modeling Uncertainty with the Composite `UNCHECKED` State: While the four core "atomic" states capture a pointer's concrete value, they do not fully express the common scenario where a pointer's state is an ambiguous mix of several possibilities pending a check.

To model this directly, we introduce a composite state, denoted as $\texttt{UNCHECKED}(\mathbb{S})$, where $\mathbb{S}$ is a non-empty subset of the atomic PSL states. This composite state represents a pointer that could be in any of the states in $\mathbb{S}$, with the ambiguity to be resolved later. For instance: After `p = malloc()`, the state of `p` becomes $\texttt{UNCHECKED}(\{\texttt{VALID}, \texttt{NULL}\})$. This precisely models that the pointer is either a valid or null, but we don't know which yet. For the purpose of data-flow ordering and merging, its value is interpreted as the join of its contents:

$$\text{Value}(\texttt{UNCHECKED}(\mathbb{S})) \equiv \bigsqcup_{s \in \mathbb{S}} s$$

This definition ensures that whenever an `UNCHECKED` state is utilized, it assumes the properties of the least precise (riskiest) element in $\mathbb{S}$, maintaining the soundness of the risk-monotonic analysis.

c) Transfer Functions over the PSL: To track the evolution of pointer states through a function, CONTRACT models each program statement as a transfer function. A transfer function is a mathematical representation of a statement's operational semantics within our abstract domain.

First, we define the abstract state, σ, at any given program point. The abstract state is a map from every pointer variable v currently in scope to its corresponding state in the PSL:

$$\sigma : \text{Var} \to \text{PSL}$$

For example, after the statements `int x; int *p = &x;`

`int *q = NULL;`, the abstract state would be $\sigma = \{p \mapsto \texttt{VALID}, q \mapsto \texttt{NULL}\}$.

Each statement s in the program is modeled as an abstract transformer, f_s, that takes an input abstract state σ and produces a new output abstract state σ'.

$$f_s : \sigma \mapsto \sigma'$$

Table II provides a semi-formal definition for several representative transfer functions. For most statements, the new state σ' is identical to σ except for the variables modified by the statement.

TABLE II: Representative Transfer Functions over PSL States.

Operation	New State $\sigma'(p)$	Note
`p = malloc(...)`	`UNCHECKED`	Models fallible allocation.
`p = &(x)`	`VALID`	Address-of operator.
`p = NULL`	`NULL`	Explicit null assignment.
`free(p)`	`DANGLING`	Release the memory space.

By iteratively applying these transfer functions over a function's control-flow graph (CFG), CONTRACT computes the set of all reachable pointer states at each program location.

2) Structure of PSTC: A Formal Contract Model: While the PSL provides a vocabulary for the state of individual pointers, we need a higher-level structure to formally describe the complete, state-aware contract of an entire function.

Definition 1 (Access Path)**.** The domain of valid *Access Paths* ($\mathcal{AP}$) represents all memory locations observable at the function interface. It is defined inductively:

$$\pi \in \mathcal{AP} ::= \mathbf{retval} \mid \mathbf{arg}_i \mid \pi \to f$$

where:

- $\mathbf{retval}$ denotes the function's return value (the output root).
- $\mathbf{arg}_i$ denotes the i-th formal argument (the input roots).
- $\pi \to f$ denotes a field access f relative to a base path π.

Definition 2 (Pointer State Transition Contracts)**.** A *Pointer State Transition Contracts, PSTC,* for a function f, denoted as $\mathcal{PSTC}_f$, is a tuple $\langle \mathcal{PRE}_f, \mathcal{POST}_f \rangle$, where:

- $\mathcal{PRE}_f : \mathcal{AP} \mapsto \mathcal{S}$ is the Precondition, mapping access paths to their required input abstract states.
- $\mathcal{POST}_f = \{(\phi_1, \sigma_1), \ldots, (\phi_n, \sigma_n)\}$ is the set of Postconditions. Each pair represents a distinct, conditional execution outcome, where:
  1) A Guard Predicate (ϕ_i), formalized as a structured tuple to ensure analytical tractability:
  $$\phi_i := (\pi, \texttt{op}, \texttt{val})$$
  where $\pi \in \mathcal{AP}$ is the target path, `val` is a sentinel constant (e.g., `NULL`, `ZERO`), and `op` is drawn from a restricted relational set:
  $$\texttt{op} \in \{\texttt{EQ}, \texttt{NEQ}, \texttt{GT}, \texttt{GTE}, \texttt{LT}, \texttt{LTE}\}$$
  2) A Conditional State Mapping ($\sigma_i : \mathcal{AP} \to \mathcal{S}$), which captures the snapshot of side effects on relevant access paths *if and only if* ϕ_i holds.

a) Meta-information: In addition to the core PSL state, our implementation can attach optional meta-information to each state, such as the allocation origin (`Heap`, `Stack`). This information is used to enhance the precision of specific checks (e.g., ensuring only heap memory is passed to `free`) but is kept separate from the primary lattice structure to maintain its simplicity and generality.

b) Example: A simplified PSTC for the `wrapper_malloc` function in figure 1, as inferred by CONTRACT, is:

$$\left\{ \begin{array}{l} \mathcal{PRE} : \{\, \texttt{arg0} \mapsto \texttt{VALID} \,\}, \\ \mathcal{POST} : \\ \left\{ \begin{array}{l} (\texttt{*arg0 EQ NULL}) \mapsto (\texttt{*arg0} \mapsto \texttt{NULL}) \\ (\texttt{*arg0 NEQ NULL}) \mapsto (\texttt{*arg0} \mapsto \texttt{VALID}^{\texttt{Heap}}) \end{array} \right\} \end{array} \right\}$$

This PSTC states that `wrap_malloc` requires its first parameter to be `VALID` before invocation (unconditional pre-condition), and its return value follows two possible outcome branches: If the allocation fails, the return value is `NULL`; otherwise, it is a `VALID` heap pointer. Its behavior is synthesized as a set of two disjoint postconditions, effectively capturing its conditional nature:

$$\{(\texttt{*arg0}, \texttt{EQ}, \texttt{NULL})), (\texttt{*arg0}, \texttt{NEQ}, \texttt{NULL})\}$$

When the analysis later encounters a check like `if (p == NULL)`, it matches this branch condition against the synthesized predicates. This allows the analyzer to refine the abstract state of `p` on the respective branches, thereby resolving the `UNCHECKED` state and preventing false positives.

By making these implicit state transitions explicit and branch-conditioned, CONTRACT enables efficient reasoning about pointer state across calls, without resorting to full cross-procedural path exploration.

C. Pointer State Transition Contracts Generation

Having formally defined the structure of a PSTC, we now detail how to infer automatically. The core challenge is acquiring semantic knowledge for two distinct types of functions: opaque, external library functions, and user-defined, source-available functions. We term a novel, hybrid inference strategy: it first bootstraps the analysis using an LLM for library functions, then propagates and refines this knowledge using dataflow analysis for user functions.

1) LLM-Powered Library function PSTC Generation: We designed a structured prompting process targeting library functions reachable within the project's call graph. For each of these utilized candidates, we systematically collect a rich set of contextual information, as detailed in Table III, to construct a comprehensive prompt. This information is embedded into a prompt that instructs the LLM to act as a security-aware C expert and translate the function's implicit semantics into our formal semantic structure.

While LLMs are known to occasionally "hallucinate"—producing outputs that are plausible-sounding but factually incorrect. To mitigate this risk and ensure the trustworthiness of our library PSTC, we employ a rigorous, two-tiered validation and refinement strategy. First, we enforce schema-driven validation to guarantee that the LLM's output

TABLE III: Collected Information of Library Functions

Field	Description
`call site context`	Source code that surrounds the function call, illustrating typical usage scenarios like parameter passing.
`doxygen documentation`	Doxygen-style documentation block from the header file if available, including function purpose, behavior, and constraints.
`signature`	The complete function declaration from the header file (including return type, name, and parameter list).
`declaration location`	Full path and line number of the function declaration in its header file.

is structurally correct and adheres to logical consistency rules (e.g., `retval` should not exist in `void` return type). Second, we engineer our prompts to require Chain-of-Thought (CoT) [30] reasoning to help us keep the reliability of LLM [31]. Specifically, our prompts adopt a reasoning-before-output template. We use the PSTC for `malloc` as the one-shot example [32] to guide the LLM to perform stronger reasoning analysis of pointer behaviors in natural language, before generating the final structured PSTC.

2) User-defined function PSTC Generation via Dataflow Analysis: After finishing library PSTC with LLMs, CONTRACT propagates PSTC to user-defined functions using a bottom-up, call-graph-driven approach.

Let the call graph be $G = (V, E)$, where V is the set of functions and E is the set of call edges. We traverse G starting from leaf nodes, moving iteratively to their callers. A function $f \in V$ becomes analyzable only if PSTC have been computed for all functions it invokes (for all $(f, g) \in E$). For each analyzable function f, the goal is to derive its PSTC. We perform the bidirectional intra-procedural dataflow analysis over f's control-flow graph. Since each statement is analyzed a constant number of times. Given that the state update within Transfer_F is an $O(1)$ operation, the total time complexity scales linearly with the number of statements n, effectively $O(\text{LOC})$. The detailed algorithms are presented below.

a) Forward Analysis for Postcondition Inference: The first pass computes the function's postconditions by propagating pointer states forward from its entry point. The process is detailed in Algorithm 1. This is a dependency-driven analysis where a statement is scheduled for analysis only after all its non-loop predecessors have been processed. Its input state, `StateIn`, is computed by joining the exit states of these predecessors. The transfer function, Transfer_F, models the effects of each statement by generating and propagating necessary input states.

The final step of the forward analysis is to construct the postcondition clause, $\mathcal{POST}_f$. A simple merge of all exit states (via the join operator $\sqcup$) would obscure the critical, conditional information required for precise modeling. Instead, for each path leading to a function exit, our algorithm preserves the distinct outcome by extracting a pair:(ϕ, σ), where ϕ is the guard predicate and σ is the corresponding state mapping.

In the context of function contracts, the guard predicate ϕ must strictly focus on observable values—data visible to the caller (the return value) or provided by the caller (arguments). As established in Definition 1, our Access Paths ($\mathcal{AP}$) are explicitly rooted in these interface elements. Consequently, we formalize ϕ not as an arbitrary boolean formula, but as a constraint over these specific access paths. This lightweight structure is expressive enough to precisely model standard error-handling patterns (e.g., `if (ret < 0)`) without the performance overhead of heavyweight SMT solving.

Complementing the guard predicate is the State Mapping (σ), which captures the observable side effects associated

Algorithm 1: Dependency-Driven Forward Dataflow Analysis for Postconditions

Input : Function f with its CFG G_{cfg}
Database $\mathcal{D}$ of callee PSTC
Output : Postcondition clause $\mathcal{POST}_f$

Let s_{entry} be the first statement of f
Let W be a worklist, initialized as $\{s_{entry}\}$
foreach *statement* $s \in V$ **do**
  StateIn$[s] \leftarrow \emptyset$
  PredStates$[s] \leftarrow \emptyset$
  pending_preds$[s] \leftarrow$ NumberOfPredecessors(s)
  analyzed$[s] \leftarrow$ false
StateIn$[s_{entry}] \leftarrow$ initial state from f's parameters
while W *is not empty* **do**
  $s \leftarrow W$.pop()
  if *analyzed*$[s]$ **then**
    continue
  analyzed$[s] \leftarrow$ true
  if s *is a branch statement* **then**
    StateOut[s]$_{true,false} \leftarrow \text{Transfer}_F(\text{StateIn}[s], s, \mathcal{D})$
  else
    StateOut$[s] \leftarrow \text{Transfer}_F(\text{StateIn}[s], s, \mathcal{D})$
  foreach *successor* s_{succ} *of* s *in* G_{cfg} **do**
    if s *is a branch statement* **then**
      Let S_{out} be StateOut$_{true}$ or StateOut$_{false}$ based on the edge (s, s_{succ})
    else
      Let S_{out} be StateOut$[s]$
    PredStates$[s_{succ}]$.add(S_{out})
    pending_preds$[s_{succ}] \leftarrow$ pending_preds$[s_{succ}] - 1$
    if $s_{succ} == n_{exit}$ **then**
      $\phi \leftarrow$ (`True`, –, –)
      foreach *observable* $v \in \{$`retval`, `params`$\}$ **do**
        $C \leftarrow$ GetConstraint(v, S_{out})
        if C *matches vocab* (`op`, `val`) **then**
          $\phi \leftarrow (v,$ `op`, `val`$)$
          break
      $\sigma \leftarrow$ ExtractSideEffects(S_{out})
      $\mathcal{POST}_f$.add$((\phi, \sigma))$
      continue
    if s_{succ} *is a loop header* ***or*** *pending_preds*$[s_{succ}] == 0$ **then**
      newState $\leftarrow \bigsqcup_{S \in \text{PredStates}[s_{succ}]} S$
      StateIn$[s_{succ}] \leftarrow$ newState
      W.push(s_{succ})

with a specific outcome. Formally, σ is a snapshot of the abstract state at the function exit, but filtered to include only relevant access paths. Crucially, this mapping is conditional: it describes the pointer state if and only if the corresponding guard predicate ϕ holds. By aggregating these (ϕ, σ) pairs for all distinct execution paths, we construct a comprehensive, disjunctive postcondition that precisely models the function's state transitions.

b) Backward Analysis for Precondition Inference: The second pass infers the function's preconditions using a backward analysis that is symmetric to the forward pass. This analysis propagates state requirements backward, focusing on statements that have intrinsic pre-conditions (e.g., dereferences `*p`, which requires `p` to be `VALID`). The backward transfer function, Transfer_B, models the requirements of each statement by generating and propagating necessary input states. The analysis proceeds analogously to Algorithm 1 but on the reverse CFG and the meet operation $\sqcap$ if needed. The final precondition, $\mathcal{PRE}_f$, is the set of all requirements that have propagated to the function's entry point. Due to its symmetric nature, we omit the detailed pseudo-code for brevity.

D. PSTC-Guided Path-Sensitive Bug Detection

With a comprehensive database of Pointer State Transition Contracts (PSTCs) inferred, CONTRACT's final phase performs a global, path-sensitive analysis to detect contract violations. Our approach revolutionizes traditional inter-procedural analysis by integrating our formal PSTCs directly into a powerful symbolic execution engine. This transforms opaque function calls from analytical roadblocks into semantically rich checkpoints where contracts are explicitly checked.

This PSTC-guided detection paradigm, illustrated in Figure 2, is the key to both our scalability and precision. Instead of recursively descending into a callee's source code at every call site—a process that leads to an exponential explosion of paths—our checker breaks this dependency. It queries the PSTC Database for the callee's pre-computed contract. This single action transforms the intractable problem of whole-program path exploration into a series of efficient, localized checking tasks performed within the caller's context. Specifically, at each checkpoint, our checker focuses on checking two primary classes of pointer state properties: (1) Pre-condition Violation Check and (2) Post-condition Handling Check, which are detailed in the following sections.

1) Pre-condition Violation Check: The first class of checks ensures that all operations are invoked in a safe state. This involves checking the implicit preconditions of both primitive C operations and entire function calls.

- **For Sensitive AST Operations:** We enforce the intrinsic contracts of fundamental C operations. For instance, any dereferencing operation—be it pointer dereference (`*p`), field access (`p->field`), or array access (`p[i]`)—has an implicit precondition that the pointer p must be in the `VALID` state. Our checker checks this precondition against the pointer's current abstract state. A violation immediately flags a memory safety bug, such as a Null Pointer Dereference or a Use-After-Free.
- **For Function Calls:** The pre-conditions are explicit in the callee's PSTC. Before a call to a function `f`, our checker retrieves `PSTC_f` from the database and validates that the current state of the arguments at the call site satisfies all specified pre-conditions. This allows us to detect a wide range of API misuse bugs, such as passing an uninitialized buffer or a potentially null pointer to a function that expects a valid one.

a) Post-condition Handling Check: The second, more subtle class of checks checks that the caller correctly handles the full spectrum of outcomes advertised by a function's post-conditions. A function's outcome is often communicated through a designated result-bearing variable—this can be the function's return value or, particularly for functions returning `void`, an output parameter (e.g., a pointer-to-a-pointer like `data_t **out`). Our checker checks that the caller's logic correctly accounts for all possible states of this result-bearing variable, as well as any other side effects on pointer arguments.

- **Missing Check on Conditional Outcomes:** If a function call contains the conditional outcomes (e.g., { `(retval EQ NULL, ...), (retval NEQ NULL, ...)` }) in its PSTC, we will mark the variable in an `UNCHECKED` state. When it yields a conditional outcome for a result-bearing variable, such as `p` (the return value or an output parameter), its state is initially represented by our structured `UNCHECKED` form. This `UNCHECKED` state acts as a "taint", signifying that the precise state of `p` is not yet determined. Our path-sensitive checker tracks this uncertainty. If it finds a path where p is consumed by a sensitive operation (e.g., dereferenced `*p`) before an intervening check resolves its state, a "Missing Check" bug is reported. The core violation is using a variable whose state is still ambiguous, breaking the safety assumptions of subsequent operations.
- **Incomplete Handling of Side Effects:** A function's post-conditions can also specify that a non-result-bearing pointer argument is modified, often on an error path. For example, a function `int process_and_free(data_t **p)` might free the pointed-to memory upon failure. Its PSTC would capture this: `(retval NEQ ZERO, *arg0` $\mapsto$ `DANGLING), (retval EQ ZERO, *arg0` $\mapsto$ `VALID)` . Our checker detects if the caller fails to handle this side effect. If the calling code checks for the normal (`ret == 0`) but forgets to check the error one, proceeds to use the original pointer `*p` on that same path, the checker will flag a Use-After-Free. This checks that the caller's logic is robust enough to handle not just the primary outcome, but all state changes advertised in the post-condition.

By systematically performing these two categories of checks at every relevant program point, CONTRACT can precisely pinpoint violations of both explicit (in PSTC) and implicit (in C operations) state required contracts.

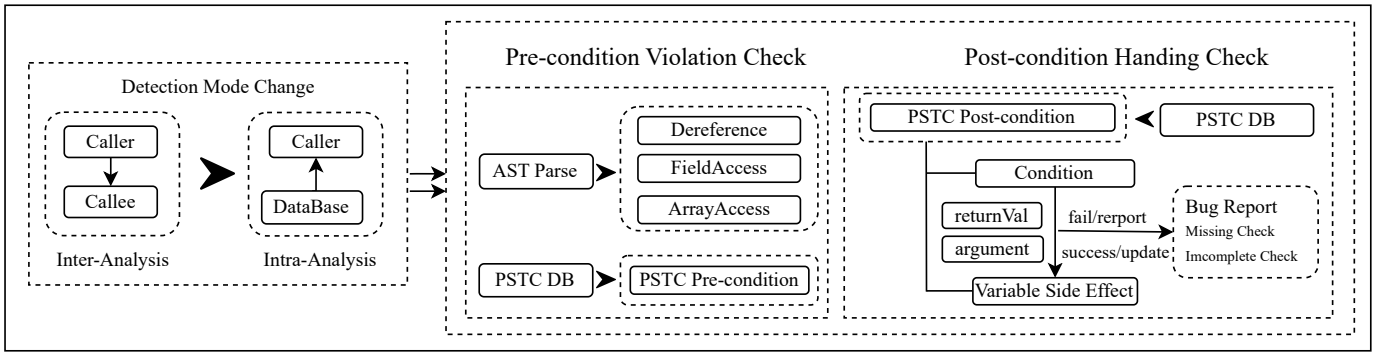

Fig. 2: The PSTC-guided detection paradigm of CONTRACT.

IV. IMPLEMENTATION

We have implemented our approach named CONTRACT, built upon the LLVM/Clang [26] infrastructure with a Python-based orchestration layer. This hybrid design leverages Clang's C++ analysis framework for performance-critical tasks and Python for flexible pipeline management.

CONTRACT's architecture comprises two primary components managed by a central Python orchestrator.

a) C++ Analysis Engine based on Clang: The core engine is a series of Clang-based tools operating on the Abstract Syntax Tree (AST) and control flow graph, which preserves crucial high-level program information. Its key responsibilities include:

- Code Representation: Extracting the program's call graph and per-function CFGs, while also collecting contextual information for library calls.
- Bidirectional Dataflow Analysis: we implements the path-sensitive inference algorithm to generate PSTC.
- Contract-Aware Checker: A custom check integrated into the Clang Static Analyzer that leverages its path-sensitive, symbolic execution engine to check contracts at checkpoints.

b) Python Orchestration: A lightweight Python controller orchestrates the multi-stage analysis pipeline, managing the invocation of C++ tools and the interaction with LLM. To ensure the reproducibility of PSTC inference, we utilize the DeepSeek-V3.2 [33] via its official API with default settings to minimize non-deterministic behavior. The controller constructs each prompt using a structured library context, such as function signatures and Doxygen comments, to guide the model's semantic reasoning. Furthermore, it schedules the bottom-up analysis, triggers the dataflow analysis and final checks via CodeChecker [34], and organizes the resulting bug reports into HTML for visualization.

V. EVALUATION

To comprehensively evaluate our approach, we designed a series of experiments to answer the following key research questions:

- **RQ1**: **Effectiveness of CONTRACT in Bug Detection.** How effective is CONTRACT in detecting real-world bugs, using the inferred PSTC?

TABLE IV: Benchmark Details and Domain Coverage

Project	Brief Description / Domain	Stars
mrubyc [35]	Ruby Interpreter for **Embedded Systems**	400+
fdns [36]	**Network Service** (DNS Server)	140+
coturn [37]	**Network Protocol** (TURN/STUN Server)	13k+
nnn [38]	**System Utility** (Terminal File Manager)	20.9k+
libzip [39]	**File Manipulation Library** (ZIP File)	960+
entr [40]	**Development Tool** (File Watcher)	5.4k+
hiredis [41]	**Database Client Library** (Redis client)	6.6k+

- **RQ2**: **Comparative Analysis with State-of-the-Art Tools.** How does CONTRACT's bug-finding capability compare to other state-of-the-art static analysis tools on the same set of benchmarks?
- **RQ3**: **Precision of Inferred Semantics.** How accurately does CONTRACT's inference engine capture the true state-aware semantics of C functions?

Benchmark. To answer our research questions, we selected a diverse set of widely used, open-source C projects, shown in Table IV. Our selection criteria are designed to cover a range of application areas, and the warehouse is actively maintained.

Platform. All experiments were conducted on a server equipped with an Intel(R) Xeon(R) Gold 5218 CPU@2.30GHz with 64 cores, 128GB RAM, running Ubuntu 20.04 LTS.

A. RQ1: Effectiveness of CONTRACT in Bug Detection.

We ran the complete CONTRACT pipeline on the benchmark. All reported warnings were manually triaged and validated by at least two authors to determine their validity. For bugs deemed authentic, we submitted patches or detailed reports to the developers.

1) Overall Findings: CONTRACT reported a total of **325** warnings in our benchmark suite. Our manual analysis confirmed **209** of these as genuine bugs, resulting in an overall precision of **64.31%** ([209]/[325]).

Table V presents a detailed breakdown of our experimental results for each target project. For every project, we report the total number of generated warnings, categorized into confirmed True Positives (TPs) and False Positives (FPs). Additionally, the table tracks the status of our bug reports (whether they have been confirmed or fixed by developers). To evaluate the accuracy of our tool, we calculate the Precision rate as follows:

TABLE V: Bug Detection Results of CONTRACT Across All Benchmarks.

Project	Total Warnings	True Positives (TPs)	False Positives (FPs)	Precision Rate	Confirmed by Devs	Fixed by Devs
mrubyc	10	9	1	90.0%	7	5
fdns	18	12	6	66.7%	6	6
coturn	216	147	69	68.1%	87	0
nnn	19	10	9	52.6%	4	4
libzip	41	11	30	26.8%	6	6
entr	9	8	1	88.9%	7	7
hiredis	12	12	0	100.0%	10	0
Total	**325**	**209**	**116**	**64.3%**	**127**	**28**

$$\text{Precision} = \frac{\text{TPs}}{\text{Total Warnings}} = \frac{\text{TPs}}{\text{TPs} + \text{FPs}} \quad (1)$$

As the results show, CONTRACT was effective across projects of varying domains. We reported the detected bugs to the developers, and till the time of paper writing, we received confirmations for **127** of our reported bugs, with **28** already fixed by developers and **2** assigned a CVE, demonstrating the real-world impact of our findings. The bugs discovered span several critical categories, including Null Pointer Dereferences, Double Frees, and numerous violations of contracts.

2) Analysis of False Positives: Through a detailed manual audit of reported warnings, we identified three primary technical drivers for False Positives. We categorize these drivers below by their relative prevalence, ranked from the most frequent to the least:

Ownership Escape: This factor occurs when pointer ownership escapes the local function via parameters, global variables, or external calls. This loss of localized ownership and impaired state tracing complicates memory lifecycle tracking, resulting in a notably high false positive rate for memory leak detections.

Path Infeasibility: FPs often arise from "ghost paths"—semantically unreachable execution sequences. These typically stem from unrecognized non-returning functions (e.g., `panic`, `die`), `assert` statements, or imprecise constraint management (e.g., complex bitwise operations). Such limitations prevent the engine from pruning infeasible branches, leading to spurious warnings.

Inaccurate Inference: This originates from the inherent limitations of LLM reasoning, compounded by overlooked syntactic nuances and complex CFG traversal during the inference phase, discussed in RQ3. Failing to capture the full scope of function behaviors leads to the inaccurate PSTC, which misguides the downstream checker.

3) Analysis of False Negatives: To handle unresolved function calls, CONTRACT adopts a precision-oriented optimistic strategy [42] by resetting pointer states to `VALID`. This design choice is motivated by the practical trade-offs in static analysis: while a strictly conservative approach would assume `UNCHECKED` or rikser states, such over-approximation in large-scale inter-procedural analysis invariably triggers prohibitive alarm fatigue and cascading false positives [43]. Apart from these algorithmic considerations, a specific practical limitation of our current implementation resides in the integration layer with the CodeChecker framework. While the report is user-friendly, we observed that it may miss certain warnings captured by a standalone CSA execution. We addressed this by running the standalone analyzer on a subset of files—primarily those that other tools can detect—to uncover missed bugs. However, since many files were not subjected to this redundant check, we acknowledge the inevitability of false negatives. Also, inaccuracies in PSTC inferences also lead to a large part of false positives.

4) Scalability Experiment: CONTRACT analyzed `SQLite` (216k LOC) in 3h 40m, using 5.5 GB peak memory (123% CPU) and detecting one confirmed bug. This covers project-specific inference and checking, while reusing pre-computed standard library PSTC.

B. RQ2: Comparative Analysis with State-of-the-Art Tools

To contextualize CONTRACT's performance, we conducted a comparative analysis against six state-of-the-art static analysis tools, focusing on pointer-related bugs: Clang Static Analyzer [26], a powerful, open-source static analysis framework; Infer [25], an industrial-strength analyzer based on separation logic; Goshawk [44], a specialized tool for memory corruption detection; IPPO [45], a differential checking tool designed to find missing security operations; Semgrep [46], a lightweight, rule-based engine for fast semantic pattern matching; CodeQL [47], a query-driven platform that models code as a relational database to support complex analysis.

We ran all tools on our benchmark suite and manually triaged their reports. The overall results are summarized in Table VI, with a detailed project-wise breakdown provided in Table VII

TABLE VI: Overall Comparison of Bug Detection Capabilities.

Tool	TPs	FPs	Unique Bugs
CONTRACT (Ours)	**209**	**116**	**152**
Clang Static Analyzer	67	26	16
Infer	7	33	0
Goshawk	1	0	0
IPPO	1	76	0
Semgrep	0	33	0
CodeQL	4	4	4

Note: "Unique Bugs" means bugs found ONLY by that tool.

As demonstrated, CONTRACT consistently identifies the highest number of True Positives (**209**) while maintaining a competitive precision-recall balance compared to existing tools. Notably, in complex benchmarks like `libzip`—which is rife with the technical hurdles identified in RQ1, specifically *Ownership Escape* and *Path Infeasibility*—CONTRACT achieves a precision of 26.8%. Although this figure appears modest

TABLE VII: Comprehensive Comparison of Bug Detection Results Across Benchmarks

Tools	mrubyc		fdns		coturn		nnn		libzip		entr		hiredis	
	TP/T	ACC	TP/T	ACC	TP/T	ACC	TP/T	ACC	TP/T	ACC	TP/T	ACC	TP/T	ACC
CONTRACT	**9/10**	90.0%	**12/18**	66.7%	**147/216**	68.1%	**10/19**	52.6%	**11/41**	**26.8%**	**8/9**	88.9%	**12/12**	**100%**
CSA	4/4	**100%**	5/7	71.4%	51/60	85.0%	4/4	**100%**	2/17	11.8%	1/1	**100%**	0/0	0%
Infer	0/9	0%	0/0	0%	3/16	18.8%	0/1	0%	2/10	20.0%	2/4	50.0%	0/0	0%
Goshawk	0/0	0%	0/0	0%	0/0	0%	1/1	**100%**	0/0	0%	0/0	0%	0/0	0%
IPPO	0/7	0%	0/0	0%	0/28	0%	0/1	0%	1/40	2.5%	0/0	0%	0/1	0%
Semgrep	0/0	0%	0/29	0%	0/0	0%	0/0	0%	0/4	0%	0/0	0%	0/0	0%
CodeQL	0/0	0%	3/3	**100%**	1/1	**100%**	0/0	0%	0/0	0%	0/0	0%	0/0	0%

Note: TP/T: True Positives / Total Warnings; ACC: Accuracy.

in absolute terms, it significantly outperforms all baselines, underscoring CONTRACT's superior capability in resolving the intricate pointer logic where traditional tools falter.

More importantly, CONTRACT successfully uncovered **152** unique bugs that remained entirely undetected by all other state-of-the-art tools. The following paragraphs provide a detailed qualitative analysis of the strengths and limitations of each tool relative to CONTRACT.

1) Clang Static Analyzer: CSA is a powerful path-sensitive bug finder that uses symbolic execution. It was effective at finding bugs within individual functions or through simple inter-procedural paths. It successfully identified 67 bugs in our benchmarks.

Limitation: CSA's primary weakness is its monolithic, whole-program analysis approach. When faced with deep call chains or complex control flow, the analysis state grows exponentially, forcing it to abandon paths or make imprecise assumptions.

How CONTRACT Differs: CONTRACT's compositional approach, built on PSTC, avoids this state-space explosion. Our conditional post-conditions allow us to precisely model the different outcomes of a function call without needing to explore its body, thus eliminating the imprecision that led to CSA's false positives. Most of the bugs that we did not check were not in our consideration, and they were basically not problems caused by pointers, but numerical calculations.

2) Infer: Facebook's Infer utilizes separation logic and bi-abduction, making it highly effective at inter-procedural reasoning about pointers and resources. It detected a total of 7 true positives and 33 false positives.

Limitation: While Infer's summary-based bi-abduction is highly scalable, it exhibits a critical blindness toward return values. Its logic is primarily argument-centric, focusing on how arguments are mutated, but it often fails to correlate these side effects with the function's return status. Consequently, it struggles to capture contracts where the outcome is contingent on the return value (e.g., "the operation failed if the function returns `NULL`"). Moreover, our experiments reveal that this scalability comes at the cost of completeness. As function complexity increases, Infer frequently hits internal resource caps and aborts the analysis or discards partial results to prevent state explosion. This dual limitation—semantic blindness and heuristic dropping—causes it to miss bugs in non-trivial C idioms.

How CONTRACT Differs: CONTRACT addresses these deficiencies by grounding its analysis in a unified domain of Access Paths ($\mathcal{AP}$), which treats both return values and arguments as first-class observable entities. Unlike Infer, which decouples return values and abandons analysis under high complexity, our approach explicitly binds them together using lightweight `(Predicate, State)` pairs. This structured abstraction ensures tractability even for complex functions. By projecting complex path constraints onto a simplified predicate vocabulary (defined in Def. 2), we avoid the state explosion that plagues Infer. This allows us to precisely model how a specific return value (e.g., `retval == NULL`) dictates the state of the arguments, thereby covering the full spectrum of the function's interface without sacrificing coverage.

3) Goshawk: Goshawk is a highly precise tool specialized in detecting memory corruptions. Its key innovation is the use of a neural network and dataflow analysis to accurately identify custom memory allocator and deallocator functions. In our benchmarks, it successfully identified 1 double-free bug.

Limitation: Goshawk's impressive precision on memory functions highlights its core limitation: Its analysis is fundamentally asymmetric. It builds powerful, reusable summaries for a small, specific class of functions (allocators/deallocators), but for the vast majority of "regular" user-defined functions that manipulate pointers, it lacks such a compositional mechanism. When analyzing a regular function that is not a memory manager, Goshawk must resort to traditional, monolithic intra-procedural analysis. This prevents it from effectively reasoning about deep, inter-procedural state transitions that span across multiple non-memory-management functions, leading to missed bugs.

How CONTRACT Differs: CONTRACT's approach is fundamentally more general and symmetric. Our PSTC inference is not restricted to any specific class of functions; It is designed to generate a state-aware summary for functions in the program, whether it allocates memory, manipulates data, or performs checks. This universal, compositional approach is why CONTRACT was able to find more bugs. For instance, we detected 2 double-free bugs compared to Goshawk's one in the same repository.

4) IPPO: IPPO is an innovative tool that detects missing security operations by leveraging a "similar paths, different operations" heuristic. It identifies pairs of syntactically similar code paths and reports a potential bug if a security check is present on one path but absent on the other. We selected this tool as a baseline for two primary reasons: first, it targets a similar

class of defects to ours; second, its reported experimental results demonstrate that it outperforms general contract mining approaches [19] [48]. In our benchmarks, IPPO identifies a total of 1 true positive.

Limitation: IPPO's core strength, its differential nature, is also its fundamental limitation. Its analysis is contingent on the existence of a "good" path to compare against. Consequently, it is structurally incapable of detecting several critical bugs:

- It cannot find bugs that exist on all paths, as there is no correct path for comparison.
- It cannot reason about complex state transitions. For example, it does not track the pointer's state changing from `VALID` to `DANGLING` in Use-After-Free. It only detects a missing check, not that the variable's underlying state changed.

How CONTRACT Differs: CONTRACT operates on a fundamentally different, more general principle. Instead of comparing syntactic paths, it checks code against a semantic model of expected behavior.

- Absolute vs Differential Check: CONTRACT performs absolute check against a formal contract. It can detect a bug even if it exists on all paths because the behavior violates the inferred semantics of the function, regardless of what other paths do.
- State-Aware vs Check-Aware: This is the most critical distinction. IPPO is "check-aware": it flags the absence of a check, assuming that a missing check is always a bug. CONTRACT, in contrast, is state-aware: It focuses on the consequences of the missing check. When a check is absent, CONTRACT does not immediately report a bug. Instead, it propagates the pointer's ambiguous state forward—for example, an `UNCHECKED(VALID, NULL)` state. A bug is reported only if this pointer is later consumed by a sensitive operation whose precondition is violated.

5) Semgrep: As a popular lightweight engine for rapid security linting, Semgrep excels at pattern matching.

Limitation: It primarily relies on semantic pattern matching and lacks path-sensitivity, failing to distinguish variable states across different execution branches. This leads to a high rate of false positives in complex pointer-related logic.

How CONTRACT Differs: Unlike Semgrep's syntax-level matching, CONTRACT performs path-sensitive traversal on the CFG. By employing symbolic execution, we accurately track the evolution of variable states, effectively mitigating the imprecision inherent in pattern-based approaches.

6) CodeQL: As an industry-standard, query-based platform, CodeQL offers a robust framework for comprehensive analysis.

Limitation: Its relational database abstraction, while facilitating global queries, often sacrifices the granularity of path-constraint tracking. Furthermore, it heavily relies on expert-defined queries and manual library modeling, focusing predominantly on dataflow and taint analysis patterns.

How CONTRACT Differs: CONTRACT is built directly upon the Clang Static Analyzer, leveraging its native symbolic execution engine for superior low-level constraint management. This enables a more precise characterization of PSTC than relational queries. Moreover, we integrate LLMs to automatically infer implicit contracts, thereby reducing reliance on manual expert modeling.

C. RQ3: Accuracy of Inferred Semantics

The fundamental premise of CONTRACT is that accurately inferred semantics enable precise bug detection. To validate this premise, this RQ assesses the accuracy of our PSTC.

1) Methodology: To establish a ground truth for evaluation, we conducted a manual audit of a statistically significant sample of functions from our benchmarks. We randomly sampled a total of **100** functions, divided into two categories:

- **50 Library Functions**, for which PSTC were generated by our LLM-based approach. This evaluates the quality of our bootstrapping process.
- **50 User-Defined Functions**, for which PSTC were inferred by our dataflow analysis. This evaluates the precision of our inference algorithms.

Two of our authors, acting as expert auditors, manually inspected the source code and documentation for each function. Instead of seeking strict semantic equivalence, we adopted a detection-oriented criterion for correctness. Specifically, we assessed whether the inferred PSTC captured sufficient behavioral details to support accurate bug detection. A PSTC was classified as *correct* if its representation did not induce any false positives or false negatives during the detection phase.

2) Results: The results of our manual audit are summarized in Table VIII. Overall, CONTRACT demonstrated a high degree of accuracy in inferring function semantics.

TABLE VIII: Accuracy of Inferred PSTC s via Manual Audit.

Function Type	Sampled	Correctly Inferred	Accuracy (%)
Library Functions	50	41	82.0%
User-Defined Functions	50	30	60.0%
Overall	**100**	**71**	**71.0%**

We achieved an overall accuracy of **71.0%**. For library functions, the LLM-based approach correctly identified the PSTC for **41** out of 50 functions. For user-defined functions, our dataflow analysis correctly inferred **30** out of 50. This high level of accuracy provides a solid foundation for our downstream bug detection.

3) Discussion: The high detection accuracy is attributed to the synergy of our core design: structured LLM prompting ensures high-quality initial seeds, while the formal PSL model enables rigorous, path-sensitive propagation of these semantics. Crucially, our analysis adopts a precision-oriented (under-approximating) strategy to maximize the *actionability* of its reports [42]. Consequently, most discrepancies manifest as false negatives, occurring when the tool refrains from reporting due to highly intricate or ambiguous logic. The substantial volume of confirmed bugs discovered across mature codebases validates that this pragmatic trade-off does not compromise the tool's practical utility.

Despite its effectiveness, CONTRACT's inaccuracies stem from two primary sources:

a) Inaccuracies in LLM-based Semantic Inference (18%): Inaccuracies in PSTC synthesis are primarily driven by external information scarcity rather than model reasoning capacity. Our experiments show that upgrading from `DeepSeek-Chat` to `DeepSeek-Reasoner` yielded negligible improvements, confirming that the bottleneck resides in the available documentation context.

- **Over-optimistic Parameters (9% of FPs):** When documentation is sparse, the LLM tends to infer over-optimistic states, for instance, misclassifying an argument permitted to be `NULL` as `VALID`, leading to FPs.
- **Incomplete Return Values (9% of FNs):** Conversely, the model may overlook certain error-state returns (e.g., missing a possible `NULL` return). Such omissions cause the checker to bypass valid risky paths, leading to undetected bugs.

b) Inaccuracies in Dataflow Inference (40%): While partially compounded by upstream LLM inference, our code sampling indicates that these inaccuracies are primarily driven by structural limitations within the dataflow engine.

- **Syntactic Resolution Gaps (26%):** Incomplete parsing of complex C constructs, such as nested macro expansions and explicit type casting, hinders the identification of critical operations. These gaps have a bidirectional impact: overlooking a validation check typically induces FPs, while missing a state-altering assignment leads to both FNs and FPs. Empirically, our qualitative analysis reveals that these resolution failures predominantly manifest as FN.
- **CFG Navigation Obstacles (14% of FNs):** Irregular CFG structures, such as multi-exit `switch` blocks and unstructured jumps, hinder precise backward exploration. These constructs introduce complex control-flow join points that complicate path reconstruction, frequently leading to state-tracking failures and incomplete coverage, which primarily results in FNs.

VI. THREATS TO VALIDITY

A significant threat arises from the absence of a ground truth for both the inferred PSTC and the reported bugs. Consequently, verifying the accuracy of our semantic inference requires an exhaustive manual audit process where authors must synthesize disparate documentation and source code context to resolve implicit ambiguities. Similarly, validating the detected inconsistencies necessitates continuous communication with project maintainers to confirm whether the reports represent genuine bugs. While this manual verification is inherently subject to human interpretation, the fact that 127 reports have already been officially confirmed by developers—with 28 fixed and 2 CVEs assigned—empirically validates the effectiveness and practical actionability of CONTRACT 's findings.

In addition, the generalizability of our results is potentially limited by the size and specific selection of our benchmark suite. While these projects were chosen to cover diverse domains such as network protocols and database clients, they may not capture the full syntactic and structural variety of all kinds of C codebases. However, we mitigated this concern by conducting a rigorous comparative evaluation against six industrial and academic state-of-the-art tools. The identification of 152 unique bugs missed by all existing baselines demonstrates that CONTRACT 's semantic inference approach remains robust and provides superior coverage across various software.

VII. RELATED WORKS

Automated API Contract Mining. Existing research infers contracts by mining patterns from available evidence. Statistical and External Methods derive contracts from frequent usage patterns [48], historical patches [20] [28], or documentation [21] [22]. There are also hybrid methods that construct probabilities [49] [50] for detection. Some novel methods utilize program slicing and similarity analysis [19] [45] [51] to infer contracts by propagating patterns across structurally similar code fragments. However, these paradigms face dual limitations. First, they rely on knowledge completeness, struggling with sparse data. Second, they suffer from a loss of conditional semantics, failing to capture the specific conditions—such as distinct return values—that actually trigger these obligations. Unlike them, CONTRACT analyzes different branches by accurately capturing the state of observable variables to achieve higher accuracy.

Static Analysis for Memory and Resource Safety. Traditional path-sensitive analysis [26] [52] offers high precision but is plagued by path explosion, leading to incomplete coverage. To address this, research has pivoted towards sparse value flow analysis [10]. Based on that, many tools transform programs into graphs [53] [54] [55] [56], converting the problem into graph reachability analysis. However, these graph-based methods face a critical dilemma. Rigorous state tracking on graphs still suffers from state explosion when handling complex internal logic. Consequently, they heavily rely on identifying sensitive functions [44] [57] to prune the search space; However, if a wrapper is complex or fails to be identified, the analysis either degenerates into the same bottlenecks as path-sensitive tools. In contrast, CONTRACT is a globally compositional framework. This allows us to reconcile scalability with the semantic richness required.

VIII. CONCLUSION

In this paper, we presented CONTRACT, a compositional static analysis framework for inferring and checking implicit API contracts on pointer state transitions. By modeling pointer state on a risk-monotonic lattice, CONTRACT effectively captures the conditional dependencies over resource states, addressing the precision-scalability dilemma faced by traditional methods. Our evaluation on diverse real-world C projects shows the tool's practical effectiveness: CONTRACT identified 209 previously unknown inconsistencies, with 127 confirmed by developers and 2 CVEs assigned. These findings underscore the prevalence of implicit contract violations in software, highlighting the need to integrate Pointer State Transition Contracts inference into modern quality assurance pipelines.

IX. ACKNOWLEDGMENT

This work was supported by the National Key R&D Program of China No 2024YFB4506200.

REFERENCES

[1] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review*, 35(5):57–72, 2001.

[2] Zuxing Gu, Jiecheng Wu, Jiaxiang Liu, Min Zhou, and Ming Gu. An empirical study on api-misuse bugs in open-source c programs. In *2019 IEEE 43rd annual computer software and applications conference (COMPSAC)*, volume 1, pages 11–20. IEEE, 2019.

[3] Microsoft. "a proactive approach to more secure code". [Online], 2019. Available: https://www.microsoft.com/en-us/msrc/blog/2019/07/a-proactive-approach-to-more-secure-code.

[4] Chromium. "memory safety". [Online], 2025. Available: https://www.chromium.org/Home/chromium-security/memory-safety/.

[5] David Evans. Static detection of dynamic memory errors. *ACM SIGPLAN Notices*, 31(5):44–53, 1996.

[6] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 53–64. IEEE, 2019.

[7] Ruishi Li, Yunfei Yang, Jinghua Liu, Peiwei Hu, and Guozhu Meng. The inconsistency of documentation: a study of online c standard library documents. *Cybersecurity*, 5(1):14, 2022.

[8] Rahul Mohanani, Iflaah Salman, Burak Turhan, Pilar Rodríguez, and Paul Ralph. Cognitive biases in software engineering: A systematic mapping study. *IEEE Transactions on Software Engineering*, 46(12):1318–1339, 2018.

[9] Anders Møller and Michael I Schwartzbach. Static program analysis. *Notes. Feb*, 2012.

[10] 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.

[11] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Phasar: An inter-procedural static analysis framework for c/c++. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 393–410. Springer, 2019.

[12] Min-Yih Hsu, Felicitas Hetzelt, and Michael Franz. Dfi: An interprocedural value-flow analysis framework that scales to large codebases. *arXiv preprint arXiv:2209.02638*, 2022.

[13] Jianjun Huang, Jianglei Nie, Yuanjun Gong, Wei You, Bin Liang, and Pan Bian. Raisin: Identifying rare sensitive functions for bug detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, 2024.

[14] Ziyang Li, Saikat Dutta, and Mayur Naik. Iris: Llm-assisted static analysis for detecting security vulnerabilities. *arXiv preprint arXiv:2405.17238*, 2024.

[15] Shiyin Lin. Llm-driven adaptive source-sink identification and false positive mitigation for static analysis. *arXiv preprint arXiv:2511.04023*, 2025.

[16] Glenn Ammons, Rastislav Bodík, and James R Larus. Mining specifications. *ACM Sigplan Notices*, 37(1):4–16, 2002.

[17] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. *ACM SIGSOFT Software Engineering Notes*, 30(5):306–315, 2005.

[18] Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. Narminer: discovering negative association rules from code for bug detection. In *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 411–422, 2018.

[19] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting {Missing-Check} bugs via semantic-and {Context-Aware} criticalness and constraints inferences. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1769–1786, 2019.

[20] Miaoqian Lin, Kai Chen, and Yang Xiao. Detecting {API}{Post-Handling} bugs using code and description in patches. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3709–3726, 2023.

[21] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 242–253, 2018.

[22] Tao Lv, Ruishi Li, Yi Yang, Kai Chen, Xiaojing Liao, XiaoFeng Wang, Peiwei Hu, and Luyi Xing. Rtfm! automatic assumption discovery and verification derivation from library document for api misuse detection. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 1837–1852, 2020.

[23] CONTRACT. "artifact of contract". [Online], 2026. Available: https://doi.org/10.5281/zenodo.19248720.

[24] DRM. Drm: a subsystem of the linux kernel responsible for interfacing with gpus. Online, 2025. Available: https://docs.kernel.org/gpu/introduction.html.

[25] facebook. "infer: a tool to detect bugs in java and c/c++/objective-c code before it ships". [Online], 2025. Available: https://fbinfer.com/.

[26] Clang Static Analyzer. clang static analyzer: a source code analysis tool that finds bugs. Online, 2025. Available: https://clang.llvm.org/docs/ClangStaticAnalyzer.html.

[27] Dinghao Liu, Zhipeng Lu, Shouling Ji, Kangjie Lu, Jianhai Chen, Zhenguang Liu, Dexin Liu, Renyi Cai, and Qinming He. Detecting kernel memory bugs through inconsistent memory management intention inferences. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4069–4086, 2024.

[28] Jinghua Liu, Yi Yang, Kai Chen, and Miaoqian Lin. Generating API parameter security rules with LLM for API misuse detection. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025.

[29] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[30] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[31] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM computing surveys*, 55(12):1–38, 2023.

[32] Toufique Ahmed and Premkumar Devanbu. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM international conference on automated software engineering*, pages 1–5, 2022.

[33] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[34] Gabor Horvath, Reka Kovacs, Richard Szalay, Zoltan Porkolab, Gyorgy Orban, and Daniel Krupp. Static code analysis with codechecker. *arXiv preprint arXiv:2408.02220*, 2024.

[35] mrubyc. "mrubyc: another implementation of mruby". [Online], 2025. Available: https://github.com/mrubyc/mrubyc.

[36] fdns. "fdns: a firejail dns-over-https proxy server". [Online], 2025. Available: https://github.com/netblue30/fdns.

[37] coturn. "coturn: a turn server project". [Online], 2025. Available: https://github.com/coturn/coturn.

[38] nnn. "nnn: an unorthodox terminal file manager". [Online], 2025. Available: https://github.com/jarun/nnn.

[39] libzip. "libzip: a c library for reading, creating, and modifying zip archives.". [Online], 2025. Available: https://github.com/nih-at/libzip.

[40] entr. "entr: a tool to run arbitrary commands when files change". [Online], 2025. Available: https://github.com/eradman/entr.

[41] redis. "hiredis: a minimalistic c client for redis >= 1.2.". [Online], 2025. Available: https://github.com/redis/hiredis/.

[42] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[43] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[44] Yunlong Lyu, Yi Fang, Yiwei Zhang, Qibin Sun, Siqi Ma, Elisa Bertino, Kangjie Lu, and Juanru Li. Goshawk: Hunting memory corruptions via structure-aware and object-centric memory operation synopsis. In *2022*

IEEE Symposium on Security and Privacy (SP), pages 2096–2113. IEEE, 2022.

[45] Dinghao Liu, Qiushi Wu, Shouling Ji, Kangjie Lu, Zhenguang Liu, Jianhai Chen, and Qinming He. Detecting missed security operations through differential checking of object-based similar paths. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1627–1644, 2021.

[46] semgrep. "semgrep: a lightweight static analysis tool for many languages". [Online], 2026. Available: https://github.com/semgrep/semgrep.

[47] Pavel Avgustinov, Oege De Moor, Michael Peyton Jones, and Max Schäfer. Ql: Object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, pages 2–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016.

[48] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. {APISan}: Sanitizing {API} usages through semantic {Cross-Checking}. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 363–378, 2016.

[49] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian specification learning for finding api usage errors. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 151–162, 2017.

[50] Yunlong Ma, Wentong Tian, Xiang Gao, Hailong Sun, and Li Li. Api misuse detection via probabilistic graphical model. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 88–99, 2024.

[51] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 361–377, 2015.

[52] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[53] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE symposium on security and privacy*, pages 590–604. IEEE, 2014.

[54] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 693–706, 2018.

[55] Xingjing Deng, Zhengyao Liu, Xitong Zhong, Shuo Hong, Yixin Yang, Xiang Gao, Xuhui Yan, and Hailong Sun. Code property graph meets typestate: A scalable framework to behavioral bug detection. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2025, Auckland, New Zealand, September 7-12, 2025*, pages 1–12. IEEE, 2025.

[56] Xiaoheng Xie, Gang Fan, Xiaojun Lin, Ang Zhou, Shijie Li, Xunjin Zheng, Yinan Liang, Yu Zhang, Na Yu, Haokun Li, et al. Codefuse-query: A data-centric static code analysis system for large-scale organizations. *arXiv preprint arXiv:2401.01571*, 2024.

[57] Chong Wang, Jianan Liu, Xin Peng, Yang Liu, and Yiling Lou. Boosting static resource leak detection via llm-based resource-oriented intention inference. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 668–668. IEEE Computer Society, 2025.