SolQDebug: Debug Solidity Quickly for Interactive Immediacy in Smart Contract Development

Inseong Jeon[1], Sundeuk Kim[1], Hyunwoo Kim[1], Hoh Peter In[1*]

[1*]Department of Computer Science, Korea University, 145, Anam-ro, Seonbuk-gu, 02841, Seoul, Republic of Korea.

*Corresponding author(s). E-mail(s): hoh_in@korea.ac.kr;
Contributing authors: iwwyou@korea.ac.kr; sd_kim@korea.ac.kr;
khw0809@korea.ac.kr;

Abstract

As Solidity becomes the dominant language for blockchain smart contracts, efficient debugging grows increasingly critical. However, current Solidity debugging remains inefficient: developers must compile, deploy, set up transactions, and step through execution line-by-line to examine each variable. This process is too slow for practical use. To address this challenge, we present SolQDebug, the first interactive source-level debugger for Solidity that delivers millisecond feedback directly on source code. Developers specify input value ranges through annotations and compare them against abstract interpretation results, thereby enabling exploration of contract behavior across multiple execution paths. We evaluate SolQDebug on 30 real-world functions from DAppSCAN, achieving 350× faster debugging (0.15s vs. 53s per function) than Remix IDE. Our evaluation provides practical debugging insights: overlapping annotation patterns improve precision in most Solidity debugging scenarios, while analysis of diverse loop patterns demonstrates improved convergence while preserving soundness guarantees. These results demonstrate that SolQDebug makes interactive debugging practical for Solidity development.

Keywords: Smart Contract Development, Solidity, Debugging, Abstract Interpretation

1 Introduction

Smart contracts are the backbone of decentralized applications, and Solidity has become the dominant language for writing them (Chen et al., 2025; Solidity, 2025). As

contracts grow more complex and control more assets, developers must reason about correctness throughout the development cycle—not just at deployment. Large language models (LLMs) such as ChatGPT (ChatGPT, 2025) or Llama (Llama, 2025) can assist with code generation but offer no guarantees of correctness. Ultimately, developers remain responsible for understanding variable interactions, control flow, and numeric boundaries during authoring.

Unfortunately, the debugging workflow for Solidity lags far behind traditional programming environments. Even a single inspection requires full compilation, deployment, transaction-based state setup, and manual bytecode-level tracing. Tools like Remix IDE (Remix IDE, 2025), Hardhat (Hardhat, 2025), and Foundry Forge (Foundry Forge, 2025) replicate this costly pipeline, providing no live feedback during edits. A prior study found that 88.8% of Solidity developers described debugging as painful, and 69% attributed this to the absence of interactive, source-level tooling (Zou et al., 2019). Despite this widely acknowledged pain point, we find no existing research or tooling that provides interactive feedback during Solidity code authoring—a gap that this paper aims to fill.

This paper presents SolQDebug, a source-level interactive Solidity debugger powered by abstract interpretation (AI). Rather than replacing runtime debuggers, it complements them by enabling symbolic, per-statement inspection during code authoring—before compilation or deployment. It targets the Solidity pattern of single-contract, single-transaction execution, where each function is isolated and stateless—ideal for static reasoning but difficult to simulate manually. To support this, SolQDebug applies interval-based AI, which generalizes over symbolic inputs, exposes edge-case behaviors, and provides sound results with low overhead. This approach gives developers immediate feedback and enables them to reason efficiently about how symbolic inputs influence variable behavior. Although these inputs enable generalization across multiple cases, certain input configurations or control structures may lead to wider output ranges. We evaluate these behaviors empirically and propose annotation strategies that help maintain interpretability across typical Solidity patterns.

To achieve this goal, SolQDebug builds on two core ideas. First, it extends the Solidity grammar with interactive parsing rules and dynamically updates the control-flow graph to reflect incremental edits, enabling keystroke-level structural changes during code authoring. Second, it performs AI seeded by inline annotations. These annotations, written directly in the source code, allow developers to specify symbolic values for both parameters and storage variables, similar to how traditional debuggers let users configure initial states and explore control flow.

We evaluate SolQDebug on real-world functions from DAppSCAN (Zheng et al., 2024), demonstrating millisecond-scale responsiveness under symbolic input. Beyond latency, we analyze how input interval structure affects interpretability in common Solidity patterns, such as division-normalized arithmetic.

This paper makes the following contributions:

- We identify the main barriers to interactive Solidity debugging: latency from compilation, deployment, and transaction setup, and EVM constraints that prevent lightweight re-execution.

- We design an interactive parser with seven specialized entry rules and a dynamic control-flow graph (CFG) engine that supports live structural updates and syntactic recovery during incremental editing.
- We introduce an annotation-guided abstract interpreter with adaptive widening thresholds that uses developer-specified symbolic inputs to achieve precise loop analysis while maintaining termination guarantees.
- We evaluate SolQDebug on 30 real-world functions from DAppSCAN, demonstrating 350× median speedup over Remix IDE, and analyze how annotation structure impacts precision in multiplication-heavy arithmetic and loop convergence patterns.

2 Background

2.1 Structure of Solidity Smart Contract

Solidity smart contracts may declare contracts, interfaces, and libraries. Executable business logic typically resides in contracts, and functions serve as transaction entry points. Variables are usefully grouped as global (EVM metadata such as msg.sender or block. timestamp), state (persistent storage owned by a contract), and local (scoped to a call). Types include fixed-width integers, address, booleans, byte arrays, and user-defined structs; containers include arrays and mappings. A mapping behaves like an associative array with an implicit zero value for unseen keys and is not directly iterable. Storage classes (storage, memory, calldata) indicate lifetime and mutability; we mention them only to fix terminology. Visibility and mutability qualifiers (public, external, internal, private; pure, view, payable) exist but are not central to our single-contract, single-transaction setting. Control flow (if/else, while/for/do-while, break/continue, return) follows C/Java conventions.

Listing 1: Minimal example used to illustrate grammar elements relevant to our analysis

```
contract Example {
    address public owner;
    uint256 public totalSupply = 1000;
    mapping(address => uint256) private balances;

    modifier onlyOwner() {
        require(msg.sender == owner, "not owner");
        _;
    }

    function burn(uint256 amount) public onlyOwner {
        uint256 bal = balances[msg.sender];
        uint256 delta;
        if (bal >= amount) {
            balances[msg.sender] = bal - amount;
            delta = amount;
        }
        else {
```

```
19                delta = 0;
20            }
21            totalSupply -= delta;
22        }
23    }
```

The example highlights the specific features we rely on later. State variables include general types (owner, totalSupply) and a mapping from addresses to balances; global variables appear implicitly in guards via msg.sender. The function burn introduces parameters and a local variable (bal). The modifier onlyOwner performs a precondition check before the function body executes; the placeholder underscore marks where the original body is inserted when the modifier is inlined. In analysis, such modifiers are expanded at their precise positions around the function body in the control-flow graph.

These grammar elements connect directly to our semantics. Guards such as require narrow feasible ranges along taken branches. Modifiers are inlined so that their precondition checks are analyzed in sequence with the function body. Containers like mappings remain symbolic until a concrete key is accessed, at which point an abstract value is materialized for that access. This level of detail suffices for our AI in the single-contract, single-transaction scope without introducing parts of the language that our evaluation does not exercise.

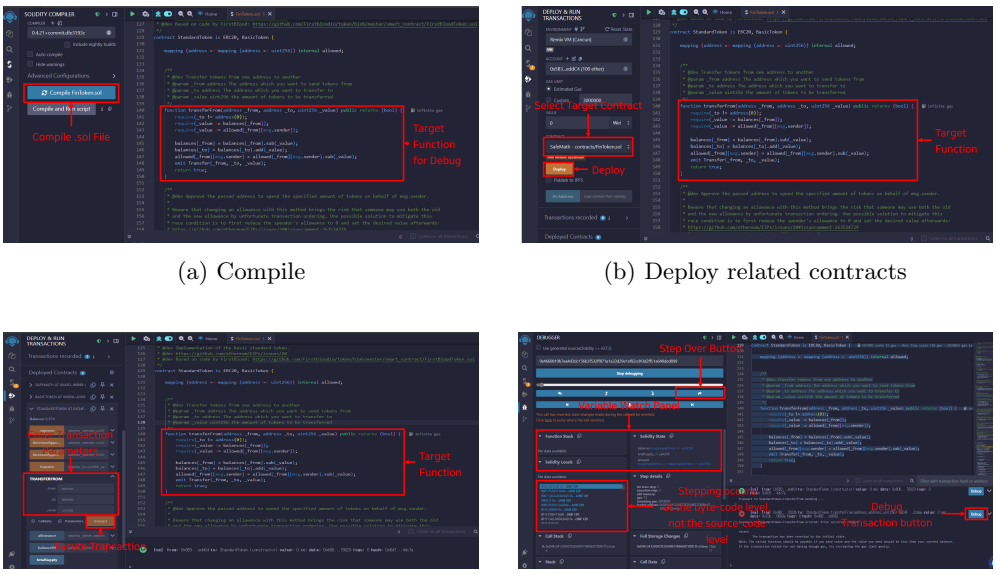
2.2 Solidity Execution Model

To execute a Solidity contract on the blockchain, it must first be deployed. Deployment occurs through a one-time transaction that stores the compiled bytecode on-chain and invokes the constructor exactly once. After deployment, all subsequent interactions are message-call transactions. In these, the caller specifies a public function along with encoded calldata. Once the transaction is mined into a block, the Ethereum Virtual Machine (EVM) jumps to the designated entry point and executes the corresponding function sequentially. At runtime, Solidity variables fall into three distinct storage classes (Solidity, 2025):

- **Global variables** represent implicit, read-only metadata provided by the EVM, such as `block.timestamp`, `msg.sender`, and `msg.value`.
- **State variables** store persistent data within the contract and retain their values across transactions.
- **Local variables** include function parameters and temporary values scoped to a single execution context.

These three classes share a unified type system comprising primitive types like `uint` , `int`, `bool`, and `address`, as well as composite types such as arrays, mappings, and structs. Composite values can be nested to arbitrary depth using field access (`.`) or indexing (`[]`). Control flow follows familiar C-style constructs such as `if/else`, `while`, `for`, and `return`, alongside Solidity-specific statements like `emit` and `revert`.

As a result, debuggers must resolve potentially complex, multi-step expressions to analyze deeply nested elements within the contract state.



(a) Compile

(b) Deploy related contracts

(c) Send preparatory transactions

(d) Bytecode-level debugging

Fig. 1: Traditional Solidity debugging workflow

2.3 Root Causes of the Solidity-Debugging Bottleneck

Debugging Solidity programs remains significantly slower than traditional application development workflows due to two orthogonal obstacles.

(1) Environmental disconnect. Unlike conventional IDEs such as PyCharm (JetBrains, 2025) or Visual Studio (Microsoft, 2025), where the source editor and execution engine run in the same process, Solidity development involves external coordination with a blockchain node at every stage of the workflow. Even a single debugging cycle must pass through four sequential stages (see Fig. 1). First, the contract must be compiled. Then, the bytecode is deployed to a local or test chain. Next, developers must manually initialize the on-chain state by sending setup transactions. Finally, the target function is invoked, and its execution is traced step by step at the bytecode level.

This workflow introduces several seconds to minutes of latency per iteration, fundamentally breaking the fast "type-and-inspect" feedback cycle expected in modern development tools. To mitigate this friction, developers often rely on `emit` logs or event outputs to observe intermediate values. However, such instrumentation provides only runtime snapshots and lacks the structural insight needed to understand symbolic variation or control-flow behavior. Moreover, modifying the expression of interest typically requires recompilation and redeployment, compounding latency and disrupting iteration. The final stage—tracing raw EVM opcodes—is particularly costly, as developers are forced to mentally reconstruct source-level semantics. This not only adds execution overhead but also imposes significant cognitive burden during fault localization and fix validation.

(2) Architectural limitations of the EVM. The Ethereum Virtual Machine (EVM) is a state-based execution engine in which each transaction mutates a globally persistent storage. Once a function executes, its side effects are irreversible unless external intervention is performed. Re-executing the same function along the same control path is nontrivial: developers must either redeploy the entire contract to restore the initial state, or manually reconstruct the required preconditions via preparatory transactions—both of which incur significant overhead.

Additionally, if a function includes conditional guards that depend on the current state—such as account balances or counters—then any debugging session must first ensure that those conditions are satisfied. Fig. 2 illustrates this challenge: the debug target function enforces a check on `_balances[account]`, requiring developers to manually assign a sufficient balance before they can observe the downstream effects on `_totalSupply`. Without such setup, the function exits early, preventing inspection of the intended execution path.

In short, these constraints make repeated debugging iterations costly and fragile. According to a developer study (Zou et al., 2019), 88.8% of Solidity practitioners reported frustration with current debugging workflows, with 69% attributing this to the lack of interactive, state-aware tooling.

2.4 Proposed Methodology and Technical Challenges

SolQDebug addresses the two root causes of Solidity's debugging bottleneck—external latency from blockchain round trips, and internal opacity due to storage-based semantics—through a pair of lightweight but complementary techniques.

(1) Eliminating blockchain latency via in-editor interpretation. The traditional debugging workflow requires compilation, deployment, transaction-based state setup, and bytecode tracing—each incurring significant latency. SolQDebug replaces this round trip by performing both parsing and AI directly inside the Solidity Editor. To support live editing, we extend the Solidity grammar with interactive parsing rules tailored for isolated statements, expressions, and control-flow blocks. When the developer types or edits code, only the affected region is reparsed using a reduced grammar.

Each parsed statement is inserted into a dynamic control-flow graph (CFG), and AI resumes from the edit point. The interpreter uses an interval lattice, assigning each variable a conservative range $[l, h]$ to expose edge conditions (e.g., overflows or failing guards) and to approximate groups of concrete executions that follow the same path. This enables millisecond-scale feedback on code structure and control flow without compilation or chain interaction.

(2) Re-instantiating symbolic state without redeployment. The EVM does not support reverting to a prior state without redeploying the contract or replaying transactions—both of which disrupt iteration. SolQDebug introduces batch annotations as a lightweight mechanism for symbolic state injection. In essence, this reflects a core debugging activity: varying inputs or contract state to observe control-flow outcomes. Rather than reconstructing such conditions through live transactions, developers can write annotations at the top of the function to define initial abstract values.

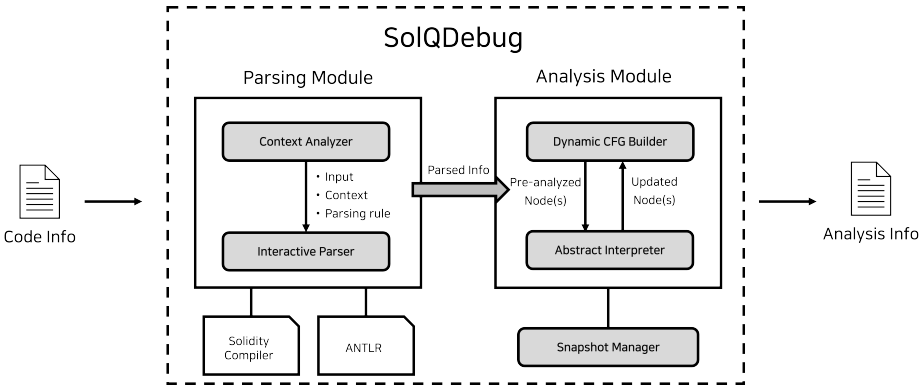


Fig. 2: SolQDebug architecture

These values are injected before analysis begins and rolled back afterward, ensuring test-case isolation.

This approach brings the debugging workflow closer to the source by making state manipulation explicit and reproducible within the code itself. Developers can explore alternative execution paths by editing annotations alone—without modifying the contract logic or incurring compilation and deployment overhead. It effectively decouples symbolic input configuration from the analysis cycle, while preserving the intuitive debugging process developers already follow.

3 The design of SolQDebug

SolQDebug processes code incrementally as developers write it, building up an AI of the program. The system operates as follows. First, each incoming statement or annotation is interpreted under abstract semantics. Second, the corresponding construct is stored in a CFG node that is inserted at a semantically valid location, determined from the surrounding context and the existing control flow. Third, when batch debug annotations are present, the system reinterprets the function with the annotated values. The following subsections describe the architecture and core mechanisms that enable this incremental analysis.

3.1 System Architecture

The system accepts either single Solidity statements or batch debug annotations as input. These inputs are processed through two main modules:

(1) Parsing Module. Each incoming edit passes through the *Context Analyzer*, which extracts the surrounding source context needed to parse the partial statement or annotation. The *Interactive Parser*, built on ANTLR (ANTLR, 2025), applies an extended grammar that adds seven reduction rules to the standard Solidity grammar, enabling it to parse partial constructs that would normally fail to compile. Although the extended grammar can parse partial constructs, the system still validates the complete reconstructed source using the official Solidity compiler before proceeding

Table 1: Incremental inputs for the running example

Step	Lines of Input Fragment	Fragment
1	11--12	`function burn(uint256 amount) public onlyOwner {` `}`
2	12	`uint256 bal = balances[msg.sender];`
3	13	`uint256 delta;`
4	14--15	`if (bal >= amount) {` `}`
5	15	`balances[msg.sender] = bal - amount;`
6	16	`delta = amount;`
7	18--19	`else {` `}`
8	19	`delta = 0;`
9	21	`totalSupply -= delta; // new input`

to analysis. This validation ensures semantic consistency and rejects malformed input early.

(2) Analysis Module. The Analysis Module operates through three coordinated components. The *Dynamic CFG Builder* maintains an incremental control-flow graph that is updated as new statements are added: it creates corresponding nodes for each statement and rewires control edges to reflect the updated program structure. The *Abstract Interpreter* incrementally analyzes the updated CFG, reusing previous results and computing abstract values only for affected program points using a combination of interval and set domains. The *Snapshot Manager* ensures that each debug annotation execution starts from a clean state by preserving and restoring the abstract memory, allowing annotations to be modified and re-executed without side effects from previous runs.

(3) Line-Level Output. Following analysis, the system produces a per-statement summary showing the computed intervals for variables affected by each statement—including declarations, assignments, and return values. All outputs are mapped to their corresponding source line numbers and displayed inline within the editor, providing immediate feedback as developers write and modify code.

3.2 Running Example

To illustrate how the proposed architecture functions in practice, we present a concrete example using the `burn` function from Listing 1. This example demonstrates two key analysis modes: incremental edits (§3.2.1) and batch annotations (§3.2.2).

3.2.1 Incremental Source Code Analysis

As shown in Table 1, the developer incrementally constructs the `burn` function through nine distinct input steps, each introducing a new code fragment. SolQDebug accepts two kinds of fragments:

- **Block fragments** such as function headers or if/else blocks. When the developer types an opening '{', most editors auto-insert the closing '}', so the complete block arrives at once and may span multiple lines (e.g., Step 1 in lines 11–12 of Listing 1).

- **Single statements** ending with semicolons (e.g., Steps 2, 3, 5, 6, 8, and 9).

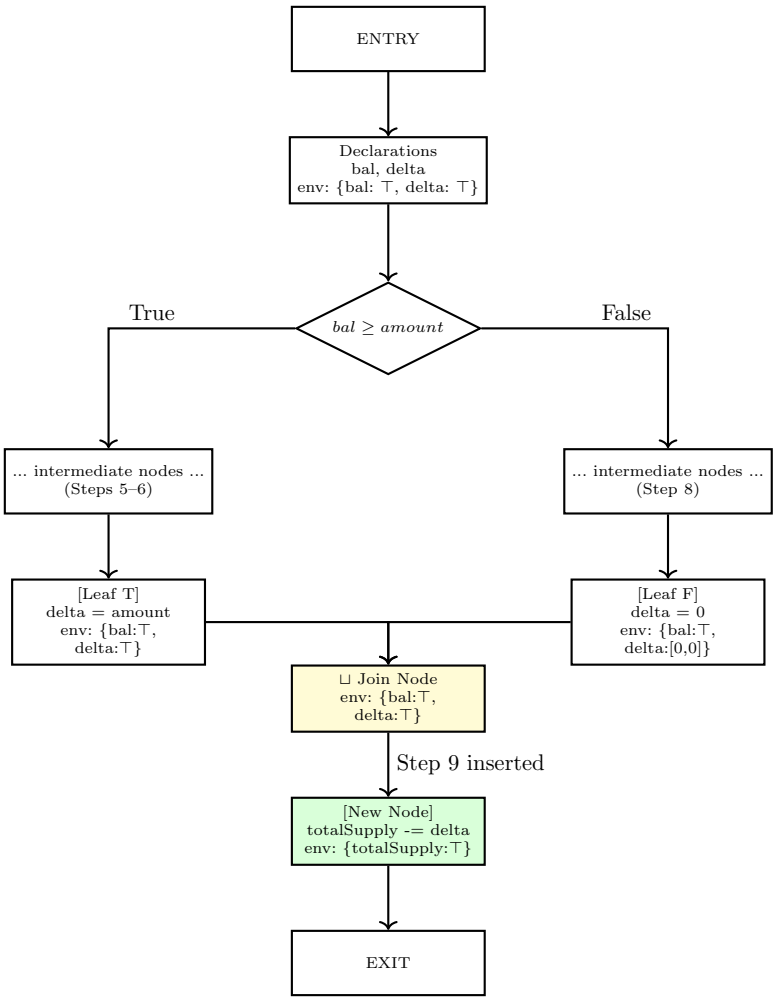


Fig. 3: CFG structure showing Step 9 insertion. Each statement occupies a separate basic node; intermediate nodes along each branch are omitted, showing only the leaf nodes before the join point. The join point node computes the least upper bound of environments from both branches

As the developer types each fragment, SolQDebug incrementally extends the CFG and recomputes abstract values only for affected program points. Figure 3 visualizes the CFG structure after Steps 1–8 have been integrated. We focus on Step 9 (`totalSupply -= delta;`), which illustrates how the system handles CFG insertion after a conditional branch merge. When Step 9 arrives, SolQDebug processes it as follows:

1. The interactive parser recognizes `totalSupply -= delta;` as an assignment.
2. SolQDebug determines the insertion point by examining the edit context and existing CFG. In this case, the insertion point is after the join node that merges the if/else branches.
3. A new CFG node is created for the assignment, and edges are rewired: the join node now flows into this new node, which in turn connects to the exit.
4. The new node receives the environment from the join node, which holds the least upper bound ($\sqcup$) of environments from both branches.
5. SolQDebug reinterprets the new node and all reachable nodes to propagate the updated environment throughout the CFG.

This reinterpretation maintains soundness: it ensures that all affected nodes reflect the updated environment, allowing subsequent edits to directly reuse the computed abstract values without re-analyzing the entire program.

3.2.2 Batch Annotation Analysis

While incremental analysis supports the write-compile-debug cycle, developers often need to explore how different input ranges affect program behavior. This includes verifying price calculations in decentralized exchanges, balance constraints in token transfers, or liquidity ratios in automated market makers. Batch annotations enable this by letting developers specify initial states declaratively and obtain line-level results in a single analysis pass, reusing the CFG constructed during incremental edits.

Listing 2 shows the `burn` function with batch annotations. Annotation blocks are enclosed by `//@Debugging BEGIN` and `//@Debugging END`. Each annotation line specifies a variable type (`@StateVar` for state variables, `@LocalVar` for local variables) and assigns an interval value—supporting both simple variables and nested accesses like `balances[msg.sender]`.

Listing 2: Burn function with batch annotations

```
function burn(uint256 amount) public onlyOwner {
    // @Debugging BEGIN
    // @StateVar balances[msg.sender] = [100,200]
    // @LocalVar amount = [50,150]
    // @Debugging END
    uint256 bal = balances[msg.sender];
    uint256 delta;
    if (bal >= amount) {
        balances[msg.sender] = bal - amount;
        delta = amount;
    }
    else {
        delta = 0;
    }
    totalSupply -= delta;
}
```

Table 2: Interactive parser entry rules

Entry Rule	Purpose
interactiveSourceUnit	Top-level declarations: functions, contracts, interfaces, libraries, state variables, pragmas, imports
interactiveEnumUnit	Enum member items added after the enum shell is defined
interactiveStructUnit	Struct member declarations added after the struct shell is defined
interactiveBlockUnit	Statements and control-flow skeletons inside function bodies
interactiveDoWhileUnit	The while tail of a do-while loop
interactiveIfElseUnit	else or else-if branches following an if statement
interactiveCatchClauseUnit	catch clauses following a try statement

In this example, we annotate `balances[msg.sender]` with the interval $[100, 200]$ and `amount` with $[50, 150]$ to explore how the `burn` function behaves under different balance and amount scenarios.

When a batch annotation block is encountered, SolQDebug follows a lightweight pipeline:

1. **Parse and validate.** Each annotation line is parsed, type-checked, and converted to the corresponding abstract domain (e.g., intervals for integers).
2. **Snapshot and overlay.** The current abstract memory is saved, and the annotated values are overlaid onto the initial environment.
3. **Single-pass analysis.** SolQDebug re-analyzes the pre-built CFG in a single pass using the annotated values as the initial environment.
4. **Restore snapshot.** After analysis completes, the snapshot is restored to isolate successive annotation runs.

Unlike incremental analysis, batch annotations leave the CFG structure unchanged—only the initial environment differs. This makes batch runs lightweight, enabling rapid what-if exploration. Variables without annotations remain at $\top$, making explicit initialization essential for meaningful results. Formal details of the interactive parser and CFG construction appear in §3.3 and §3.4.

3.3 Interactive Parser

The Interactive Parser extends the official Solidity language grammar (Solidity Language Grammar, 2025) with specialized entry rules that accept partial code fragments during incremental editing. The parser defines eight specialized entry rules: seven for partial Solidity constructs during incremental editing, and one for batch-annotation blocks that enable symbolic input scenarios.

Table 2 shows the seven rules, divided into two categories. *Primary rules* (`interactiveSourceUnit`, `interactiveBlockUnit`) handle independent constructs, while *continuation rules* complete partially-written structures by filling enum or struct shells or by appending control-flow branches. This separation prevents syntactically

invalid constructs (e.g., an else-branch without a preceding if-statement) from being parsed as independent statements.

For concreteness, we refer to the burn function in Listing 1. The function header triggers `interactiveSourceUnit`, creating a function with an empty body. Each new statement invokes `interactiveBlockUnit`, which includes productions for both complete statements and control-flow skeletons (see Appendix A for the complete grammar hierarchy). For instance, typing `if (condition) { }` produces a skeletal if-statement. In the burn function, when the developer adds the `else` branch, `interactiveIfElseUnit` attaches it to the existing if-statement. This skeleton-based approach allows incremental construction, one construct at a time, without requiring syntactic completeness.

Beyond these seven interactive rules for Solidity constructs, the parser includes a specialized `debugUnit` rule for testing scenarios. The debugUnit rule parses batch-annotation lines that specify initial abstract values for variables, enabling symbolic input scenarios without contract deployment. The grammar defines three annotation types:

- GlobalVar assigns values to global variables such as msg.sender or block.timestamp
- StateVar assigns values to contract state variables, supporting nested access patterns like balances[msg.sender] or user.balance
- LocalVar assigns values to function parameters and local variables

Each annotation accepts an L-value and a value specification. Supported value formats include integer intervals, symbolic addresses, boolean values, and symbolic placeholders for bytes and strings. The parser validates type compatibility and range bounds at parse time, warning developers if annotated values are incompatible with declared types.

The annotation syntax and validation rules are specified in Appendix A, with the full ANTLR4 implementation available at (SolQDebug Language Grammar Rule, 2025).

3.4 Dynamic CFG Construction

Dynamic CFG construction maintains the control-flow graph incrementally as developers insert new statements. Rather than rebuilding from scratch, our approach modifies the graph in place. We proceed in three steps. First, we construct and splice a CFG fragment for each statement form. Second, we locate where to insert it in the existing graph. Third, we re-interpret only the affected region to update abstract environments.

Our CFG consists of the following node types:

- ENTRY NODE: The unique function entry point where execution begins.
- BASIC NODE: Holds exactly one statement (e.g., a variable declaration, an assignment, or a function call).
- CONDITION NODE: Represents branching constructs such as `if`, `else if`, `while`, `require`/`assert`, and `try`.
- JOIN NODE: Merges control flow from multiple branches (e.g., IF JOIN, ELSE-IF JOIN).

- FIXPOINT EVALUATION NODE (ϕ): The loop join point used for widening and narrowing during fixpoint computation.
- LOOP EXIT NODE: The false branch that exits a loop when the guard condition fails.
- RETURN NODE: A statement node whose outgoing edge is immediately rewired to the function's unique RETURN EXIT.
- ERROR EXIT: The function's unique exceptional exit (targets the exceptional path via `revert`, `require`, or `assert` failures).
- EXIT NODE: The function's unique normal exit point where execution terminates successfully.

3.4.1 Statement-Local, Incremental Construction

Every insertion operates at the CURRENT NODE without restructuring the rest of the graph. To enable direct insertion, each basic node holds exactly one statement. SOLQDEBUG supports all Solidity statements; we present representative examples below.

Assignments, function calls, and unary operations create a single BASIC NODE inserted between the current node and its successors (Figure 4). An `if` statement creates a CONDITION NODE, true/false BASIC NODES, and an IF JOIN (Figure 5). An `else if` replaces the previous false branch with a new condition and its own join, connecting to the outer IF JOIN (Figure 6). An `else` attaches directly to the false branch without creating a new condition node (Figure 7).

A `while` loop creates a FIXPOINT EVALUATION NODE ϕ, a CONDITION NODE, a loop body node, and a LOOP EXIT NODE. The body connects back to ϕ for fixpoint iteration (Figure 8).

A `break` redirects its outgoing edge to the LOOP EXIT NODE (Figure 9). A `continue` redirects to the loop's ϕ node (Figure 10). A `return` is immediately rewired to the function's unique RETURN EXIT, detaching its original successors (Figure 11). A `require` statement creates a CONDITION NODE with the true edge connecting to a continuation node and the false edge pointing to the ERROR EXIT (Figure 12).

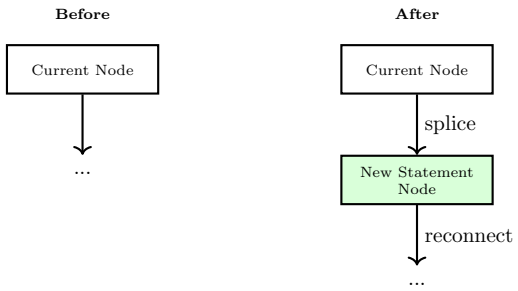


Fig. 4: Simple statement insertion. The builder creates one node and splices it between the current node and the original successors

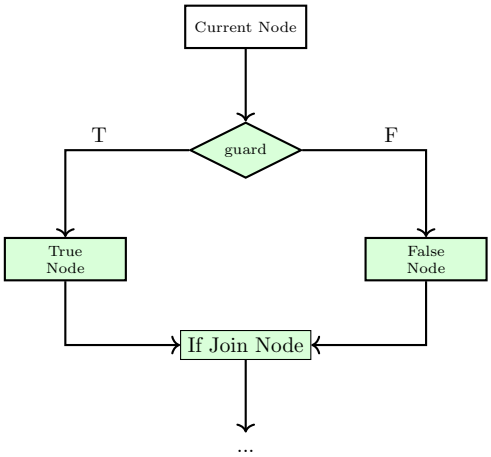


Fig. 5: If statement insertion. The builder creates a CONDITION NODE, two nodes for true/false arms, and an IF JOIN

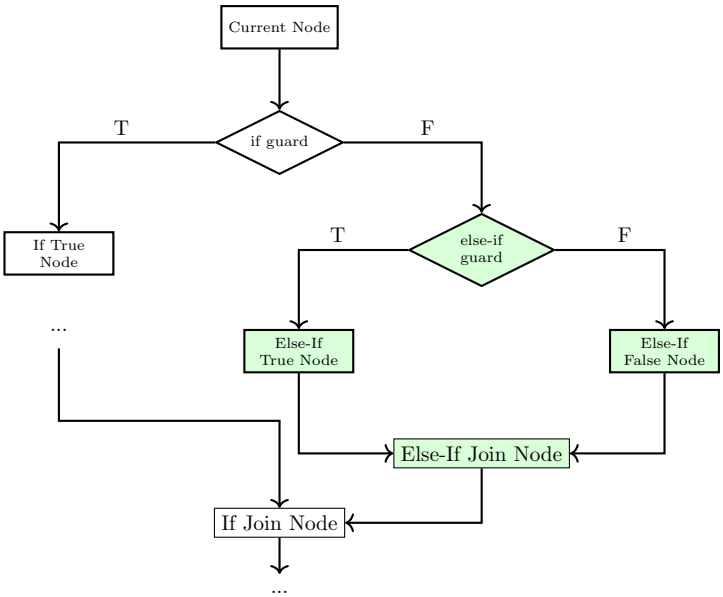


Fig. 6: Else-if statement insertion. The builder replaces the false arm with a new CONDITION NODE, two nodes, and an ELSE-IF JOIN

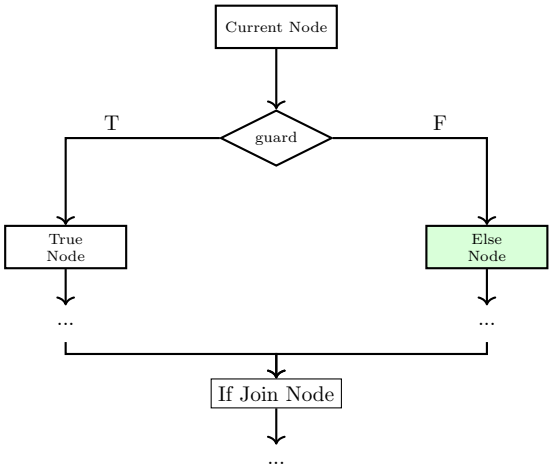


Fig. 7: Else statement insertion. The builder attaches a node to the false branch of the corresponding `if`/`else if`, connecting to the IF JOIN

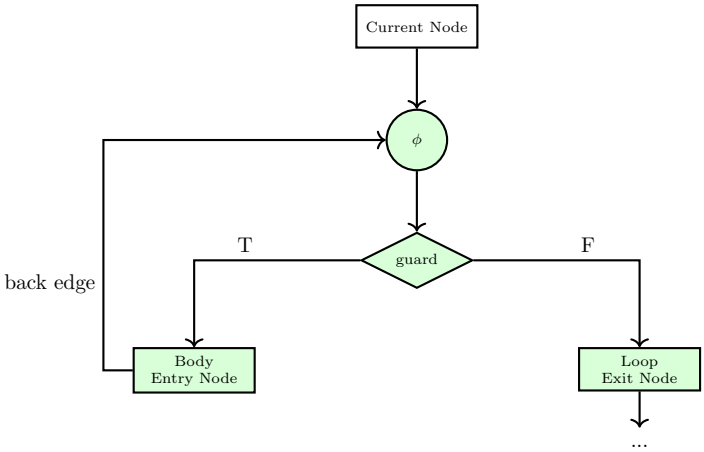


Fig. 8: While loop insertion. The builder creates a FIXPOINT EVALUATION NODE ϕ, a CONDITION NODE, a loop body node, and a LOOP EXIT NODE

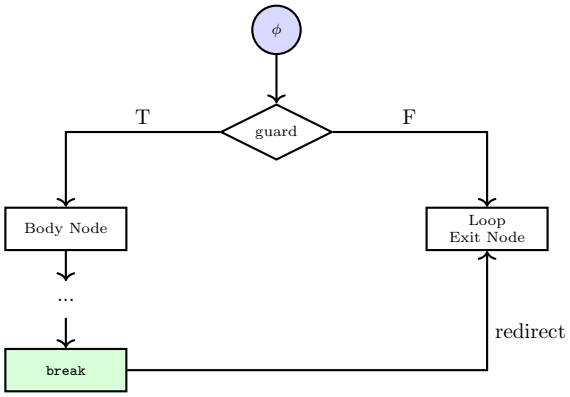


Fig. 9: Break statement insertion. The `break` node's outgoing edge is redirected to the LOOP EXIT NODE

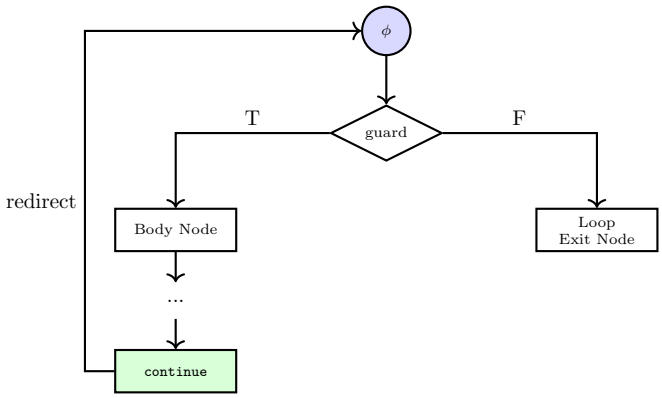


Fig. 10: Continue statement insertion. The `continue` node's outgoing edge is redirected to the loop's FIXPOINT EVALUATION NODE ϕ

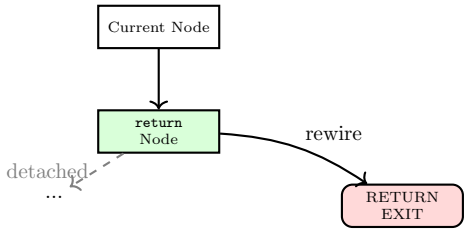


Fig. 11: Return statement insertion. The `return` node is rewired to the function's unique RETURN EXIT

Table 3: Line-to-node index mapping by statement type

Simple Statements			Compound Statements		
Statement	**Start Line**	**End Line**	**Statement**	**Start Line**	**End Line**
`Variable decl`	statement node	—	`if`	condition node	join node
`Assignment`	statement node	—	`else if`	condition node	join node
`break`	statement node	—	`else`	else node	join node
`continue`	statement node	—	`while`	condition node	exit node
`return`	statement node	—			
`require`	condition node	—			

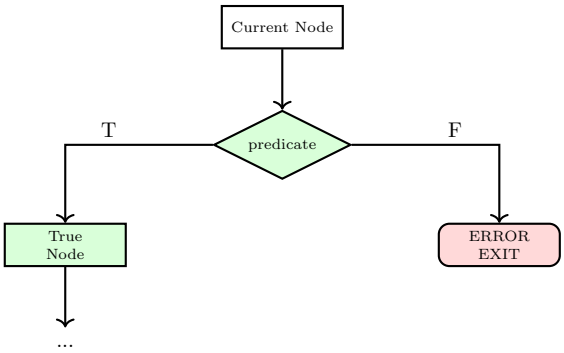


Fig. 12: Require statement insertion. The builder creates a CONDITION NODE with true edge to a node and false edge to the ERROR EXIT

These construction patterns enable SolQDebug to build the CFG incrementally as the user types each statement, without requiring the entire function body. The key challenge is determining *where* to insert each new node, which we address next.

3.4.2 Line-Based Insertion-Site Selection

Traditional CFG construction processes complete programs sequentially, building the entire graph in a single pass. In contrast, SolQDebug must handle partial code edits that specify only target line numbers. Since the CFG structure itself carries no line information, we cannot determine where an edit belongs without additional context. To enable line-based insertion-site selection, we maintain a line-to-node index during construction. Table 3 summarizes how statement types map lines to CFG nodes.

Simple statements index their start line only: most map to a statement node, while `require` maps to a condition node. Compound statements index both start and end lines: `if`/`else if`/`while` map their start line to a condition node and end line to a join/exit node, while `else` maps its start line to an else node and end line to the enclosing conditional's join node (see Figure 6 and Figure 7).

Algorithm 1 Dependent-Context Insertion

Require: CFG $G = (V, E)$, edit context $ctx \in \{\texttt{else\_if}, \texttt{else}\}$, current line L
Ensure: Condition node $c \in V$

1: $Queue \leftarrow$ FINDJOINNODE(L) ▷ find join node at or before line L
2: **if** $Queue = \emptyset$ **then**
3: **error** "No join node found at or before line L"
4: **end if**
5: $Visited \leftarrow \emptyset$
6: **while** $Queue \neq \emptyset$ **do** ▷ BFS through predecessors
7: $n \leftarrow$ DEQUEUE($Queue$)
8: **if** $n \in Visited$ **then continue**
9: **end if**
10: $Visited \leftarrow Visited \cup \{n\}$
11: **if** isCond(n) **and** CONDTYPE(n) $\in \{\texttt{if}, \texttt{else\_if}\}$ **then**
12: **return** n
13: **end if**
14: **for** $p \in$ PREDECESSORS(n) **do**
15: **if** $p \notin Visited$ **then**
16: ENQUEUE($Queue, p$)
17: **end if**
18: **end for**
19: **end while**
20: **error** "No matching condition node found for context ctx"

This indexing scheme enables Algorithms 1 and 2 to locate insertion sites efficiently. We dispatch based on whether the statement can exist independently:

- `else/else if` **(dependent contexts):** Must attach to a preceding `if/else if` condition. Algorithm 1 traverses CFG predecessors to find the condition node.
- **All other statements (independent contexts):** Can exist independently. Algorithm 2 uses a successor-first strategy to find the insertion point.

Both algorithms never mutate the graph and rely solely on the line-to-node index for efficient lookup.

Algorithm 1: Dependent-Context Insertion. Dependent contexts (`else/else if`) cannot exist independently and must attach to a preceding `if/else if` condition node. The algorithm proceeds as follows:

- **Line 1–3 (initialization):** Retrieves CFG nodes at or before line L to initialize the BFS queue. These nodes include the join node of the preceding conditional, which serves as the starting point for backward traversal. If no nodes are found, the dependent context is invalid.
- **Line 5–13 (BFS traversal):** Performs BFS through CFG predecessors to find the matching condition node of type `if` or `else_if`. The BFS ensures we find the *nearest* enclosing condition.
- **Line 15:** Reports an error if no matching condition is found.

Algorithm 2 Independent-Context Insertion

Require: CFG $G = (V, E)$, edit span ending at line L
Ensure: Insertion-site node $A \in V$ (no graph mutation here)

```
1:  s ← FindPostNode(L)                          ▷ find first node after line L
2:  if isLoopExit(s) or isJoin(s) then           ▷ closing a loop or selection
3:      n ← FindPreviousNode(L)                  ▷ condition if exists, else last node
4:      if isCond(n) then
5:          return BranchBlock(n, true)          ▷ insert in TRUE branch
6:      else
7:          return n
8:      end if
9:  else                                         ▷ basic successor
10:     Pred ← Predecessors(s)
11:     if |Pred| = 1 then
12:         return the unique element of Pred
13:     else
14:         error "Basic successor must have exactly 1 predecessor"
15:     end if
16: end if
```

Algorithm 2: Independent-Context Insertion. For independent contexts (all statements except `else`/`else if`), we employ a successor-first strategy: by first identifying the post node (the next statement by line number), we determine the correct insertion point based on its CFG structure. This approach handles all statement types uniformly:

- **Line 1 (find post node):** FindPostNode(L) retrieves the first CFG node after line L.
- **Line 3–8 (loop-exit/join):** If the post node s is a loop-exit or join node, we search backward from L to find the previous node. FindPreviousNode(L) returns a condition node if present (loop header or `if`), otherwise the last node before L. If it is a condition node, we return its TRUE branch to place the new statement inside the construct; otherwise, we return the node itself.
- **Line 10–13 (basic post node):** Otherwise, s is a basic statement node. We retrieve its CFG predecessors and verify there is exactly one. Our CFG construction ensures this invariant (branches merge at join nodes, loops exit through loop-exit nodes); any other count indicates a malformed CFG.

3.4.3 Abstract Interpretation for Incremental Analysis

SolQDebug provides instant feedback on source code edits by propagating updates only along affected CFG paths, avoiding full re-analysis. When the user inserts statements, Algorithms 1 and 2 splice new nodes into the CFG, and incremental reinterpretation propagates updates from seed nodes marking insertion points. For debug annotations, SolQDebug performs full interpretation from the function entry node,

ensuring all inspection points receive complete abstract states. Algorithm 3 performs incremental interpretation by propagating abstract states through a worklist-based dataflow analysis. When encountering loop headers, it delegates to Algorithm 4, which computes loop fixpoints using adaptive widening. The key innovation is EstimateIterations, which analyzes loop conditions to compute an adaptive threshold τ that defers widening. When debug annotations materialize concrete bounds (e.g., array lengths, parameter values), the analyzer infers tighter intervals for condition operands, raising τ to delay widening and preserve precision. Additionally, CondConverged detects early convergence by checking whether loop condition operands have stabilized to singleton intervals, allowing fixpoint computation to terminate before exhausting τ.

Algorithm 3 Incremental Interpretation

Require: CFG $G = (V, E)$; seed set S
Ensure: Environments updated along forward-reachable paths from S

1: $WL \leftarrow \langle\rangle$; $\quad inQ \leftarrow \emptyset$; $\quad Out \leftarrow$ snapshot map
2: **for all** $s \in S$ **do**
3: **if** $\neg$isSink$(s) \ \wedge \ s \notin inQ$ **then**
4: WL.enqueue(s); $\quad inQ \leftarrow inQ \cup \{s\}$
5: **end if**
6: **end for**
7: **while** $WL \neq \langle\rangle$ **do**
8: $n \leftarrow WL$.pop(); $\quad inQ \leftarrow inQ \setminus \{n\}$
9: $\hat{\sigma}_{\text{in}} \leftarrow \bigsqcup_{p \in \text{Pred}(n)}$ RefineByCondition(p, n) ▷ join predecessors with path refinement
10: **if** isLoopHeader(n) **then**
11: $exit \leftarrow$ Fixpoint(n) ▷ compute loop fixpoint (Algorithm 4)
12: EnqueueSuccessors$(exit, WL, inQ)$ **continue**
13: **end if**
14: $\hat{\sigma}_{\text{out}} \leftarrow$ Transfer$(n, \hat{\sigma}_{\text{in}})$
15: **if** $\hat{\sigma}_{\text{out}} \neq Out[n]$ **then**
16: $(n) \leftarrow \hat{\sigma}_{\text{out}}$; $\quad Out[n] \leftarrow \hat{\sigma}_{\text{out}}$
17: EnqueueSuccessors(n, WL, inQ)
18: **end if**
19: **end while**

Algorithm 3: Incremental Interpretation.

- **Line 1 (worklist and snapshot initialization):** Initialize empty worklist WL, in-queue set inQ to track enqueued nodes, and snapshot map Out to store previous node outputs for change detection.
- **Line 2–4 (seed node initialization):** Enqueue all seed nodes $s \in S$ (marking insertion points for incremental edits, or function entry for batch annotations), filtering out sink nodes (exit, error, return) that have no successors to propagate to.
- **Line 6–7 (worklist iteration and incoming environment):** Dequeue node n and compute its incoming environment $\hat{\sigma}_{\text{in}}$ by joining outputs from all predecessors.

RefineByCondition applies path-sensitive refinement: for condition nodes, it narrows operand intervals based on the edge truth label (true/false branch), pruning infeasible paths.

- **Line 9–11 (loop header delegation):** When encountering a loop header, delegate to Algorithm 4 to compute the loop fixpoint. After fixpoint converges, enqueue the loop-exit node's successors to continue analysis beyond the loop.
- **Line 13 (transfer function):** Apply the abstract transfer function to node n, computing output environment $\hat{\sigma}_{\text{out}}$ by interpreting statements (assignments, calls, etc.) using interval arithmetic and domain operations.
- **Line 14–17 (change detection and propagation):** Compare $\hat{\sigma}_{\text{out}}$ with the previous snapshot $Out[n]$. Only if changed, update node environment and snapshot, then enqueue successors. This ensures fixpoint termination by stopping propagation when environments stabilize.

Algorithm 4 Loop Fixpoint with Adaptive Widening

Require: loop header node h
Ensure: Converged abstract environments for loop body and exit

1: $L \leftarrow \textsc{LoopNodes}(h)$; $\quad Start \leftarrow \bigsqcup\{(p) \mid p \in (h) \setminus L\}$
2: $\tau \leftarrow \textsc{EstimateIterations}(h, Start)$ ▷ annotation-aware threshold
3: $vis[\cdot] \leftarrow 0$
4: **// Widening phase**
5: $WL \leftarrow \langle h \rangle$; $\quad In[h] \leftarrow Start$
6: **while** $WL \neq \langle\rangle$ **do**
7: $\quad n \leftarrow WL.\mathsf{pop}()$; $\quad vis[n] \leftarrow vis[n] + 1$
8: $\quad \hat{o} \leftarrow \textsc{Transfer}(n, In[n])$
9: $\quad$ **if** $\mathsf{isJoin}(n) \wedge vis[n] > \tau$ **then**
10: $\quad\quad \hat{o} \leftarrow \textsc{Widen}(Out[n], \hat{o})$ ▷ widen after τ visits
11: $\quad$ **else**
12: $\quad\quad \hat{o} \leftarrow Out[n] \sqcup \hat{o}$
13: $\quad$ **end if**
14: $\quad$ **if** $\mathsf{isJoin}(n) \wedge \textsc{CondConverged}(n)$ **then break** ▷ early stop
15: $\quad$ **end if**
16: $\quad$ **if** $\hat{o} \neq Out[n]$ **then**
17: $\quad\quad Out[n] \leftarrow \hat{o}$; $\quad \textsc{PropagateToSuccessors}(n, L, WL)$
18: $\quad$ **end if**
19: **end while**
20: $\textsc{NarrowingPhase}(L)$ ▷ standard descending iteration
21: **return** Out

Algorithm 4: Loop Fixpoint with Adaptive Widening.

- **Line 1 (identify loop nodes and pre-loop environment):** Collect all CFG nodes within the loop body. Compute $Start$ by joining environments from all loop-entry predecessors (excluding back edges).

- **Line 2 (EstimateIterations):** Evaluate both operands of the loop condition (e.g., `i < array.length`) in the pre-loop environment *Start*. For comparison operators ($<, \leq, >, \geq$), compute τ as the difference between operand bounds: e.g., given `i = [0,0]` and `array.length = [10,10]`, compute $\tau = 10 - 0 = 10$. When annotations provide concrete values, this yields tight thresholds; otherwise, a conservative default (typically 2) is used.
- **Line 3 (initialize visit counts):** Track how many times each join node has been visited to trigger widening after τ iterations.
- **Line 6–15 (widening phase):** Perform fixpoint iteration with worklist-based propagation. At each join node, apply widening only after τ visits (Line 10), otherwise perform standard join (Line 12). Line 11 implements **early convergence**: if loop condition operands stabilize to singleton intervals, terminate early without exhausting τ.
- **Line 17 (narrowing phase):** Apply standard narrowing (descending iteration with narrowing operator at join nodes) to refine over-approximations from widening.

Abstract Interpretation Framework. Our approach builds upon abstract interpretation frameworks for Solidity smart contracts (Halder et al., 2023; Halder, 2024). SolQDebug computes sound over-approximations of variable ranges using interval domains for integer types ($\widehat{\mathbb{Z}}_N, \widehat{\mathbb{U}}_N$), set abstractions for addresses and booleans, and on-demand materialization for composite types (arrays, mappings, structs). Unlike prior work focusing on invariant generation (Halder, 2024) or information flow analysis (Halder et al., 2023), we target interactive debugging with incremental refinement. The complete formal semantics (Tables 5 and 6) are in Appendix B.

4 Evaluation

To evaluate the practical benefits of SolQDebug in real-world development scenarios, we structure our empirical analysis around three key questions:

- **RQ1 – Responsiveness**: How much does SolQDebug reduce debugging latency compared to Remix?
- **RQ2 – Precision Sensitivity to Annotation Structure**: In a common Solidity pattern where inputs are normalized by division, how does the structure of operand intervals—overlapping vs. distinct—impact interval growth?
- **RQ3 – Loops**: How does SolQDebug's analysis approach affect precision in loop structures?

4.1 Experimental Setup

We evaluate SolQDebug on a controlled local setup to measure responsiveness, precision, and loop-handling capabilities under realistic debugging scenarios.

Experimental Setting. The evaluation environment consists of an 11th Gen Intel® Core™ i7-11390H CPU at 3.40GHz with 16.0 GB RAM, running Windows 10 (64-bit). SolQDebug is implemented in Python 3.x and operates directly on Solidity source

code without requiring compilation or deployment, using the ANTLR4-based parser described in Section 4.

Dataset Collection. We derive our dataset from DAppSCAN (Zheng et al., 2024), a large-scale benchmark containing 3,344 Solidity contracts compiled with version `>=0.8.0`. To ensure representative coverage across contract sizes, we first exclude contracts smaller than 4 KB (2,142 samples), which typically contain minimal logic unsuitable for debugging analysis. From the remaining 1,202 contracts, we sample approximately 10% from each of three size brackets:

- 4–10 KB (735 contracts): 70 samples
- 11–20 KB (304 contracts): 30 samples
- Over 20 KB (163 contracts): 20 samples

yielding 120 candidate contracts.

We then apply two filtering criteria: (1) excluding functions with multi-contract interactions (i.e., accessing variables or invoking functions from other contracts), as our analysis focuses on single-contract scenarios, and (2) excluding logic-free functions (e.g., those containing only assignments or return statements). From the filtered set, we select 30 representative functions for evaluation.

The selected contracts represent diverse DeFi scenarios including token transfers with custom logic, staking/vesting mechanisms, liquidity pool operations, oracle data processing, and marketplace transactions. Our selection prioritizes three dimensions of debugging complexity:

- **Computational complexity**—complex arithmetic patterns with chained computations across multiple statements, making value ranges hard to predict without interval tracking;
- **Data structure complexity**—structs with multiple fields, nested mappings, dynamic arrays, and mapping-to-struct patterns;
- **Control flow complexity**—loops with varying termination conditions, nested conditionals, and modifier-based access control.

Table 4 lists all 30 benchmark contracts with their source files, target functions, and line counts.

4.2 RQ1 - Responsiveness

To evaluate responsiveness, we measure *debugging latency*, defined as the time required to step through the execution of a function line-by-line. For Remix IDE, this corresponds to the duration from opening the debugger to stepping through the entire execution using the "step forward" button. For SolQDebug, it represents the time from a code modification to the display of updated variable information.

Since Remix IDE lacks built-in automated benchmarking capabilities, we developed `remix_benchmark` (Remix Benchmark, 2025), a Selenium (Selenium with Python, 2025)-based automation framework that programmatically drives the Remix web

File Name	Function	Lines
AloeBlend.sol	`_earmarkSomeForMaintenance`	537-552
Amoss.sol	`_burn`	453-467
AOC_BEP.sol	`updateUserInfo`	422-436
ATIDStaking.sol	`_insertLockedStake`	127-172
AvatarArtMarketPlace.sol	`_removeFromTokens`	163-176
Balancer.sol	`_addActionBuilderAt`	79-91
BitBookStake.sol	`viewFeePercentage`	205-208
CitrusToken.sol	`transferFrom`	53-60
Claim.sol	`getCurrentClaimAmount`	65-72
Core.sol	`revokeStableMaster`	147-163
CoreVoting.sol	`quorums`	38-50
Dai.sol	`transferFrom`	72-83
DapiServer.sol	`calculateUpdateInPercentage`	838-854
DeltaNeutralPancakeWorker02.sol	`getReinvestPath`	392-405
Dripper.sol	`_availableFunds`	111-119
EdenToken.sol	`transferFrom`	227-240
GovStakingStorage.sol	`updateRewardMultiplier`	103-120
GreenHouse.sol	`_calculateFees`	328-344
HubPool.sol	`_allocateLpAndProtocolFees`	907-923
Lock.sol	`pending`	49-63
LockupContract.sol	`_getReleasedAmount`	75-89
Meter_flat.sol	`_transfer`	349-361
MockChainlinkOracle.sol	`latestRoundData`	114-130
OptimisticGrants.sol	`configureGrant`	62-79
PercentageFeeModel.sol	`getEarlyWithdrawFeeAmount`	72-95
PoolKeeper.sol	`keeperTip`	235-247
ThorusBond.sol	`claimablePayout`	531-538
ThorusLottery.sol	`isWinning`	708-714
TimeLockPool.sol	`getTotalDeposit`	90-96
WASTR.sol	`withdrawFrom`	211-232

Table 4: Benchmark dataset: 30 representative contracts from DAppSCAN with diverse debugging scenarios.

interface to measure debugging latency. For each test function, `remix_benchmark` automates the full workflow: compilation, contract deployment, state variable initialization via manual storage slot assignment, parameter entry, transaction execution, and step-through debugging.

Unlike Remix's concrete execution model, which requires stepping through every bytecode instruction for each test input, SolQDebug uses AI with interval domain (Section **??**) that operates directly on the Solidity AST. Rather than enumerating concrete values one by one, the abstract interpreter represents inputs as intervals and computes abstract states over those intervals, analyzing all possible behaviors in a single pass without blockchain deployment. With this interval-based approach, SolQDebug maintains consistent latency regardless of test-case width Δ, which specifies the size of each input interval in debug annotations. In contrast, Remix must

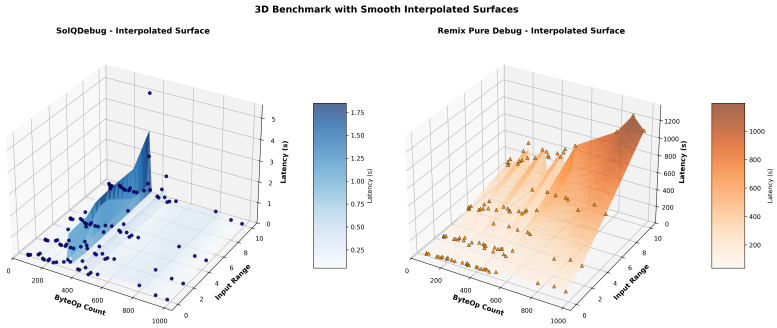


Fig. 13: Debugging latency comparison between Remix and SolQDebug across varying ByteOp counts and test-case widths Δ. SolQDebug maintains consistent sub-second latency regardless of function complexity or input interval size, while Remix's latency scales linearly with both dimensions.

execute each concrete input value separately, so its latency scales linearly with the number of test inputs.

Although SolQDebug is designed for interactive use within a Solidity editor, all experiments simulate this behavior in a controlled scripting environment. For each function, we reconstruct a sequence of incremental edits and annotations that mimic realistic developer activity. These fragments are streamed into the interpreter to measure latency and interval growth under reproducible conditions.

We evaluated 30 functions across 4 test-case widths $\Delta \in \{0, 2, 5, 10\}$, yielding 120 total measurements for SolQDebug. For Remix, we measured each function once per Δ value to demonstrate the linear scaling behavior. Figure 13 illustrates the latency comparison across these dimensions.

For Remix, the debugging latency ranged from 25.1 to 124.6 seconds (median: 53.0s), reflecting the time required to step through bytecode operations in the debugger. This latency scales linearly with test-case width, as each additional input value requires a separate transaction execution and complete bytecode step-through.

In contrast, SolQDebug completed analysis in 0.03–5.09 seconds (median: 0.15s) across all 120 measurements. SolQDebug's latency remains nearly constant regardless of test-case width, as AI analyzes all input combinations symbolically in a single pass. This results in a median speedup of approximately 350× over Remix for pure debugging time. In practice, Remix users also incur additional overhead from compilation, deployment, and state initialization, further increasing total debugging time. SolQDebug eliminates these preparatory steps entirely, enabling immediate feedback during code editing.

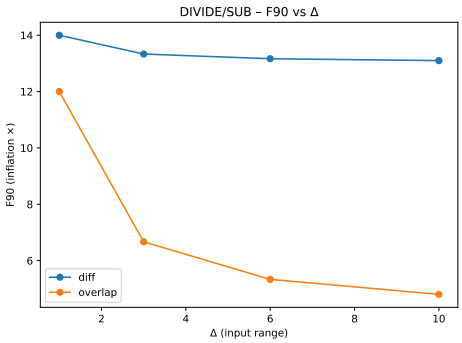


Fig. 14: F90 (90th percentile of interval inflation) for `Lock::pending` under OVERLAP and DIFF annotation patterns. As input width (Δ) increases, OVERLAP achieves progressively tighter precision (F90: 12.0 $\rightarrow$ 4.8), while DIFF maintains near-constant inflation (F90 $\approx$ 13–14).

Answer to RQ1: SOLQDEBUG achieves sub-second debugging latency (median: 0.15s), providing approximately 350× faster line-by-line variable inspection compared to Remix's bytecode step-through (median: 53.0s). Unlike Remix, whose latency scales linearly with input space size, SOLQDEBUG maintains consistent performance regardless of test-case width through abstract interpretation. When including compilation and deployment overhead, the practical speedup reaches approximately 650×.

4.3 RQ2 - Impact of Annotation Patterns on Precision in Complex Arithmetic Operations

Real-world smart contracts frequently employ complex arithmetic operations involving multiplication and division to compute financial quantities such as rewards, fees, and vesting schedules. These operations inherently amplify interval widths during AI due to the combinatorial nature of interval arithmetic. Understanding how annotation structure influences precision in such contexts is critical for practical adoption of SOLQDEBUG.

To investigate this, we examine the `pending` function from `Lock.sol` in our benchmark dataset. The function uses complex arithmetic operations involving multiplication and division. In particular, multiplication is critical: in interval arithmetic, it computes the Cartesian product of endpoint combinations $\{a_{\min} \times b_{\min}, a_{\min} \times b_{\max}, a_{\max} \times b_{\min}, a_{\max} \times b_{\max}\}$. Consequently, when operand intervals are disjoint, this combinatorial expansion generates significantly wider output ranges.

To assess this effect, we evaluate two annotation strategies under varying input widths $\Delta \in \{1, 3, 6, 10\}$. In the OVERLAP style, all input variables share a common base range (e.g., $[100, 100 + \Delta]$). In the DIFF style, each variable occupies a distinct range (e.g., $[100, 100 + \Delta]$, $[300, 300 + \Delta]$, $[500, 500 + \Delta]$). We measure F90, the 90th percentile of the inflation factor $F = \frac{\text{exit\_width}}{\text{input\_width}}$.

As shown in Figure 14, the OVERLAP strategy consistently produces tighter bounds: as Δ increases from 1 to 10, F90 decreases from 12.0 to 4.8, indicating that wider inputs lead to proportionally smaller relative growth. In contrast, the DIFF strategy maintains nearly constant inflation (F90 $\approx$ 13–14) regardless of input width. This difference arises from interval multiplication semantics: when annotations align operands to overlapping ranges, the extreme products remain closer to the midpoint, limiting excessive interval expansion. Conversely, disjoint ranges maximize the distance between endpoint combinations, causing output intervals to span unnecessarily large ranges.

Moreover, we observe similar patterns in other contracts from our benchmark dataset that employ multiplication or division in their arithmetic. These include reward computations (GovStakingStorage_c), fee calculations (GreenHouse_c, HubPool_c), vesting schedules (LockupContract_c), and proportional payouts (ThorusBond_c). Across these contracts, overlapping annotations consistently yield tighter precision than disjoint ranges. This pattern demonstrates that for real-world contracts with multiplication or division, developers can significantly improve analysis precision by choosing overlapping rather than disjoint annotations.

Answer to RQ2: For real-world contracts using multiplication or division, overlapping annotations yield significantly lower interval inflation than disjoint annotations, due to interval multiplication's combinatorial nature.

4.4 RQ3 - Loops

AI with the interval domain faces a well-known precision challenge in loops. To guarantee termination, widening operators must be applied after a bounded number of iterations, often causing intervals to expand to $\top$ or $[0, \infty]$ even when the actual loop bounds are finite. However, Solidity's properties create opportunities for mitigation. Gas costs limit loop complexity, and loop conditions commonly depend on simple values such as array lengths, mapping sizes, or bounded counters.

We address this challenge through an annotation-guided widening threshold mechanism. When developers annotate values that determine loop bounds (e.g., array lengths), the analyzer evaluates the loop condition using this information to compute adaptive thresholds that delay widening, improving precision while maintaining soundness and termination guarantees.

To evaluate the effectiveness of this approach, we analyze the five loop-containing functions from our benchmark dataset (Table 4): `updateUserInfo` (AOC_BEP), `_addActionBuilderAt` (Balancer), `revokeStableMaster` (Core), `getTotalDeposit` (TimeLockPool), and `_removeFromTokens` (AvatarArtMarketPlace). We identify four distinct patterns that demonstrate varying levels of precision under our approach.

Pattern 1: Constant-Bounded Loops with Simple Updates. When loop conditions reference only constants and the loop body contains only simple assignments, ESTIMATEITERATIONS computes precise thresholds without annotations. For example, `updateUserInfo` (AOC_BEP) uses `for (uint256 i = 1; i <= 4; i++)` with $\tau = 4$ computed from the constant bound. The small constant bound and simple updates

allow convergence without triggering widening. The analysis produces precise interval `userInfo[account].level` $\in [1, 4]$.

Pattern 2: Annotation-Enabled Convergence. When loop bounds depend on dynamic values but the loop body performs only simple updates, annotations enable precise convergence. `_addActionBuilderAt` (Balancer) uses `for (uint8 i = 0; i < additionalCount; i++)` where `additionalCount` is computed from function inputs. Annotating these inputs allows the evaluator to compute `additionalCount = 4` and set $\tau = 4$. The simple loop body converges precisely.

`revokeStableMaster` (Core) exhibits similar behavior. It iterates `for (uint256 i = 0; i < stablecoinListLength - 1; i++)` with simple index-based operations. Annotating the array length enables precise threshold computation and convergence.

Pattern 3: Uninitialized Local Variables (Developer-Fixable). When local variables lack explicit initialization, precision loss can occur. `getTotalDeposit` (TimeLockPool) declares `uint256 total;` without initialization and then accumulates values in a loop. SolQDebug conservatively models uninitialized variables as $\top$ (unknown). Any arithmetic operation propagates $\top$. This causes `total` to remain $\top = [0, 2^{256} - 1]$ throughout the analysis.

This pattern represents a developer-fixable limitation. Explicitly initializing `total = 0` would enable precise tracking. Annotations cannot compensate for missing initialization because the interval domain soundly treats uninitialized reads as arbitrary values.

Pattern 4: Data-Dependent Accumulation. Even when loop bounds are precisely known, variables that accumulate based on data-dependent conditions may diverge under widening. `_removeFromTokens` (AvatarArtMarketplace) illustrates this limitation. The loop iterates `for (uint tokenIndex = 0; tokenIndex < tokenCount; tokenIndex++)` where `tokenCount` is known from annotations. Inside the loop, `resultIndex` increments conditionally based on array element comparisons. The accumulator depends on data values rather than the loop index itself.

Once the widening threshold is exceeded, SolQDebug widens `resultIndex` to $[0, \infty]$. The interval domain cannot track correlations between array contents and conditional accumulation. This pattern represents an inherent limitation of the interval domain. Annotations of iteration bounds cannot prevent widening when variable updates depend on unpredictable data rather than iteration count.

Answer to RQ3: SolQDebug improves loop analysis precision for constant-bounded and annotation-enabled dynamic loops. Remaining precision loss arises from developer-fixable initialization issues and inherent interval domain limitations in tracking data-dependent accumulation.

5 Discussion

5.1 Why use Abstract Interpretation for Debugging

In this work, debugging refers to a developer-led, interactive exploration activity that occurs during code authoring, before deployment: developers vary symbolic (interval)

inputs and immediately observe branch reachability, guard validity, and value bounds at the source level. This edit-time feedback loop requires a technique that (1) terminates reliably, (2) produces results developers can inspect and interpret, and (3) scales to near-keystroke responsiveness.

AI was selected over symbolic execution and proof-based verification for three primary reasons:

- **Termination.** AI enforces convergence through widening at loops and joins at merges, avoiding the path explosion inherent in symbolic execution.
- **Explainability.** Each result is an abstract value in a well-defined lattice. With interval domains, the mapping from inputs to outputs is explicit as ranges, enabling developers to trace dataflow effects and reason about behavior at the statement level.
- **Responsiveness.** Interval transfer functions are lightweight, enabling millisecond-scale updates that align with the edit cycle. In contrast, symbolic engines routinely explore multiple paths even for small edits, which can degrade interactivity.

While formal verification provides stronger guarantees, it requires fully specified properties and invariants, which are costly to develop during early iterations. SolQDebug is designed to bridge the gap between code authoring and testing or verification—offering immediate, sound feedback with minimal annotation overhead.

In the debugging context, intervals strike a practical balance between precision and speed, offering three key advantages: (i) they align with developers' mental model of "possible ranges," (ii) they expose boundary effects (e.g., overflow thresholds, guard satisfaction regions) without requiring concrete inputs, and (iii) they compose predictably through joins and widenings. Furthermore, intervals provide a natural interface for annotations: developers can *shape* symbolic inputs (e.g., make them overlapping or disjoint) and directly observe how these configurations affect control flow and computed ranges.

While AI's precision is conservative by design, edit-time usability depends on providing developers with simple mechanisms to control precision without sacrificing responsiveness. Our evaluation demonstrates the effectiveness of three such mechanisms:

- **Annotation structure.** Overlapping operand intervals often bound output ranges more tightly than disjoint ones in division-normalized arithmetic (cf. RQ2), reducing imprecision with no runtime cost.
- **Annotation width.** Narrower inputs shrink joins and delay widening; developers can start narrow and broaden gradually to probe stability.
- **Guard-guided narrowing.** Making explicit the intended `require`/`if` guards in annotations tightens feasible states early and improves precision along the taken branch at negligible cost.

In scenarios requiring stricter precision (e.g., inside data-driven loops), developers can temporarily fall back to concrete inputs for detailed inspection, then return to intervals for broader exploration. This "concrete when needed, symbolic by default" approach preserves interactivity while maintaining actionable results.

5.2 Limitation

Our current scope and analysis introduce several limitations. First, we focus on single-contract, single-transaction functions. Inter-contract calls, multi-transaction workflows, proxies, and inheritance hierarchies are out of scope in the present implementation. As a result, we have not yet conducted a developer study in larger project settings; the usability and interpretability of edit-time feedback across multi-contract workflows remain unvalidated.

Second, the interval domain exhibits inherent precision limitations in loops with data-dependent accumulation. As demonstrated in RQ3 (Pattern 4), when loop variables accumulate conditionally based on data values rather than iteration count, the domain cannot track these correlations and may widen variables to imprecise ranges. While this trade-off preserves edit-time responsiveness, developers encountering such patterns should consider concrete inputs or relational domains for more precise analysis.

6 Related Works

6.1 Solidity IDEs and Debuggers

Modern Solidity development environments either embed a debugger or integrate external debugging plug-ins. Remix IDE (Remix IDE, 2025) is the most widely used web IDE; it supports syntax highlighting, one-click compilation, and a bytecode-level debugger that lets users step through EVM instructions and inspect stack, memory, and storage. Hardhat (Hardhat, 2025) is a Node.js-based framework that couples the Solidity compiler with an Ethereum runtime; its Hardhat Debug plug-in attaches a Remix-style debugger to locally broadcast transactions inside Visual Studio Code. Foundry Forge (Foundry Forge, 2025) is a command-line toolchain oriented toward fast, reproducible unit testing; the command `forge test` spins up an ephemeral fork, deploys contracts, executes annotated test functions, and enables replay through Forge Debug. Solidity Debugger Pro (Solidity Debugger Pro, 2025) is a Visual Studio Code extension that performs runtime debugging over concrete transactions and integrates with Hardhat; in practice, many workflows create a small auxiliary contract that calls the target functions so that state changes can be observed step by step.

In short, these debuggers operate on compiled artifacts or post-deployment traces and rely on transaction replay and EVM-level stepping. They do not accept partial, in-flight source fragments nor provide symbolic (interval) input modeling or millisecond edit-time feedback. By contrast, SolQDebug targets pre-deployment authoring, accepts partial fragments and symbolic annotations, and reports line-level effects via AI during editing.

6.2 Solidity Vulnerability Detection and Verification

A rich body of work analyzes smart contracts for security issues using four main families of techniques. Static analysis tools reason over source or bytecode without running the contract. Representative systems include rule- or pattern–based analyzers such as Securify and Slither (Tsankov et al., 2018 2019), symbolic-execution–assisted

detectors like Mythril (Yao et al., 2022), knowledge-graph–based reasoning such as Solidet (Hu et al., 2023), and bytecode CFG refinement as in Ethersolve (Pasqua et al., 2023). Dynamic testing and fuzzing exercise deployed or locally simulated contracts to uncover faults and security issues: ContractFuzzer mutates ABI-level inputs (Jiang et al., 2018), Echidna brings property-based fuzzing into developer workflows (Grieco et al., 2020), sFuzz adapts scheduling for higher coverage (Nguyen et al., 2020), TransRacer finds transaction-ordering races (Ma et al., 2023), and Ityfuzz leverages snapshotting to decouple executions from chain nondeterminism (Shou et al., 2023). Formal verification aims to prove safety properties or refute counterexamples at compile time; examples include ZEUS, VeriSmart, and SmartPulse (Kalra et al., 2018; So et al., 2020; Stephens et al., 2021). Finally, AI-based approaches train models to predict vulnerabilities or triage candidates, e. g., via data-flow–aware pretraining, IoT-oriented classifiers, or prompt-tuning for detector adaptation (Wu et al., 2021; Zhou et al., 2022; Yu et al., 2023).

These approaches have substantially advanced vulnerability detection and property checking for fully written contracts. However, they are not designed to provide interactive, edit-time feedback to developers while code is still under construction. They typically analyze post-compilation artifacts or deployed bytecode and expect complete program units. SolQDebug complements this line of work by focusing on pre-deployment authoring: it accepts partial fragments and symbolic (interval) inputs and produces line-by-line feedback inside the editor.

6.3 Solidity-Specific Abstract Interpretation Frameworks

AI is a well-established framework for static analysis and has been adapted to many programming languages. Two recent studies apply it to Solidity (Halder et al., 2023; Halder, 2024). The first uses the Pos domain to construct a theoretical model for taint (information-flow) analysis (Halder et al., 2023), while the second employs the Difference-Bound Matrix (DBM) domain to generate state invariants and detect reentrancy vulnerabilities, including the DAO attack (Halder, 2024; Mehar et al., 2019). However, both approaches operate on fully written contracts and provide no support for line-by-line interpretation or developer interaction within an IDE.

SolQDebug adapts AI for an interactive setting. It incrementally updates both the control-flow graph and the abstract state in response to each edit. Developer-supplied annotations serve as a first-class input mechanism, reflecting how debugging often involves varying symbolic inputs. These annotations are internally represented as linear-inequality constraints, and form an integral part of interactive debugging by enabling symbolic reasoning over developer-specified inputs. This design improves interpretability and control within the interval domain by leveraging symbolic constraints, while maintaining keystroke-level responsiveness. As a result, SolQDebug updates variable ranges directly in the Solidity editor, allowing developers to observe how values evolve in response to each edit.

6.4 Interactive Abstract Interpretation for Traditional Languages

In recent years, traditional languages have seen a surge of interest in making abstract interpretation interactive, integrating it directly into IDEs to provide live analysis feedback during editing (Stein et al., 2021 2024; Erhard et al., 2024; Riouak et al., 2024; Chimdyalwar, 2024). Stein et al. (2021) proposed demanded abstract interpretation, which incrementally rebuilds only the analysis nodes touched by an edit. A follow-up Stein et al. (2024) generalized this to procedure summaries, enabling interprocedural reuse. Erhard et al. (2024) extended Goblint with incremental support for multithreaded C, selectively recomputing only genuinely affected facts and maintaining IDE-level responsiveness. Riouak et al. (2024) introduced IntraJ, an LSP-integrated analyzer for Java 11 that computes only the AST and data-flow facts needed for the current view, keeping feedback under 100 ms. Chimdyalwar (2024) achieved fast yet precise interval analysis on call graphs via one top-down and multiple bottom-up passes, and later introduced an incremental variant that revisits only the impacted functions.

Unlike these frameworks for C or Java, SolQDebug is designed specifically for Solidity. It supports in-flight code fragments and range annotations as first-class input. It incrementally updates only the current basic block in the CFG while reusing previously computed abstract states. Finally, it combines these with an interval domain guided by developer-supplied annotations, which act as input to represent the exploratory nature of debugging. This architecture enables keystroke-level feedback without requiring recompilation, redeployment, or transaction execution. It bridges the gap between Solidity development and the interactive tooling common in traditional programming environments.

7 Conclusion

We introduced SolQDebug, a source-level interactive debugger for Solidity that provides millisecond feedback without requiring compilation, deployment, or transaction replay. By combining an interactive parser with seven specialized entry rules, dynamic control-flow graph updates, and annotation-guided interval-based abstract interpretation with adaptive widening, SolQDebug enables responsive, line-by-line inspection directly within the Solidity editor. Our evaluation on 30 real-world functions from DAppSCAN demonstrates a median speedup of 350× over Remix IDE (0.15s vs. 53.0s), achieving sub-second debugging latency regardless of symbolic input width. We show that overlapping annotation patterns yield significantly tighter precision than disjoint patterns in multiplication-heavy arithmetic, and that adaptive widening thresholds enable precise convergence for constant-bounded and annotation-enabled dynamic loops while soundly handling data-dependent accumulation patterns. These results demonstrate that SolQDebug's design effectively bridges the interactivity gap in Solidity debugging and brings the development experience closer to that of modern debugging workflows.

Future work includes extending SolQDebug to inter-contract and multi-transaction contexts, incorporating loop summarization for higher precision, and conducting user

studies to assess its practical adoption and usability. We also plan to apply analysis based on the EVM Object Format (EOF) to support inter-contract debugging when source code is unavailable, as Ethereum moves toward structured bytecode formats in upcoming hard forks.

Acknowledgements

This work was supported by the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (RS-2021-II210177, High Assurance of Smart Contract for Secure Software Development Life Cycle).

Author Contributions

Inseong Jeon participated in conceptualization, methodology design, system implementation, data collection, experiments, and manuscript writing. Sundeuk Kim and Hyunwoo Kim assisted in experiments, data collection and analysis, and contributed to manuscript writing. Hoh Peter In provided resources, assisted in editing the manuscript, and supervised the entire project. All authors reviewed and approved the final version of the manuscript.

Data Availability

The curated benchmark dataset of 30 Solidity contracts derived from DAppSCAN (Zheng et al., 2024), along with the evaluation scripts and experimental results, are available at https://github.com/iwwyou/SolDebug/tree/main.

Declarations

Competing interests The authors declare no competing interests.
Ethical approval Not applicable since there are no human and/or animal studies included in this paper.

References

ANTLR: https://www.antlr.org/ (2025). Accessed November 2025

ChatGPT: https://chatgpt.com/ (2025). Accessed November 2025

Chen, X., et al.: Characterizing smart contract evolution. ACM Transactions on Software Engineering and Methodology (2025)

Chimdyalwar, B.: Fast and precise interval analysis on industry code. In: 2024 IEEE 35th International Symposium on Software Reliability Engineering Workshops (ISSREW) (2024)

ConsenSys Diligence: Python Solidity Parser. https://github.com/ConsenSysDiligence/python-solidity-parser (2025). Accessed November 2025

Cousot, P., Cousot, R.: 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 (POPL) (1977)

Erhard, J., et al.: Interactive abstract interpretation: reanalyzing multithreaded C programs for cheap. International Journal on Software Tools for Technology Transfer (2024)

Foundry Forge: https://book.getfoundry.sh/reference/forge/forge/ (2025). Accessed November 2025

Grieco, G., et al.: Echidna: effective, usable, and fast fuzzing for smart contracts. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 557–560 (2020)

Halder, R., et al.: Analyzing information flow in Solidity smart contracts. In: Distributed Computing to Blockchain, pp. 105–123. Academic Press (2023)

Halder, R.: State-based invariant property generation of Solidity smart contracts using abstract interpretation. In: 2024 IEEE International Conference on Blockchain (2024)

Hardhat: https://hardhat.org/ (2025). Accessed November 2025

Hu, T., et al.: Detect defects of Solidity smart contract based on the knowledge graph. IEEE Transactions on Reliability 73(1), 186–202 (2023)

JetBrains: PyCharm. https://www.jetbrains.com/pycharm/ (2025). Accessed November 2025

Jiang, B., Liu, Y., Chan, W.K.: ContractFuzzer: fuzzing smart contracts for vulnerability detection. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE) , pp. 259–269 (2018)

Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS) (2018)

Llama: https://www.llama.com/ (2025). Accessed November 2025

Ma, C., Song, W., Huang, J.: TransRacer: function dependence-guided transaction race detection for smart contracts. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 947–959 (2023)

Mehar, M.I., et al.: Understanding a revolutionary and flawed grand experiment in blockchain: the DAO attack. Journal of Cases on Information Technology (2019)

Microsoft Visual Studio: https://visualstudio.microsoft.com/ko/ (2025). Accessed November 2025

Nguyen, T.D., et al.: sFuzz: an efficient adaptive fuzzer for Solidity smart contracts. In: Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE), pp. 778–788 (2020)

Pasqua, M., et al.: Enhancing Ethereum smart-contracts static analysis by computing a precise control-flow graph of Ethereum bytecode. Journal of Systems and Software 200, 111653 (2023)

Remix IDE: https://remix.ethereum.org/ (2025). Accessed November 2025

Remix Benchmark: https://github.com/iwwyou/SolDebug/tree/main/Evaluation/RQ1_Latency (2025). Accessed November 2025

Riouak, I., et al.: IntraJ: an on-demand framework for intraprocedural Java code analysis. International Journal on Software Tools for Technology Transfer (2024)

Rival, X., Yi, K.: Introduction to Static Analysis: an Abstract Interpretation Perspective (2020)

Selenium with Python: https://selenium-python.readthedocs.io/ (2025). Accessed November 2025

Shou, C., Tan, S., Sen, K.: Ityfuzz: snapshot-based fuzzer for smart contract. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 322–333 (2023)

So, S., et al.: Verismart: a highly precise safety verifier for Ethereum smart contracts. In: 2020 IEEE Symposium on Security and Privacy (SP), pp. 1678–1694 (2020)

Solidity Compiler in Python (solcx): https://solcx.readthedocs.io/en/latest/ (2025). Accessed November 2025

Solidity documentation: https://docs.soliditylang.org/en/v0.8.30/ (2025). Accessed November 2025

Solidity Debugger Pro: https://www.soliditydbg.org/ (2025). Accessed November 2025

Solidity Language Grammar: https://docs.soliditylang.org/en/v0.8.30/grammar.html (2025). Accessed November 2025

Solidity Language Grammar Rule of SolQDebug : https://github.com/iwwyou/SolDebug/blob/main/Parser/Solidity.g4 . Accessed November 2025

Stein, B., Chang, B.-Y.E., Sridharan, M.: Demanded abstract interpretation. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI) (2021)

Stein, B., Chang, B.-Y.E., Sridharan, M.: Interactive abstract interpretation with demanded summarization. ACM Transactions on Programming Languages and Systems (2024)

Stephens, J., et al.: SmartPulse: automated checking of temporal properties in smart contracts. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 555–571 (2021)

Tsankov, P., et al.: Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 67–82 (2018)

Tsankov, P., et al.: Slither: a static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), pp. 8–15 (2019)

Wu, H., et al.: Peculiar: smart contract vulnerability detection based on crucial data-flow graph and pre-training techniques. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), pp. 378–389 (2021)

Yao, Y., et al.: An improved vulnerability detection system of smart contracts based on symbolic execution. In: 2022 IEEE International Conference on Big Data (Big Data), pp. 3225–3234 (2022)

Yu, L., et al.: PSCVFinder: a prompt-tuning based framework for smart contract vulnerability detection. In: 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE), pp. 556–567 (2023)

Zheng, Z., et al.: Dappscan: building large-scale datasets for smart contract weaknesses in dApp projects. IEEE Transactions on Software Engineering (2024)

Zhou, Q., et al.: Vulnerability analysis of smart contract for blockchain-based IoT applications: a machine learning approach. IEEE Internet of Things Journal 9(24), 24695–24707 (2022)

Zou, W., et al.: Smart contract development: challenges and opportunities. IEEE Transactions on Software Engineering (2019)

A Interactive Parser Grammar Specification

This appendix provides the complete grammar specification for SolQDebug's interactive parser.

A.1 Entry Rules for Solidity Program Fragments

A.1.1 Rule 1: `interactiveSourceUnit`

Purpose. Accepts top-level declarations: functions, contracts, interfaces, libraries, state variables, pragmas, and imports.
Grammar:

```
interactiveSourceUnit
  : (interactiveStateVariableElement | interactiveFunctionElement
    | interfaceDefinition | libraryDefinition | contractDefinition
    | pragmaDirective | importDirective)* EOF ;
```

A.1.2 Rule 2: `interactiveEnumUnit`

Purpose. Accepts enum member items added after the enum shell.
Grammar:

```
interactiveEnumUnit : (interactiveEnumItems)* EOF;
interactiveEnumItems : identifier (',' identifier)*;
```

A.1.3 Rule 3: `interactiveStructUnit`

Purpose. Accepts struct member declarations added after the struct shell.
Grammar:

```
interactiveStructUnit : (structMember)* EOF;
structMember : typeName identifier ';' ;
```

A.1.4 Rule 4: `interactiveBlockUnit`

Purpose. Accepts statements and control-flow skeletons inside function bodies.
Grammar:

```
interactiveBlockUnit
  : (interactiveBlockItem)* EOF;

interactiveBlockItem
  : interactiveStatement | uncheckedBlock;

interactiveStatement
  : interactiveSimpleStatement
  | interactiveIfStatement
```

```
  | interactiveForStatement
  | interactiveWhileStatement
  | interactiveDoWhileDoStatement
  | interactiveTryStatement
  | returnStatement
  | emitStatement
  | revertStatement
  | requireStatement
  | assertStatement
  | continueStatement
  | breakStatement
  | assemblyStatement;

interactiveIfStatement
  : 'if' '(' expression ')' '{' '}' ;

interactiveForStatement
  : 'for' '(' (simpleStatement | ';') expression? ';' expression? ')' '{' '}' ;

interactiveWhileStatement
  : 'while' '(' expression ')' '{' '}' ;

interactiveDoWhileDoStatement
  : 'do' '{' '}' ;

interactiveTryStatement
  : 'try' expression ('returns' '(' parameterList ')')? '{' '}' ;
```

The `interactiveStatement` production includes skeleton rules for control structures with empty bodies (e.g., `interactiveIfStatement`, `interactiveForStatement`), enabling incremental construction of control flow. As developers type statements inside these empty bodies, `interactiveBlockUnit` is recursively invoked to parse each new line.

A.1.5 Rule 5: `interactiveDoWhileUnit`

Purpose. Accepts the `while` tail of a `do{...}` loop.
Grammar:

```
interactiveDoWhileUnit : (interactiveDoWhileWhileStatement)* EOF;
interactiveDoWhileWhileStatement : 'while' '(' expression ')' ';' ;
```

A.1.6 Rule 6: `interactiveIfElseUnit`

Purpose. Accepts `else` or `else if` branches.
Grammar:

```
interactiveIfElseUnit : (interactiveElseStatement)* EOF;
interactiveElseStatement : 'else' (interactiveIfStatement | '{' '}') ;
```

A.1.7 Rule 7: `interactiveCatchClauseUnit`

Purpose. Accepts `catch` clauses following a `try`.
Grammar:

```
interactiveCatchClauseUnit : (interactiveCatchClause)* EOF;
interactiveCatchClause : 'catch' (identifier? '(' parameterList ')')? '{' '}' ;
```

A.2 Entry Rule for Debugging Annotations

A.2.1 `debugUnit`

Purpose. Parses batch-annotation lines that specify initial abstract values for variables.
Annotation types:

- `@GlobalVar`: Assigns values to global variables (e.g., `msg.sender`, `block.timestamp`)
- `@StateVar`: Assigns values to contract state variables
- `@LocalVar`: Assigns values to function parameters and local variables

Grammar:

```
debugUnit : (debugGlobalVar | debugStateVar | debugLocalVar)* EOF;
debugGlobalVar : '//' '@GlobalVar' identifier ('.' identifier)? '=' globalValue ';' ;
debugStateVar : '//' '@StateVar' lvalue '=' value ';' ;
debugLocalVar : '//' '@LocalVar' lvalue '=' value ';' ;
```

Supported L-value patterns: Simple variables, array/mapping access (`arr[i]`, `map[key]`), struct fields (`s.field`), and nested combinations.
Value specification: Integer intervals `[l,u]`, symbolic addresses `symbolicAddress n`, boolean values, and symbolic placeholders.

B Abstract Domain and Formal Semantics

This appendix presents the abstract domain definitions and formal semantics used by SolQDebug's abstract interpreter. The framework is based on interval analysis for numeric types, set domains for addresses, and lazy materialization for composite data structures.

B.1 Language Syntax

We consider a subset of Solidity focusing on core control structures, expressions, and state manipulation relevant to our analysis.

Expressions:

$$\begin{aligned} e \in \mathsf{Expr} \ ::= \ & n \mid x \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{address\_literal} \\ & \mid e_1 \oplus e_2 \mid e_1 \ \odot \ e_2 \mid e_1 \ ? \ e_2 : e_3 \\ & \mid e.f \mid e_1[e_2] \mid f(\overline{e}) \mid \neg e \mid \mathsf{delete}\ e \end{aligned}$$

where $\oplus \in \{+, -, *, /, \%, **, \&\&, ||, \&, |, \wedge, <<, >>\}$ and $\odot \in \{<, \leq, >, \geq, ==, \neq\}$.
Statements:

$$\begin{aligned} s \in \mathsf{Stmt} \ ::= \ & \mathsf{skip} \mid s_1; s_2 \mid \tau\ x; \mid \tau\ x = e; \\ & \mid lv := e \mid \mathsf{delete}\ lv \\ & \mid \mathsf{if}\ p\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \\ & \mid \mathsf{while}\ p\ \mathsf{do}\ s \\ & \mid \mathsf{for}\ init; p; incr\ \mathsf{do}\ s \\ & \mid \mathsf{do}\ s\ \mathsf{while}\ p \\ & \mid \mathsf{return}\ e \mid \mathsf{assert}(p) \mid \mathsf{require}(p) \\ & \mid \mathsf{revert}(\cdots) \mid \mathsf{try}\ e\ (\mathsf{returns}\ (x))\ s_t\ \mathsf{catch}\ s_c \\ & \mid f(\overline{e}) \end{aligned}$$

where τ ranges over types (`uint`, `int`, `bool`, `address`, structs, arrays, mappings), lv denotes l-values (variables, fields, array/mapping elements), and p denotes boolean expressions.

B.2 Abstract Domain

Atomic abstract values:

- **Unsigned integers:** $\widehat{\mathbb{U}}_N = \{[\ell, u] \mid 0 \leq \ell \leq u \leq 2^N{-}1\} \cup \{\bot, \top_N\}$
- **Signed integers:** $\widehat{\mathbb{Z}}_N = \{[\ell, u] \mid -2^{N-1} \leq \ell \leq u \leq 2^{N-1}{-}1\} \cup \{\bot, \top_N^{\pm}\}$
- **Booleans:** $\widehat{\mathbb{B}} = \{\bot, \widehat{\mathsf{false}}, \widehat{\mathsf{true}}, \top\}$
- **Addresses:** $\widehat{\mathbb{A}} = \wp_{\text{fin}}(\mathsf{AddrID}) \cup \{\top\}$ (finite set of symbolic address identifiers)
- **Bytes:** $\widehat{\mathbb{BY}}_K = \{\bot, \top_K\}$ (symbolic/opaque)
- **Enums:** $\widehat{\mathsf{Enum}}(E) = \{[\ell, u] \mid 0 \leq \ell \leq u \leq |E| - 1\} \cup \{\bot, \top_E\}$

Composite values:

- **Structs:** $\widehat{\mathsf{Struct}}(C) = \prod_{f \in \mathsf{fields}(C)} \widehat{\mathsf{Val}}_f$ (pointwise order)
- **Arrays:** $\widehat{\mathsf{Arr}}(\tau) = (\hat{\ell}, \hat{d}, M)$ where $\hat{\ell} \in \widehat{\mathbb{U}}_{256}$ is length, $\hat{d}$ is default element, $M : \mathbb{N}_{\text{fin}} \rightharpoonup \widehat{\tau}$ stores observed indices
- **Mappings:** $\widehat{\mathsf{Map}}(\kappa \Rightarrow \tau) = (\hat{d}, M)$ with default $\hat{d}$ and finite map M for observed keys

Table 5: Concrete semantics (denotational)

Statement	**Meaning**
skip	$[\![\mathsf{skip}]\!](\sigma) = \mathsf{Norm}(\sigma)$
$s_1; s_2$	$[\![s_1; s_2]\!](\sigma) = \big([\![s_1]\!](\sigma)\big) \triangleright (\lambda\sigma'.\, [\![s_2]\!](\sigma'))$
$\tau\ x;$	$[\![\tau\ x;]\!](\sigma) = \mathsf{Norm}\big(\sigma[x \mapsto \mathrm{zero}_\tau]\big)$
$\tau\ x = e;$	$[\![\tau\ x = e;]\!](\sigma) = \mathsf{Norm}\big(\sigma[x \mapsto [\![e]\!]_\sigma]\big)$
$lv := e$	$[\![lv := e]\!](\sigma) = \mathsf{Norm}\big(\mathrm{write}(\sigma,\, \mathrm{loc}_\sigma(lv),\, [\![e]\!]_\sigma)\big)$
delete lv	$[\![\mathsf{delete}\ lv]\!](\sigma) = \mathsf{Norm}\big(\mathrm{write}(\sigma,\, \mathrm{loc}_\sigma(lv),\, \mathrm{zero}_{\tau(lv)})\big)$
if p then s_t else s_f	$[\![\cdot]\!](\sigma) = \begin{cases} [\![s_t]\!](\sigma) & \text{if } [\![p]\!]_\sigma = \mathsf{true}, \\ [\![s_f]\!](\sigma) & \text{if } [\![p]\!]_\sigma = \mathsf{false} \end{cases}$
while p do s	$F(H)(\sigma) = \begin{cases} \big([\![s]\!](\sigma)\big) \triangleright H & \text{if } [\![p]\!]_\sigma = \mathsf{true}, \\ \mathsf{Norm}(\sigma) & \text{if } [\![p]\!]_\sigma = \mathsf{false} \end{cases}$; $[\![\mathsf{while}\ p\ \mathsf{do}\ s]\!] = \mathrm{lfp}(F)$
for $init$; p; $incr$ do s	$F(H)(\sigma) = \begin{cases} \big([\![s]\!](\sigma)\big) \triangleright (\lambda\sigma'.\, [\![incr]\!](\sigma') \triangleright H) & \text{if } [\![p]\!]_\sigma = \mathsf{true}, \\ \mathsf{Norm}(\sigma) & \text{if } [\![p]\!]_\sigma = \mathsf{false} \end{cases}$; $[\![\mathsf{for}\ init; p; incr\ \mathsf{do}\ s]\!](\sigma) = [\![init]\!](\sigma) \triangleright (\lambda\sigma'.\, \mathrm{lfp}(F)(\sigma'))$
do s while p	$F(H)(\sigma) = [\![s]\!](\sigma) \triangleright (\lambda\sigma'.\, \begin{cases} H(\sigma') & \text{if } [\![p]\!]_{\sigma'} = \mathsf{true}, \\ \mathsf{Norm}(\sigma') & \text{if } [\![p]\!]_{\sigma'} = \mathsf{false} \end{cases})$; $[\![\mathsf{do}\ s\ \mathsf{while}\ p]\!] = \mathrm{lfp}(F)$
return e	$[\![\mathsf{return}\ e]\!](\sigma) = \mathsf{Ret}([\![e]\!]_\sigma,\, \sigma)$
assert(p), require(p)	$[\![\cdot]\!](\sigma) = \begin{cases} \mathsf{Norm}(\sigma) & \text{if } [\![p]\!]_\sigma = \mathsf{true}, \\ \mathsf{Abort} & \text{if } [\![p]\!]_\sigma = \mathsf{false} \end{cases}$
revert($\cdots$)	$[\![\mathsf{revert}(\cdots)]\!](\sigma) = \mathsf{Abort}$
try e (returns (x)) s_t catch s_c	$[\![\cdot]\!](\sigma) = \begin{cases} [\![s_t]\!](\sigma[x \mapsto v]) & \text{if call succeeds with } v, \\ [\![s_c]\!](\sigma) & \text{if call reverts} \end{cases}$
call($\overline{e}$)	Internal: parameter binding; external: unspecified

Standard interval domain operations (order, join, meet, widening, narrowing) apply to integer and enum domains.

B.3 Concrete Semantics

- **Variables:** Var = set of variable identifiers
- **Values:** Val includes:
  - Unsigned integers: $\mathbb{U}_N = \{0, 1, \ldots, 2^N - 1\}$
  - Signed integers: $\mathbb{Z}_N = \{-2^{N-1}, \ldots, 2^{N-1} - 1\}$
  - Booleans: $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$
  - Addresses: $\mathbb{A} = \mathsf{AddrID}$ (symbolic identifiers, e.g., msg.sender, symbolicAddress 1)

  - Composite values: structs, arrays, mappings with concrete elements

- **Stores:** $\sigma \in \Sigma = \mathsf{Var} \rightharpoonup \mathsf{Val}$

L-value resolution $\mathrm{loc}_\sigma(lv)$ and write $\mathrm{write}(\sigma, \ell, v)$ update the store. Expressions are pure: $[\![e]\!]_\sigma \in \mathsf{Val}$.
Array/mapping materialization: $\mathrm{loc}_\sigma(a[i])$ extends a up to i with defaults if needed; $\mathrm{loc}_\sigma(m[k])$ creates $m[k]$ lazily if absent.

B.4 Collecting Semantics

For abstraction, we lift concrete semantics to sets of states.
Collecting function semantics: Given a function f and a set of states $S \subseteq \Sigma$, the collecting semantics is:

$$\mathcal{S}[[f]](S) = \{\sigma' \mid \sigma \in S,\, (\sigma', v_{\text{out}}) \in \mathcal{S}[[f]](\sigma, v_{\text{in}}),\, v_{\text{in}} \in \mathsf{Val}\}$$

Reachable states: The set of all reachable states during contract execution forms the collecting semantics, serving as the basis for abstract interpretation.

B.5 Abstract Semantics (Denotational)

Our abstract semantics forms a Galois connection with the concrete semantics, ensuring soundness **?**. The abstraction function α and concretization function γ connect concrete and abstract domains, guaranteeing that abstract computations safely over-approximate concrete behaviors.

Abstract semantic domains:

- **Abstract values:** $\widehat{\mathsf{Val}}$ = union of atomic abstract values ($\widehat{\mathbb{U}}_N$, $\widehat{\mathbb{Z}}_N$, $\widehat{\mathbb{B}}$, $\widehat{\mathbb{A}}$, etc.) and composite abstract values ($\widehat{\mathsf{Struct}}$, $\widehat{\mathsf{Arr}}$, $\widehat{\mathsf{Map}}$) from §B.2
- **Abstract stores:** $\hat{\sigma} \in \widehat{\Sigma} = \mathsf{Var} \rightharpoonup \widehat{\mathsf{Val}}$

Expressions evaluate to $[\![e]\!]^\sharp_{\hat{\sigma}} \in \widehat{\mathsf{Val}}$.

Auxiliary functions:

- $\mathrm{refine}(\hat{\sigma}, p, b)$: narrows operands of p by interval meets
- $\widehat{\mathrm{write}}(\hat{\sigma}, lv, \hat{v})$: strong update if singleton index/key, weak update otherwise
- $\mathrm{joinRes}(r_1, r_2)$: componentwise join of abstract results

Table 6: Abstract semantics (denotational)

Statement	**Meaning**
skip	$[\![\mathsf{skip}]\!]^\sharp(\hat{\sigma}) = \widehat{\mathsf{Norm}}(\hat{\sigma})$
$s_1; s_2$	$[\![s_1; s_2]\!]^\sharp(\hat{\sigma}) = \big([\![s_1]\!]^\sharp(\hat{\sigma})\big) \triangleright^\sharp (\lambda\hat{\sigma}'.\, [\![s_2]\!]^\sharp(\hat{\sigma}'))$
$\tau\ x;$	$[\![\tau\ x;]\!]^\sharp(\hat{\sigma}) = \widehat{\mathsf{Norm}}\big(\hat{\sigma}[x \mapsto \hat{\mathrm{init}}(\tau)]\big)$
$\tau\ x = e;$	$[\![\tau\ x = e;]\!]^\sharp(\hat{\sigma}) = \widehat{\mathsf{Norm}}\big(\hat{\sigma}[x \mapsto \alpha_\tau([\![e]\!]^\sharp_{\hat{\sigma}})]\big)$
$lv := e$	$[\![lv := e]\!]^\sharp(\hat{\sigma}) = \widehat{\mathsf{Norm}}\big(\widehat{\mathrm{write}}(\hat{\sigma},\ lv,\ [\![e]\!]^\sharp_{\hat{\sigma}})\big)$
delete lv	$[\![\mathsf{delete}\ lv]\!]^\sharp(\hat{\sigma}) = \widehat{\mathsf{Norm}}\big(\widehat{\mathrm{write}}(\hat{\sigma},\ lv,\ \hat{\mathrm{zero}}_{\tau(lv)})\big)$
if p then s_t else s_f	$\hat{\sigma}_t = \mathrm{refine}(\hat{\sigma}, p, \mathsf{true})$, $\hat{\sigma}_f = \mathrm{refine}(\hat{\sigma}, p, \mathsf{false})$; $[\![\cdot]\!]^\sharp(\hat{\sigma}) = \mathrm{joinRes}\big([\![s_t]\!]^\sharp(\hat{\sigma}_t),\ [\![s_f]\!]^\sharp(\hat{\sigma}_f)\big)$
while p do s	$G^\sharp(H)(\hat{\sigma}) = \mathrm{joinRes}\big([\![s]\!]^\sharp(\mathrm{refine}(\hat{\sigma}, p, \mathsf{true})) \triangleright^\sharp H,\ \widehat{\mathsf{Norm}}(\mathrm{refine}(\hat{\sigma}, p, \mathsf{false}))\big)$; $[\![\mathsf{while}\ p\ \mathsf{do}\ s]\!]^\sharp = \mathrm{lfp}^\nabla(G^\sharp)$
for $init$; p; $incr$ do s	$G^\sharp(H)(\hat{\sigma}) = \mathrm{joinRes}\big([\![s]\!]^\sharp(\mathrm{refine}(\hat{\sigma}, p, \mathsf{true})) \triangleright^\sharp (\lambda\hat{\sigma}'.\, [\![incr]\!]^\sharp(\hat{\sigma}') \triangleright^\sharp H),\ \widehat{\mathsf{Norm}}(\mathrm{refine}(\hat{\sigma}, p, \mathsf{false}))\big)$; $[\![\mathsf{for}\ init; p; incr\ \mathsf{do}\ s]\!]^\sharp(\hat{\sigma}) = [\![init]\!]^\sharp(\hat{\sigma}) \triangleright^\sharp (\lambda\hat{\sigma}'.\, \mathrm{lfp}^\nabla(G^\sharp)(\hat{\sigma}'))$
do s while p	$G^\sharp(H)(\hat{\sigma}) = [\![s]\!]^\sharp(\hat{\sigma}) \triangleright^\sharp (\lambda\hat{\sigma}'.\, \mathrm{joinRes}\big(H(\mathrm{refine}(\hat{\sigma}', p, \mathsf{true})),\ \widehat{\mathsf{Norm}}(\mathrm{refine}(\hat{\sigma}', p, \mathsf{false}))\big))$; $[\![\mathsf{do}\ s\ \mathsf{while}\ p]\!]^\sharp = \mathrm{lfp}^\nabla(G^\sharp)$
return e	$[\![\mathsf{return}\ e]\!]^\sharp(\hat{\sigma}) = \widehat{\mathsf{Ret}}([\![e]\!]^\sharp_{\hat{\sigma}},\ \hat{\sigma})$
assert(p), require(p)	$\widehat{\mathsf{Norm}}(\mathrm{refine}(\hat{\sigma}, p, \mathsf{true}))$ if p must-hold; $\widehat{\mathsf{Abort}}$ if p must-fail; joinRes otherwise
revert($\cdots$)	$[\![\mathsf{revert}(\cdots)]\!]^\sharp(\hat{\sigma}) = \widehat{\mathsf{Abort}}$
try e (returns (x)) s_t catch s_c	$[\![\cdot]\!]^\sharp(\hat{\sigma}) = \mathrm{joinRes}\big([\![s_t]\!]^\sharp(\hat{\sigma}[x \mapsto \top]),\ [\![s_c]\!]^\sharp(\hat{\sigma})\big)$
call($\overline{e}$)	Internal: parameter binding; external: havoc footprint or $\widehat{\mathsf{Abort}}$