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, Incremental Analysis

1 Introduction

Blockchain technology has evolved from a simple cryptocurrency platform into a comprehensive ecosystem for decentralized applications. At the center of this evolution, Ethereum ranks second in market capitalization at over $460 billion (**?**). This ecosystem is powered by smart contracts written primarily in Solidity (**?**), the dominant language for contract development. As these contracts grow more complex and handle increasing value, ensuring their correctness becomes critical. This is especially critical because once deployed to the blockchain, contracts are immutable and cannot be easily fixed. Recently, many developers rely on large language models (LLMs) such as ChatGPT (**?**) or Llama (**?**) for code generation assistance. However, these tools cannot provide formal correctness guarantees. Therefore, developers must still rigorously understand program behavior at the statement level during editing.

Unfortunately, the debugging workflow for Solidity lacks the maturity of traditional programming environments. Even a single inspection requires full compilation, deployment, manual state initialization, and manual bytecode-level tracing. Moreover, once a transaction modifies contract state, reverting to previous conditions requires costly redeployment or manual reconstruction, making iterative debugging impractical. Tools like Remix IDE (**?**), Hardhat (**?**), and Foundry Forge (**?**) suffer from these fundamental limitations. A prior study found that 88.8% of Solidity developers described debugging as painful. Moreover, 69% attributed this pain to the absence of interactive source-level tooling (**?**). To the best of our knowledge, no existing research or tooling provides interactive feedback during Solidity code editing, a gap that this paper aims to fill.

This paper presents SolQDebug, a source-level interactive Solidity debugger powered by abstract interpretation (AI) with interval domain. Unlike traditional debugging workflows that require compilation and deployment, SolQDebug provides live feedback as developers type, helping them understand program behavior at the statement level. To achieve this goal, SolQDebug builds on two core ideas. First, it provides incremental analysis through interactive parsing and incremental control-flow graph (CFG) construction. As developers type each statement, the system extends the CFG and recomputes abstract states only for affected program points. Second, it enables annotation-guided exploration. Developers specify symbolic interval inputs directly in source code, and the interpreter uses these ranges to analyze multiple execution paths in a single pass.

To validate these design choices, we evaluate SolQDebug on 30 real-world functions from DAppSCAN (**?**). Our evaluation demonstrates millisecond-scale responsiveness and examines how annotation structure affects precision across common smart contract patterns. We also show how annotation-guided analysis addresses the precision challenges typically encountered in loops.

This paper makes the following contributions:

- We identify the main barriers to interactive Solidity debugging: (1) temporal inefficiency from compilation, deployment, and state initialization, and (2) iterative inefficiency from EVM constraints that prevent repeated execution with different states.

```
1  contract Example {
2      address public owner;
3      uint256 public totalSupply = 1000;
4      mapping(address => uint256) private balances;
5
6      modifier onlyOwner() {
7          require(msg.sender == owner, "not owner");
8          _;
9      }
10
11     function burn(uint256 amount) public onlyOwner {
12         uint256 bal = balances[msg.sender];
13         uint256 delta;
14         if (bal >= amount) {
15             balances[msg.sender] = bal - amount;
16             delta = amount;
17         }
18         else {
19             delta = 0;
20         }
21         totalSupply -= delta;
22     }
23 }
```

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

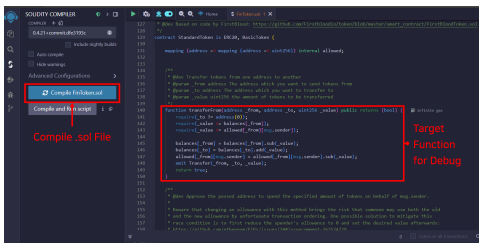
- We design an interactive parser with seven specialized entry rules and an incremental CFG engine that supports structural updates and syntactic recovery during editing.
- We introduce an annotation-guided abstract interpreter with adaptive widening thresholds that achieves precise loop analysis while maintaining termination guarantees.
- We evaluate SolQDebug on 30 real-world functions from DAppSCAN and demonstrate a 350× median speedup over Remix IDE. We analyze how annotation structure impacts precision in common smart contract patterns and loop convergence.

2 Background

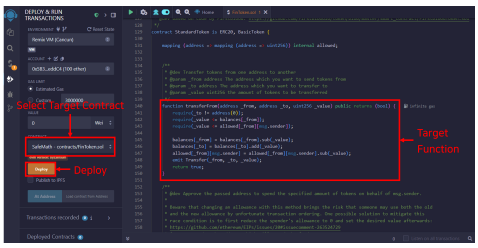
2.1 Structure of Solidity Smart Contract

Figure **??** illustrates the key structural elements of a Solidity smart contract. At the top level, Solidity programs may declare contracts, interfaces, and libraries. A contract is composed of variables and functions. We first describe the variable categories.

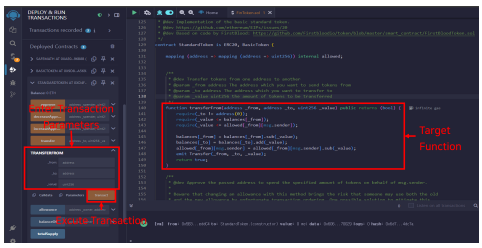
Variables are categorized by their scope and lifetime. Global variables such as `msg.sender` (appearing at line 7) and `block.timestamp` represent read-only EVM metadata provided implicitly by the runtime. State variables such as `owner`, `totalSupply`, and `balances` (lines 2–4) persist across transactions and provide the



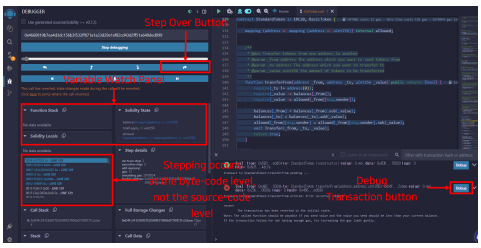
(a) Compile



(b) Deploy related contracts



(c) Send preparatory transactions



(d) Bytecode-level debugging

Fig. 2: Traditional Solidity debugging workflow

contract's permanent storage. Local variables such as `bal` and `delta` (lines 12–13) are scoped to a single function call and do not persist beyond execution.

Functions such as `burn` (lines 11–22) use control flow constructs including `if/else`, `while/for/do-while`, `break/continue`, and `return`. Functions can be augmented with modifiers. The `onlyOwner` modifier (lines 6–9) performs a precondition check before the function body executes. The placeholder underscore at line 8 marks where the original function body is inserted when the modifier is inlined. These structural elements define Solidity contracts.

2.2 Solidity Execution and Debugging Workflow

Unlike traditional programs, Solidity contracts require blockchain deployment before execution. Deployment is a one-time transaction that stores compiled bytecode on-chain and invokes the constructor. Once deployed, the bytecode becomes immutable.

After deployment, users interact with the contract by invoking public functions through blockchain transactions. Once a transaction is mined into a block, the EVM jumps to the designated entry point and executes the corresponding function sequentially. Crucially, the execution model is strictly state-based. State variables persist across transactions, forming a global storage that is shared by all function calls. Each transaction can read from and write to this persistent state, and these modifications become permanent once the transaction is confirmed. This state-based execution model introduces a critical constraint. Changes to contract state cannot be undone automatically and require costly external actions such as redeployment or manual state reconstruction.

These constraints directly impact the debugging process. To debug Solidity code, developers must follow a multi-stage workflow illustrated in Figure **??**. Before invoking the target function, the contract must be compiled and its state must be manually initialized through setup transactions to satisfy any preconditions. The target function is then invoked, and its execution is traced step by step at the bytecode level. This workflow interacts with the execution model constraints described above, introducing significant challenges in the debugging process.

2.3 Two Sources of Inefficiency in Solidity Debugging

The workflow described above reveals temporal and iterative inefficiencies. These issues stem from two orthogonal obstacles that make debugging Solidity programs significantly slower than traditional application development.

(1) Environmental disconnect between editor and execution engine causes temporal inefficiency. Unlike conventional IDEs such as PyCharm (**?**) or Visual Studio (**?**), where the source editor and execution engine run in the same process, Solidity development involves external coordination with a blockchain node at every stage. The multi-stage workflow described in the previous subsection introduces several seconds to minutes of latency per iteration, preventing the immediate feedback that developers expect when writing and testing code.

Given this workflow overhead, 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 because 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 EVM cause iterative inefficiency. The EVM's state-based execution model makes state modifications irreversible. This constraint fundamentally conflicts with iterative debugging, which requires repeatedly re-executing the same function under different conditions.

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. For example, consider a function that enforces a check on `_balances[account]`. Developers must 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 (**?**), 88.8% of Solidity practitioners reported frustration with current debugging workflows, with 69% attributing this to the lack of interactive, state-aware tooling. The next subsection outlines our proposed approach to address these two root causes.

2.4 Proposed Approach Overview

SolQDebug addresses the two root causes of Solidity's debugging bottleneck through a pair of complementary techniques. The first technique targets temporal inefficiency by eliminating blockchain round trips, while the second addresses iterative inefficiency by enabling state manipulation.

(1) Removing the multi-stage workflow to address temporal inefficiency. The traditional debugging workflow requires compilation, deployment, transaction-based state setup, and bytecode tracing. Each of these stages incurs significant latency. SolQDebug replaces this round trip by performing both parsing and abstract interpretation 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 incrementally.

Each parsed statement is inserted into an incremental CFG, and abstract interpretation resumes from the edit point. We use abstract interpretation with interval domain because it provides three key properties for edit-time debugging: guaranteed termination through widening, explainable results as variable ranges, and millisecond-scale responsiveness. This enables immediate feedback on code structure and control flow without compilation or chain interaction.

(2) Enabling repeated execution via batch annotations to address iterative inefficiency. The EVM does not support reverting to a prior state without redeploying the contract or replaying transactions. Both approaches disrupt iteration. SolQDebug introduces batch annotations as a mechanism for symbolic state injection. In essence, this reflects a core debugging activity, which involves 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 symbolic interval inputs. These inputs 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 interval input configuration from the analysis cycle while preserving the intuitive debugging process developers already follow.

3 The design of SolQDebug

SolQDebug is designed to analyze single-contract, single-transaction Solidity functions through incremental statement-by-statement processing. As developers write code, the system maintains an evolving abstract interpretation of the program, enabling immediate feedback on variable values and potential errors without requiring compilation or deployment. The following subsections detail the architecture (§3.1), provide a running example (§3.2), and explain the core mechanisms for parsing (§3.3) and incremental analysis (§3.4).

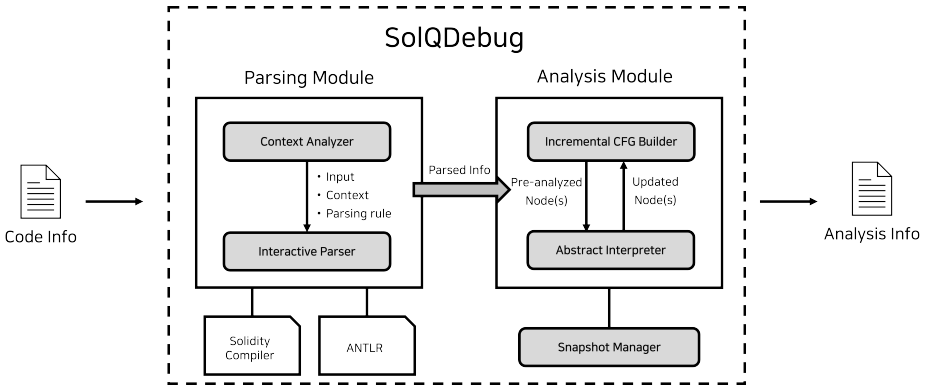


Fig. 3: SolQDebug architecture

3.1 System Architecture

Figure **??** illustrates the overall architecture of SolQDebug. The system accepts either single Solidity statements or batch debug annotations as input, processing them through two main modules before producing line-level output:

(1) Parsing Module. Each input passes through the Context Analyzer. For source code fragments, it identifies existing code structures when needed (e.g., an `if` statement for an `else` branch). For debug annotations, it identifies the target function and extracts its context. The Interactive Parser, built on ANTLR (**?**), applies an extended grammar with seven additional entry rules that enable parsing of partial constructs. To ensure correctness, the system validates the complete reconstructed source using the official Solidity compiler before proceeding to analysis, ensuring semantic consistency and rejecting malformed input early.

(2) Analysis Module. The Analysis Module operates through three coordinated components. The Incremental CFG Builder maintains a control-flow graph that evolves as statements are added, creating nodes and rewiring edges to reflect the updated structure. The Abstract Interpreter analyzes the updated CFG incrementally, reusing previous results and recomputing abstract values only for affected program points. The Snapshot Manager preserves and restores abstract memory states, ensuring that debug annotations can be modified and re-executed without side effects from previous runs.

Output. Following analysis, SolQDebug produces a line-level summary showing 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.

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`

3.2 Running Example

To illustrate how the proposed architecture functions in practice, we present a concrete example using the `burn` function from Figure **??**. As the developer incrementally constructs this function, the system (1) parses each input fragment, (2) updates the control-flow graph, and (3) performs abstract interpretation to compute variable intervals. Additionally, when batch debug annotations are present, the system re-analyzes the function using the annotated symbolic interval input as starting points. This example demonstrates these two key analysis modes: incremental source code analysis (§3.2.1) and batch annotation analysis (§3.2.2).

3.2.1 Incremental Source Code Analysis

Table **??** shows how the developer incrementally constructs the `burn` function through nine distinct input steps, each introducing a new code fragment.

SolQDebug accepts two kinds of code fragments as input:

- **Block fragment:** Includes contract and function definitions and control-flow constructs (if/else blocks, while loops). When the developer types an opening {, most editors auto-insert the closing }, so the complete block arrives at once (e.g., Steps 1, 4, and 7 in Table **??**).
- **Statement fragment:** Includes declarations, assignments, and expressions that end with semicolons. Each statement arrives individually as the developer completes typing, triggering immediate parsing and analysis (e.g., Steps 2, 3, 5, 6, 8, and 9 in Table **??**).

As the developer types each fragment, SolQDebug incrementally extends the CFG and recomputes abstract values only for affected program points. Figure **??** visualizes the CFG structure after Steps 1–8 have been integrated. To illustrate incremental analysis, we focus on Step 9 (`totalSupply -= delta;`). By this point, Steps 1–8 have been analyzed, and their results are already in the CFG. When Step 9 arrives, it processes the statement as follows:

1. The Interactive Parser recognizes `totalSupply -= delta;` as an assignment.

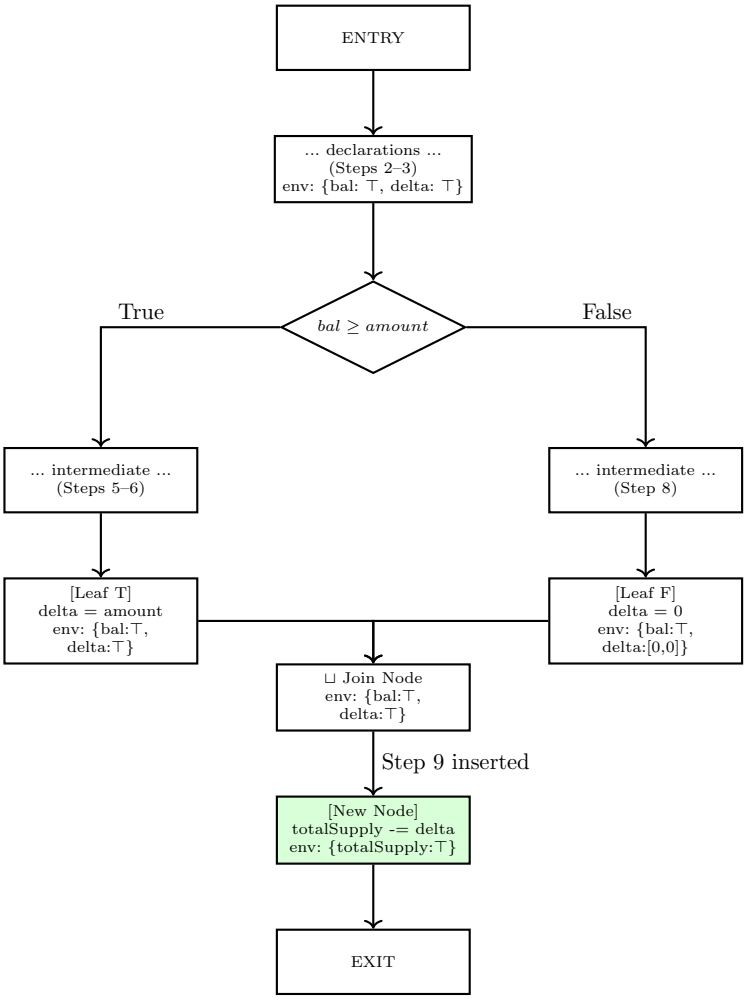


Fig. 4: CFG structure showing Step 9 insertion. Each statement occupies a separate basic node; intermediate nodes along each branch are omitted, showing only the leaf nodes before the join point. The join point node computes the least upper bound of environments from both branches

2. The Incremental CFG Builder 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 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.

```
1  uint256 public totalSupply = 1000;
2
3  function burn(uint256 amount) public onlyOwner {
4      // @Debugging BEGIN
5      // @StateVar balances[msg.sender] = [100,200]
6      // @LocalVar amount = [50,70]
7      // @Debugging END
8      uint256 bal = balances[msg.sender];
9      uint256 delta;
10     if (bal >= amount) {
11         balances[msg.sender] = bal - amount;
12         delta = amount;
13     }
14     else {
15         delta = 0;
16     }
17     totalSupply -= delta;
18 }
```

Fig. 5: Burn function with batch annotations

5. The Abstract Interpreter performs worklist-based reinterpretation, propagating the updated environment from the newly inserted node to all affected nodes in the CFG.

This reinterpretation ensures soundness by propagating each edit's effects to all affected nodes. Subsequent inputs can then safely reuse the computed abstract values without re-analyzing the entire program.

3.2.2 Batch Annotation Analysis

While incremental analysis supports the edit-test cycle, developers often need to explore how their program behaves under different testing scenarios. Batch annotations enable this by letting developers specify test inputs declaratively and obtain line-level results in a single run.

Listing **??** 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 a symbolic interval input, supporting both simple variables and nested accesses like `balances[msg.sender]`.

In this example, we annotate `balances[msg.sender]` with the interval $[100, 200]$ and `amount` with $[50, 70]$ to explore how the `burn` function behaves under different balance and amount scenarios. The analysis propagates these intervals through the conditional branches, computing `delta` as $[0, 150]$ at the join point and reducing `totalSupply` from $[1000, 1000]$ to $[850, 1000]$.

When SolQDebug processes a batch annotation block, it follows this pipeline:

Table 2: Interactive parser entry rules

Type	Entry Rule	Purpose
Primary	interactiveSourceUnit	Top-level declarations: functions, contracts, interfaces, libraries, state variables, pragmas, imports
	interactiveBlockUnit	Statements and control-flow skeletons inside function bodies
Continuation	interactiveEnumUnit	Enum member items added after the enum shell is defined
	interactiveStructUnit	Struct member declarations added after the struct shell is defined
	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

1. **Parse and validate.** Each annotation line is parsed, type-checked, and converted to the corresponding abstract domain (e.g., intervals for integers).
2. **Save state and overlay.** The unannotated abstract memory is saved, and the annotated values replace it for the analysis run.
3. **Single-pass analysis.** SolQDebug re-analyzes the pre-built CFG with the annotated environment.
4. **Restore previous state.** After analysis, the unannotated abstract memory is restored.

Details of these mechanisms appear in §3.3–§3.4.

3.3 Interactive Parser

The Interactive Parser extends the official Solidity language grammar (**?**) with specialized entry rules that accept partial code fragments during incremental editing. The parser defines eight specialized entry rules: seven for incremental code edits and one for batch annotations.

Table **??** shows the seven rules for Solidity constructs, divided into two categories that differ in how they interact with existing code:

- **Primary rules** parse constructs that stand alone, requiring no context from previous edits.
- **Continuation rules** require context from existing structures. They explicitly depend on previously parsed constructs (e.g., `interactiveIfElseUnit` requires a preceding `if` statement to attach the `else` branch).

This distinction maintains syntactic validity during incremental edits. SolQDebug validates that continuation rules have their required antecedent structure, ensuring that the resulting program structure remains well-formed at each step.

For concreteness, we refer to the burn function in Table **??**. When the developer types the function definition (Step 1), it is parsed by `interactiveSourceUnit`, which

creates a function with an empty body. Subsequent inputs within the function body are parsed by `interactiveBlockUnit`, which handles both statement fragments (Steps 2, 3, 5, 6, 8) and block fragments (Step 4). Step 7 illustrates how continuation rules work: the `else` block is parsed by `interactiveIfElseUnit`, which depends on the preceding `if` from Step 4 (see Appendix **??**).

Beyond these seven interactive rules for Solidity constructs, the parser includes a specialized `debugUnit` rule for batch annotations. 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

The complete grammar specification appears in Appendix **??**, with the full ANTLR4 implementation available at (**?**).

3.4 Incremental CFG Construction and Abstract Interpretation

Incremental CFG construction maintains a control-flow graph that evolves as developers insert new statements. Rather than rebuilding from scratch, our approach modifies the graph in place. We first describe the hierarchical organization of CFGs at the contract and function levels (§3.4.1). We then explain how individual statements are incrementally spliced into the function-level CFG (§3.4.2), how the insertion site is determined (§3.4.3), and how reinterpretation propagates abstract values only through affected regions (§3.4.4).

Our CFG consists of the following node types:

- ENTRY NODE: The unique entry point where execution begins. Contract-level CFGs have a contract entry, and function-level CFGs have a function entry.
- STATE VARIABLE NODE: A contract-level node that holds all state variable declarations. Since state variables have no branching logic, they are collected into a single node.
- 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 (§3.4.3).
- 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 unique normal exit point where execution terminates successfully. Contract-level CFGs have a contract exit, and function-level CFGs have a function exit.

3.4.1 CFG Hierarchy

SolQDebug organizes control flow at two levels. At the contract level, a ContractCFG sequences state variable initializations, the constructor, and function definitions. This ordering reflects how Solidity deploys contracts. State variables are initialized first, then the constructor executes and may modify those state variables. Function definitions receive the environment that results from constructor execution.

At the function level, each FunctionCFG represents control flow starting from a function's entry point. When a function declares modifiers, the function body is spliced into the modifier's control flow at the placeholder (_) position. When a function invokes another function, the callee's FunctionCFG is incorporated as part of the caller's control flow graph. The complete CFG hierarchy structures are shown in Appendix **??**.

3.4.2 Statement-Local, Incremental Construction

Having described the hierarchical organization of CFGs, we now focus on how individual statements are incrementally spliced into the function-level CFG. Each insertion operates at the CURRENT NODE, the insertion point for new statements. The following statement types create their corresponding CFG structures:

- **Simple statements** are spliced between the current node and its successors, creating a single BASIC NODE. This includes variable declarations, assignments, function calls, and expression statements. The environment flows sequentially through the new node, updated by the statement's semantics (Figure **??**).
- **If statement** creates a CONDITION NODE, true/false BASIC NODES, and an IF JOIN. The environment propagates from the current node to the condition, then branches to the true/false arms with the environment refined by the branch condition, and merges at the join node via the least upper bound operation (Figure **??**).
- **Else-if clause** replaces the preceding false branch with a new CONDITION NODE and its own join, which then connects to the outer IF JOIN. The environment flows through the else-if condition, branches to its true/false arms with refinement, merges at the else-if join, and continues to the outer join (Figure **??**).
- **Else clause** replaces the false branch node of the preceding `if`/`else if` with a new else node, which connects to the IF JOIN. The environment is refined by the negated condition and merges at the join (Figure **??**).
- **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. The environment propagates through ϕ to the condition, then branches to the body or exit with the environment refined by the loop guard, with the body flowing back to ϕ until convergence (Figure **??**).
- **Break statement** is inserted in the loop body with its outgoing edge redirected to the LOOP EXIT NODE (Figure **??**).

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

Simple Statements			Compound Statements		
Statement	**Maps Start Line To**	**Maps End Line To**	**Statement**	**Maps Start Line To**	**Maps End Line To**
`simple statement`	basic node	(none)	`if`	condition node	join node
			`else if`	condition node	join node
`break`	basic node	(none)	`else`	else node	join node
`continue`	basic node	(none)	`while`	condition node	exit node
`return`	return node	(none)			
`require`	condition node	(none)			

- **Continue statement** is inserted in the loop body with its outgoing edge redirected to the loop's ϕ node (Figure **??**).
- **Return statement** is inserted with its edge rewired to the function's unique RETURN EXIT, detaching original successors. The environment flows directly to the return exit (Figure **??**).
- **Require statement** creates a CONDITION NODE with the true edge connecting to a continuation node and the false edge pointing to the ERROR EXIT. The environment propagates with refinement based on the predicate to the continuation or error exit (Figure **??**).

These construction patterns enable SolQDebug to build the CFG incrementally as the user types each statement, with environment updates propagating only through affected nodes. Representative CFG construction patterns are provided in Appendix **??**. The complete implementation supports all Solidity statements (**?**).

3.4.3 Line-to-Node Indexing for Incremental Insertion

The key challenge in incremental CFG construction is determining where to insert each new node. 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 CFG edges encode control flow rather than source positions, we cannot determine where an edit belongs without additional context. To enable incremental insertion, we maintain a line-to-node index during construction. Table **??** summarizes how statement types map lines to nodes.

`Simple statements` index only their start line. Most statements map to a basic node, though return maps to a return node and require maps to a condition node. Compound statements index both start and end lines. Conditionals (`if` and `else if`) map their start line to a condition node and end line to a join node, while `else` maps its start line to an else node and end line to the join node. Loops (`while`) map their start line to a condition node and end line to an exit node.

Algorithm 1 Dependent-Context Insertion

```
1: Input: CFG G, line-to-node index M, current line L
2: Output: Condition node c
3: function GetBranchContext(G, M, L)
4:     queue ← FindJoinNode(M, L)              ▷ find join node at or before line L
5:     if queue = ∅ then
6:         error "No join node found at or before line L"
7:     end if
8:     visited ← ∅
9:     while queue ≠ ∅ do                      ▷ BFS through predecessors
10:        n ← Dequeue(queue)
11:        if n ∈ visited then continue
12:        end if
13:        visited ← visited ∪ {n}
14:        if IsCond(G, n) and CondType(n) ∈ {if, else_if} then
15:            return n
16:        end if
17:        for p ∈ Pred(G, n) do
18:            if p ∉ visited then
19:                Enqueue(queue, p)
20:            end if
21:        end for
22:    end while
23:    error "No matching condition node found"
24: end function
```

This indexing scheme enables Algorithms **??** and **??** to locate insertion sites efficiently. These algorithms handle the representative statement types shown in Table **??**. Both algorithms rely solely on the line-to-node index without mutating the graph.

The choice between these algorithms depends on whether the statement can exist independently:

- **Dependent contexts:** `else`/`else if` must attach to a preceding `if`/`else if` condition. Algorithm **??** traverses CFG predecessors to find the condition node.
- **Independent contexts:** All other statements can exist independently. Algorithm **??** uses a successor-first strategy, leveraging the fact that successors are already inserted and have complete CFG structure.

Algorithm ??: 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:

- **Initialization (Lines 4–8):** Uses the line-to-node index M to retrieve the join node at or before line L, initializing the Breadth-First Search (BFS) queue. This node serves as the starting point for backward traversal to find the preceding conditional.

Algorithm 2 Independent-Context Insertion

```
1: Input: CFG G, line-to-node index M, insertion position at line L
2: Output: Insertion-site node A
3: function GetInsertionSite(G, M, L)
4:     s ← FindPostNode(M, L)                          ▷ find first node after line L
5:     if IsLoopExit(G, s) or IsJoin(G, s) then        ▷ closing a loop or selection
6:         n ← FindPreviousNode(M, L)                  ▷ condition if exists, else last node
7:         if IsCond(G, n) then
8:             return BranchBlock(n, true)             ▷ insert in TRUE branch
9:         else
10:            return n
11:        end if
12:    else                                            ▷ basic successor
13:        pred ← Pred(G, s)
14:        if |pred| = 1 then
15:            return the unique element of pred
16:        else
17:            error "Basic successor must have exactly 1 predecessor"
18:        end if
19:    end if
20: end function
```

If no node is found, the dependent context is invalid (Lines 5–7). The visited set is initialized to track explored nodes (Line 8).

- **BFS traversal (Lines 9–22):** 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.
- **Error handling (Line 23):** Reports an error if no matching condition is found.

Algorithm ??: Independent-Context Insertion. For independent contexts, which include all statements except `else`/`else if`, we use a successor-first strategy. By 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:

- **Find post node (Line 4):** FindPostNode(M, L) retrieves the first node after line L from the line-to-node index.
- **Loop-exit/join case (Lines 5–11):** If the post node s is a loop-exit or join node, we search backward using FindPreviousNode(M, L) to find the previous node. This returns a condition node if present (loop condition node 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 (Line 8).
  - Otherwise, we return the node itself (Line 10).

- **Basic post node (Lines 12–19):** Otherwise, s is a basic statement node (Line 12). We retrieve its CFG predecessors and verify there is exactly one (Lines 13–15), which must hold by construction since conditionals are created with their join nodes and loops are created with their exit nodes. Any other count indicates a malformed CFG (Lines 16–18).

3.4.4 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, new nodes are spliced into the CFG. Incremental reinterpretation then propagates updates from these insertion points as seed nodes. While for debug annotations, SolQDebug performs full interpretation from the function entry node, ensuring all inspection points receive complete abstract states. Algorithm **??** performs incremental interpretation by propagating abstract states through a worklist-based dataflow analysis. When encountering loop condition nodes, it delegates to Algorithm **??** for loop fixpoint computation.

Algorithm ??: Incremental Interpretation.

- **Worklist and output cache initialization (Line 4):** Initialize empty worklist wl, in-queue set inq to track enqueued nodes, and output map out to store previous node outputs for change detection.
- **Seed node initialization (Lines 5–9):** Enqueue all seed nodes $s \in S$, which mark insertion points for incremental edits or function entry for batch annotations. Sink nodes (exit, error, return) with no successors are filtered out.
- **Worklist iteration (Lines 10–31):** The while loop processes nodes until the worklist is empty. Each iteration performs the following steps for the dequeued node n:
- **Incoming environment computation (Lines 11–12):** Compute incoming environment $\hat{\sigma}_{\text{in}}$ by joining outputs from all predecessors. For condition nodes, RefineByCondition narrows operand intervals based on the edge truth label (true/false branch), pruning infeasible paths.
- **Loop condition node delegation (Lines 13–21):** When encountering a loop condition node, delegate to Algorithm **??** to compute the loop fixpoint. After fixpoint converges, enqueue the loop-exit node's successors to continue analysis beyond the loop.
- **Transfer function (Line 22):** 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.
- **Change detection and propagation (Lines 23–30):** Compare $\hat{\sigma}_{\text{out}}$ with the previously stored output $out[n]$. Only if changed, update node environment and the stored output, then enqueue successors. This ensures fixpoint termination by stopping propagation when environments stabilize.

Algorithm **??** computes loop fixpoints using annotation-aware adaptive widening. The key innovation is that EstimateIterations analyzes loop conditions to compute an adaptive threshold τ that defers widening. When debug annotations provide precise

Algorithm 3 Incremental Interpretation

```
1: Input: CFG G, seed set S
2: Output: Environments updated along forward-reachable paths from S
3: function IncrementalInterpretation(G, S)
4:     wl ← ⟨⟩;    inq ← ∅;    out ← output map
5:     for all s ∈ S do
6:         if ¬IsSink(G, s) ∧ s ∉ inq then
7:             wl.enqueue(s);    inq ← inq ∪ {s}
8:         end if
9:     end for
10:    while wl ≠ ⟨⟩ do
11:        n ← wl.dequeue();    inq ← inq \ {n}
12:        σ̂_in ← ⨆_{p∈Pred(G,n)} RefineByCondition(p, n)    ▷ join predecessors with path refinement
13:        if IsLoopHeader(G, n) then
14:            exit ← Fixpoint(G, n)          ▷ compute loop fixpoint (Algorithm ??)
15:            for all s ∈ Succ(G, exit) do
16:                if ¬IsSink(G, s) ∧ s ∉ inq then
17:                    wl.enqueue(s);    inq ← inq ∪ {s}
18:                end if
19:            end for
20:            continue
21:        end if
22:        σ̂_out ← Transfer(n, σ̂_in)
23:        if σ̂_out ≠ out[n] then
24:            (n) ← σ̂_out;    out[n] ← σ̂_out
25:            for all s ∈ Succ(G, n) do
26:                if ¬IsSink(G, s) ∧ s ∉ inq then
27:                    wl.enqueue(s);    inq ← inq ∪ {s}
28:                end if
29:            end for
30:        end if
31:    end while
32: end function
```

interval bounds for condition operands such as array lengths or parameter values, the analyzer raises τ to delay widening and preserve precision. The fixpoint iteration terminates when all variables converge, meaning node outputs equal their previously stored values, ensuring soundness.

Algorithm ??: Loop Fixpoint with Adaptive Widening.

- **Identify loop nodes and pre-loop environment (Line 4):** Collect all nodes within the loop body. Compute *start* by joining environments from all loop-entry predecessors, specifically those from outside the loop rather than from within the loop body itself.

Algorithm 4 Loop Fixpoint with Adaptive Widening

1: **Input:** CFG G, loop condition node h
2: **Output:** Loop exit node with converged environment
3: **function** LoopFixpoint(G, h)
4: $L \leftarrow \textsc{LoopNodes}(G, h)$; $start \leftarrow \bigsqcup\{(p) \mid p \in \textsc{Pred}(G, h) \setminus L\}$
5: $\tau \leftarrow \textsc{EstimateIterations}(h, start)$ ▷ annotation-aware threshold
6: $vis[\cdot] \leftarrow 0$; $in[h] \leftarrow start$
7: // **Widening phase**
8: $wl \leftarrow \langle h \rangle$
9: **while** $wl \neq \langle\rangle$ **do**
10: $n \leftarrow wl.\mathsf{dequeue}()$; $vis[n] \leftarrow vis[n] + 1$
11: **if** $in[n] = \bot$ **then** ▷ compute input if not initialized
12: $in[n] \leftarrow \bigsqcup_{p \in \textsc{Pred}(G,n) \cap L} \textsc{RefineByCondition}(p, n)$
13: **end if**
14: $\hat{o}_{\text{raw}} \leftarrow \textsc{Transfer}(n, in[n])$
15: **if** $\textsc{IsFixpointEvalNode}(n) \ \wedge \ vis[n] > \tau$ **then**
16: $\hat{o} \leftarrow \textsc{Widen}(out[n], \hat{o}_{\text{raw}})$ ▷ widen at fixpoint eval node after τ visits
17: **else**
18: $\hat{o} \leftarrow out[n] \sqcup \hat{o}_{\text{raw}}$
19: **end if**
20: **if** $\hat{o} = out[n]$ **then**
21: $out[n] \leftarrow \hat{o}$; **continue** ▷ fixpoint reached, skip propagation
22: **end if**
23: $out[n] \leftarrow \hat{o}$
24: **for all** $s \in \textsc{Succ}(G, n) \cap L$ **do**
25: $wl.\mathsf{enqueue}(s)$
26: **end for**
27: **end while**
28: $\textsc{NarrowingPhase}(L)$ ▷ standard descending iteration
29: $exit \leftarrow \textsc{FindLoopExit}(G, h)$
30: $(exit) \leftarrow \bigsqcup_{p \in \textsc{Pred}(G,exit)} \textsc{RefineByCondition}(p, exit)$ ▷ join with false-branch pruning
31: **return** $exit$
32: **end function**

- **EstimateIterations (Line 5):** Evaluate both operands of the loop condition in the pre-loop environment *start*. For comparison operators ($<, \leq, >, \geq$), compute adaptive threshold τ as follows:
  - When both operands evaluate to concrete constant intervals, τ is computed as the difference between operand bounds. For example, given `i < array.length` where `i = [0,0]` and `array.length = [10,10]`, $\tau = 10 - 0 = 10$. The operands may be variables or complex expressions, but both must reduce to constant intervals through annotation-provided values.

  - When either operand cannot be reduced to a constant interval, τ defaults to 1, allowing one iteration to initialize node environments before widening is applied.

- **Initialization (Line 6):** Initialize visit counts *vis* to track iterations and node input environments *in*, setting $in[h] \leftarrow start$ for the loop condition node.
- **Widening phase (Lines 9–27):** Perform fixpoint iteration with worklist-based propagation. Each iteration performs the following steps:
  - Dequeue node and increment visit count (Line 10); compute incoming environment if not initialized (Lines 11–13).
  - Apply transfer function to compute raw output (Line 14).
  - Apply adaptive widening (Lines 15–19): use widening operator if visit count exceeds τ (Line 16), otherwise use join (Line 18).
  - Check for convergence (Lines 20–22): if output unchanged, skip propagation (Line 21); otherwise, store output (Line 23) and enqueue successors (Lines 24–26).
- **Narrowing phase and exit node (Lines 28–31):** Apply standard narrowing using descending iteration with narrowing operator at fixpoint evaluation nodes to refine over-approximations from widening (Line 28). Then compute the exit node environment (Lines 29–30) by joining predecessor outputs where the loop condition node's false-branch is refined by the negated loop condition. Return the exit node with its computed environment (Line 31).

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

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 – Impact of Annotation Patterns on Precision**: In complex arithmetic operations involving multiplication and 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 single-contract, single-transaction functions to validate our contributions of incremental CFG construction and annotation-guided abstract interpretation, with multi-contract analysis as future work.

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 3.10 and operates directly on Solidity source code without requiring compilation or deployment, using the ANTLR4-based parser described in Section 3.

Dataset Collection. We derive our dataset from DAppSCAN (**?**), 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 such as accessing variables or invoking functions from other contracts, as our analysis focuses on single-contract scenarios, and (2) excluding logic-free functions that contain 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 manually.
- **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 **??** 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

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.

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` (**?**), a Selenium (**?**)-based automation framework that programmatically drives the Remix web 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

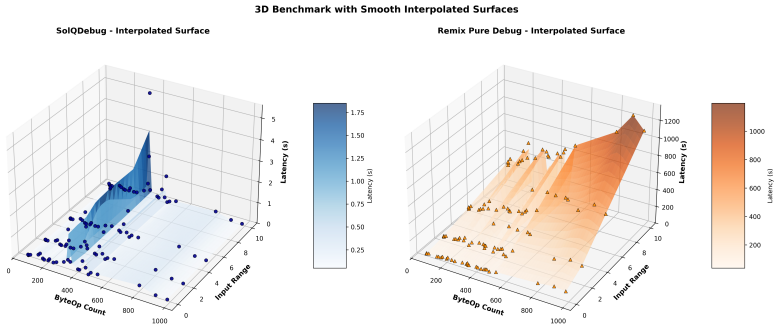


Fig. 6: Debugging latency comparison between Remix and SolQDebug across varying ByteOp counts and test-case widths Δ. SolQDebug maintains consistent sub-second latency regardless of function complexity or symbolic interval input width, while Remix's latency scales linearly with both dimensions.

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 width of each symbolic interval input in debug annotations. In contrast, Remix must 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 **??** 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.

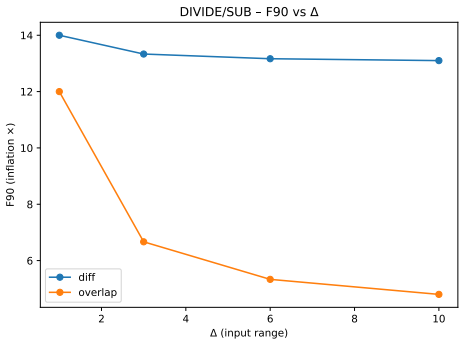


Fig. 7: F90 (90th percentile of interval inflation) for `Lock::pending` under OVERLAP and DIFF annotation patterns. As input width (Δ) increases, OVERLAP achieves progressively tighter precision (F90: 12.0 $\rightarrow$ 4.8), while DIFF maintains near-constant inflation (F90 $\approx$ 13–14).

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 use 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 **??**, 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 use 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 **??**): `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. For example, providing annotations `// @StateVar actionBuilders = arrayAddress[1,2,3];` and `// @LocalVar index = [3,3];` yields `additionalCount = 3 - 3 + 1 = 1`, allowing the analyzer to set $\tau = 1$ and achieve precise convergence with the final result `i = [0,1]`. 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. The annotation `// @StateVar _stablecoinList = arrayAddress[2,3,4];` yields `stablecoinListLength = 3`, so the loop bound becomes `3 - 1 = 2`, setting $\tau = 2$ and achieving precise convergence with `i = [0,2]`.

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. For example, providing annotations `// @StateVar depositsOf[_account][0].amount = [100,200];`, `// @StateVar depositsOf[_account][1].amount = [200,300];`, and `// @StateVar depositsOf[_account][2].amount = [400,500];` yields a loop bound of 3, setting $\tau = 3$. The analyzer correctly infers `i = [0,3]` after the loop. However, SolQDebug conservatively models uninitialized variables as $\top$ (unknown), causing `total` to remain $\top = [0, 2^{256} - 1]$ regardless of the precise loop bound.

This pattern represents a developer-fixable limitation. Explicitly initializing `total = 0` would enable precise tracking (yielding `total = [700,1000]` for the annotated example). 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. For example, providing annotations `// @StateVar _tokens = array [1,2,3];` and `// @LocalVar tokenId = [4,4];` yields `tokenCount = 3`, setting $\tau = 3$. The analyzer correctly infers `tokenIndex = [0,3]` after the loop. However, inside the loop, `resultIndex` increments conditionally (`if (tokenItemId != tokenId) resultIndex++`) 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 with Interval Domain for Debugging

In this work, debugging refers to a developer-led, interactive exploration activity that occurs during code editing, before deployment. Developers vary debug annotations and immediately observe program behaviors 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.

We use AI with interval domain because it uniquely satisfies these debugging requirements, offering three key properties:

- **Termination.** AI enforces convergence through widening at loops and joins at control-flow merges, avoiding path explosion in symbolic execution.
- **Explainability.** 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.** Unlike relational domains (e.g., zones, polyhedra) that track variable correlations, intervals require only basic arithmetic on bounds, enabling millisecond-scale updates that align with the edit cycle.

Although AI's precision is inherently conservative, we address this limitation while preserving edit-time responsiveness by providing developers with techniques to control precision through annotation design. Our evaluation shows that such techniques can reduce imprecision in practice (cf. RQ2, RQ3).

5.2 Current Limitations

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 may require more expressive abstract domains such as 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 (**?**) 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 (**?**) is a Node.js-based framework that couples the Solidity compiler with an Ethereum runtime. Its Hardhat Debug plug-in provides step-by-step debugging of transaction execution within Visual Studio Code. Foundry Forge (**?**) 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 (**?**) is a Visual Studio Code extension that performs runtime debugging over concrete transactions and integrates with Hardhat.

In summary, these debuggers operate on compiled artifacts or post-deployment traces. 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

Security analysis of smart contracts can be categorized into four main families of techniques. Static analysis tools reason over source or bytecode without running the contract. Representative systems include rule-based analyzers such as Securify and Slither (**??**), symbolic-execution tools like Mythril (**?**), knowledge-graph approaches such as Solidet (**?**), and CFG refinement techniques as in Ethersolve (**?**). Dynamic testing and fuzzing exercise deployed or locally simulated contracts to uncover faults and security issues. ContractFuzzer mutates Application Binary Interface (ABI)-level inputs (**?**), Echidna brings property-based fuzzing into developer workflows (**?**), sFuzz adapts scheduling for higher coverage (**?**), TransRacer finds transaction-ordering races (**?**), and Ityfuzz leverages snapshotting to decouple executions from chain nondeterminism (**?**). Formal verification aims to prove safety properties or refute counterexamples at compile time. Examples include ZEUS, VeriSmart, and SmartPulse (**???**). Finally, learning-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 (**???**).

These approaches have substantially advanced vulnerability detection and property checking for smart contracts. However, they are designed to find security issues rather

than to support interactive debugging during development. In contrast, SolQDebug is a debugging tool that helps developers understand code behavior interactively during editing.

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 (**??**). The first uses the Pos domain to construct a theoretical model for taint (information-flow) analysis (**?**), while the second uses the Difference-Bound Matrix (DBM) domain to generate state invariants and detect re-entrancy vulnerabilities, including the DAO attack (**??**). However, both approaches operate on fully written contracts and provide no support for line-by-line interpretation or developer interaction within an IDE.

SolQDebug builds upon these AI frameworks but targets interactive debugging rather than invariant generation or information flow analysis. It incrementally updates the control-flow graph and abstract state during editing. Developers use debug annotations to guide interval analysis. This approach provides edit-time feedback without requiring complete contracts.

6.4 Interactive Abstract Interpretation for Traditional Languages

Beyond Solidity, the broader programming languages community has increasingly focused on making abstract interpretation interactive. Recent work integrates static analysis directly into IDEs for C, Java, and other traditional languages, delivering live feedback during editing (**?????**). While these systems share SolQDebug's goal of edit-time responsiveness, they differ in language semantics, domain requirements, and input mechanisms. **?** proposed demanded abstract interpretation, which incrementally rebuilds only the analysis nodes touched by an edit. A follow-up **?** generalized this to procedure summaries, enabling inter-procedural reuse. **?** extended Goblint with incremental support for multithreaded C, selectively recomputing only genuinely affected facts and maintaining IDE-level responsiveness. **?** introduced IntraJ, a Language Server Protocol (LSP)-integrated analyzer for Java 11 that computes only the Abstract Syntax Tree (AST) and data-flow facts needed for the current view, keeping feedback under 100 ms. **?** 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.

While these frameworks demonstrate the feasibility of interactive abstract interpretation in traditional languages, SolQDebug represents the first application of this paradigm to Solidity. More fundamentally, it introduces an annotation-guided approach where developers specify debug annotations directly in source code, and SolQDebug uses them to guide interval analysis. This design bridges the gap between Solidity development and the interactive tooling common in traditional programming environments.

7 Conclusion

This paper presents SolQDebug, the first interactive source-level debugger for Solidity that eliminates the traditional compile-deploy-debug cycle. Through interactive parsing, dynamic control-flow analysis, and annotation-guided abstract interpretation, SolQDebug provides millisecond-scale feedback directly within the editor.

Evaluation on 30 real-world DAppSCAN functions demonstrates a 350× median speedup over Remix IDE (0.15s vs. 53.0s). Our analysis shows that annotation-guided interpretation enables precise analysis across diverse execution paths while maintaining soundness guarantees. These results demonstrate that SolQDebug effectively bridges the interactivity gap in Solidity debugging, bringing the development experience closer to modern IDEs and enhancing software quality in the blockchain ecosystem.

Future work includes extending support to inter-contract and multi-transaction contexts, conducting user studies to assess practical adoption, and applying EOF-based analysis to support inter-contract debugging when source code is unavailable.

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 (**?**), 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

CoinMarketCap: https://coinmarketcap.com/ (2025). Accessed November 2025

Chen, X., et al.: Characterizing smart contract evolution. ACM Transactions on Software Engineering and Methodology (2025). https://doi.org/10.1145/3719004

Chimdyalwar, B.: Fast and precise interval analysis on industry code. In: 2024 IEEE 35th International Symposium on Software Reliability Engineering Workshops (ISSREW) (2024). https://doi.org/10.1109/ISSREW63542.2024.00049

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). https://doi.org/10.1145/512950.512973

Erhard, J., et al.: Interactive abstract interpretation: reanalyzing multithreaded C programs for cheap. International Journal on Software Tools for Technology Transfer (2024). https://doi.org/10.1007/s10009-024-00768-9

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). https://doi.org/10.1145/3395363.3404366

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). https://doi.org/10.1109/Blockchain62396.2024.00038

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). https://doi.org/10.1109/TR.2023.3233999

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). https://doi.org/10.1145/3238147.3238177

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). https://doi.org/10.14722/ndss.2018.23082

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). https://doi.org/10.1145/3611643.3616281

Mehar, M.I., et al.: Understanding a revolutionary and flawed grand experiment in blockchain: the DAO attack. Journal of Cases on Information Technology (2019). https://doi.org/10.4018/JCIT.2019010102

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). https://doi.org/10.1145/3377811.3380334

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). https://doi.org/10.1016/j.jss.2023.111653

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). https://doi.org/10.1007/s10009-024-00771-0

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). https://doi.org/10.1145/3597926.3598059

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). https://doi.org/10.1109/SP40000.2020.00032

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

SolQDebug Complete Implementation: https://github.com/iwwyou/SolDebug/tree/main (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). https://doi.org/10.1145/3453483.3454044

Stein, B., Chang, B.-Y.E., Sridharan, M.: Interactive abstract interpretation with demanded summarization. ACM Transactions on Programming Languages and Systems (2024). https://doi.org/10.1145/3648484

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). https://doi.org/10.1109/SP40001.2021.00085

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). https://doi.org/10.1145/3243734.3243780

Feist, J., Grieco, G., Groce, A.: 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). https://doi.org/10.1109/WETSEB.2019.00008

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). https://doi.org/10.1109/ISSRE52982.2021.00047

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). https://doi.org/10.1109/BigData55660.2022.10020730

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). https://doi.org/10.1109/ISSRE59848.2023.00030

Zheng, Z., et al.: Dappscan: building large-scale datasets for smart contract weaknesses in dApp projects. IEEE Transactions on Software Engineering (2024). https://doi.org/10.1109/TSE.2024.3383422

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). https://doi.org/10.1109/JIOT.2022.3196269

Zou, W., et al.: Smart contract development: challenges and opportunities. IEEE Transactions on Software Engineering (2019). https://doi.org/10.1109/TSE.2019.2942301

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 Incremental CFG Construction Patterns

This appendix provides the complete CFG construction patterns for SolQDebug's incremental CFG builder. The patterns show how each statement type is incrementally spliced into the existing control-flow graph.

B.1 CFG Hierarchy

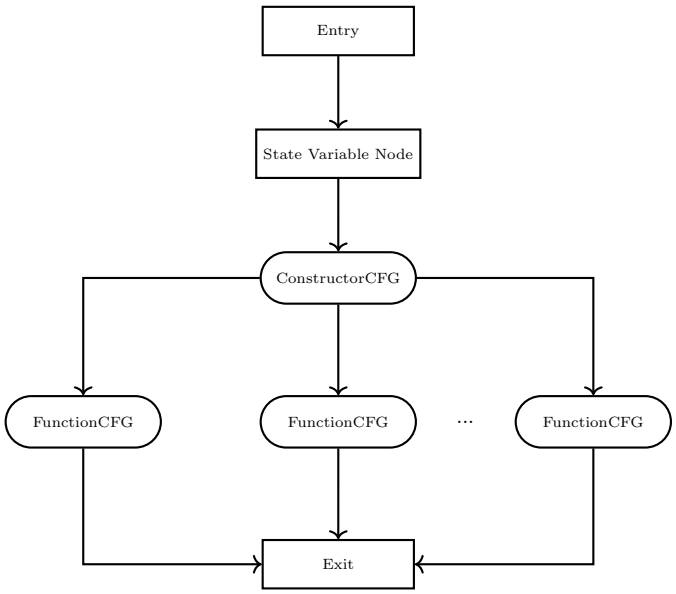


Fig. 8: Structure of ContractCFG. The contract-level CFG sequences state variable initialization, constructor execution, and all function definitions.

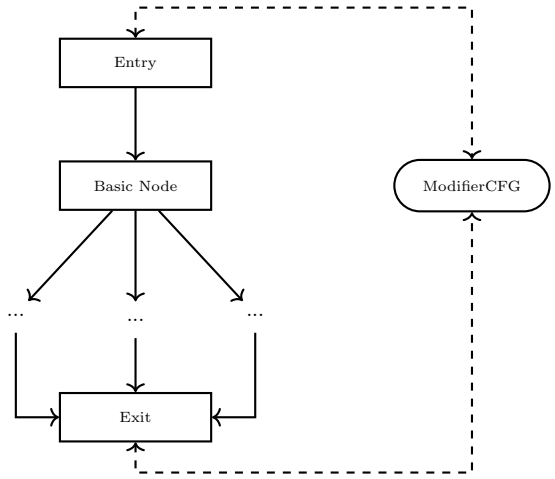


Fig. 9: Structure of FunctionCFG. Each function maintains its own control-flow graph. When modifiers are present, control flows from the function entry to the modifier, and returns from the modifier's placeholder (_) back to the function body, eventually reaching the function exit.

B.2 Statement-Level Construction Patterns

The following patterns show how each statement type is incrementally spliced into the existing control-flow graph. All patterns assume an initial state with a current node connected to successors (...), as illustrated in the simple statement example.

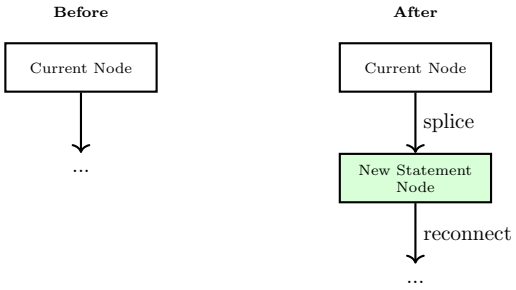


Fig. 10: Simple statement insertion: single node spliced between current node and successors

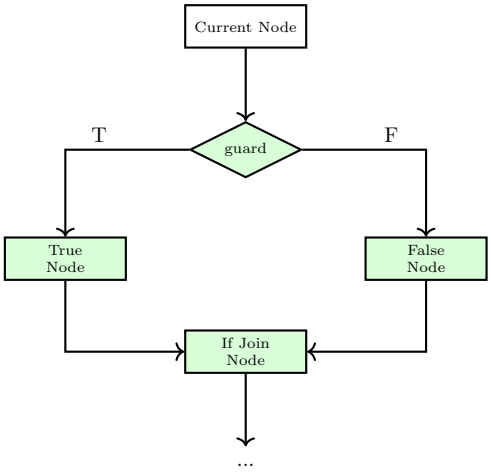


Fig. 11: If statement insertion. The builder creates a CONDITION NODE, two nodes for true/false arms, and an IF JOIN

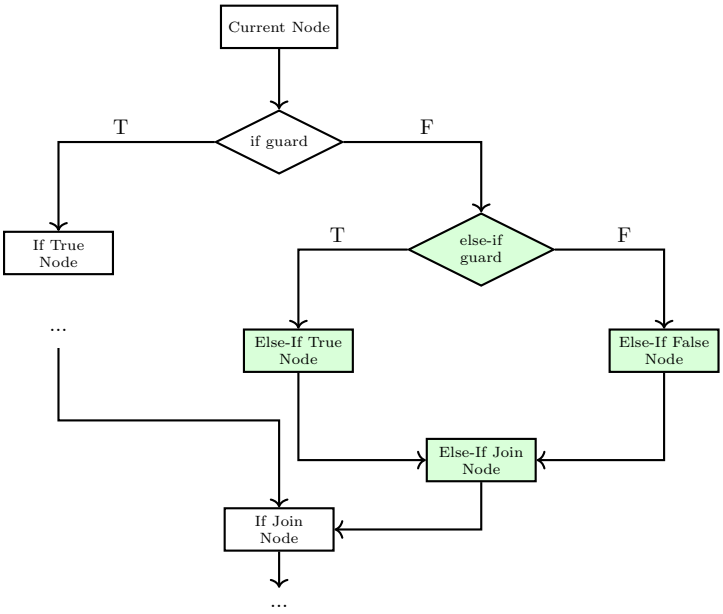


Fig. 12: Else-if statement insertion. The builder replaces the false arm with a new CONDITION NODE, two nodes, and an ELSE-IF JOIN

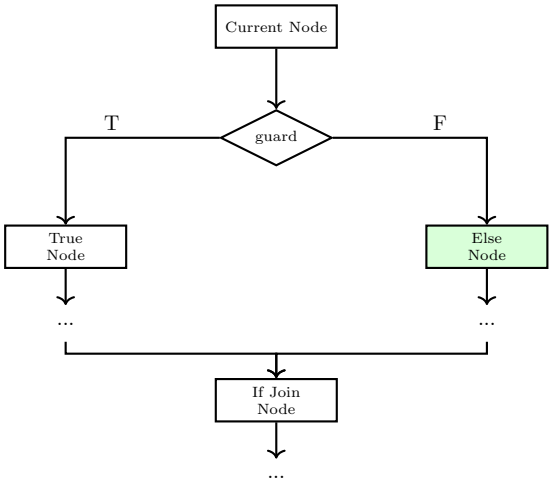


Fig. 13: Else statement insertion. The builder replaces the false branch node with a new else node, connecting to the IF JOIN

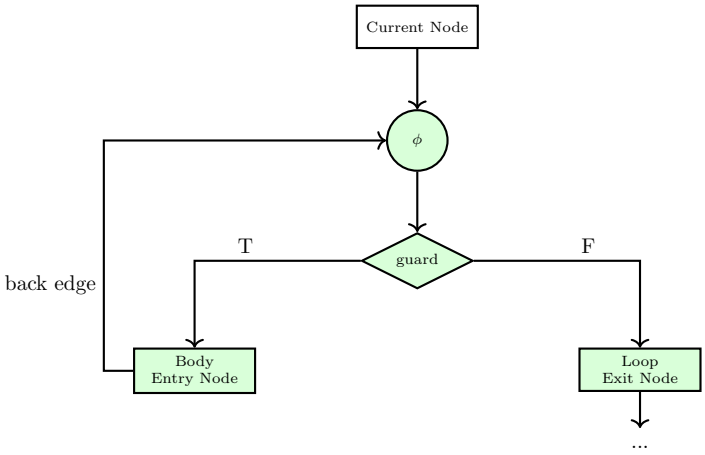


Fig. 14: While loop insertion. The builder creates a FIXPOINT EVALUATION NODE ϕ, a CONDITION NODE, a loop body node, and a LOOP EXIT NODE

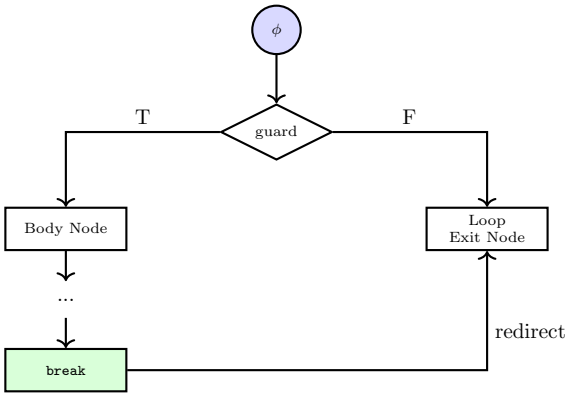


Fig. 15: Break statement insertion. The `break` node's outgoing edge is redirected to the LOOP EXIT NODE

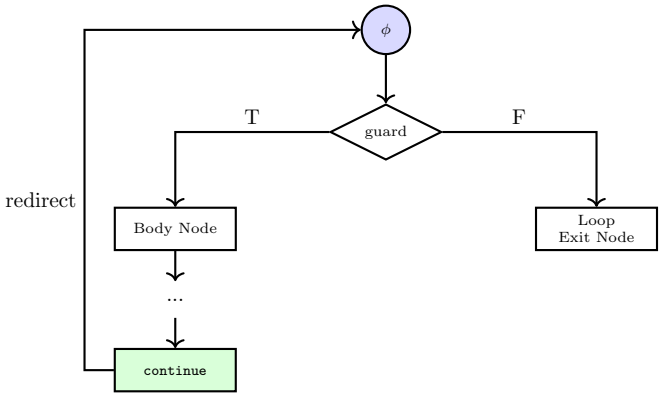


Fig. 16: Continue statement insertion. The `continue` node's outgoing edge is redirected to the loop's FIXPOINT EVALUATION NODE ϕ

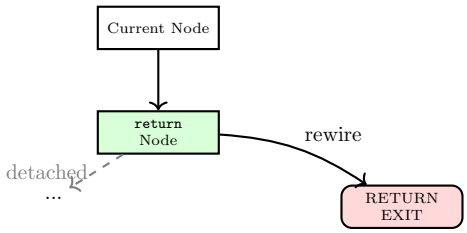


Fig. 17: Return statement insertion. The `return` node is rewired to the function's unique RETURN EXIT

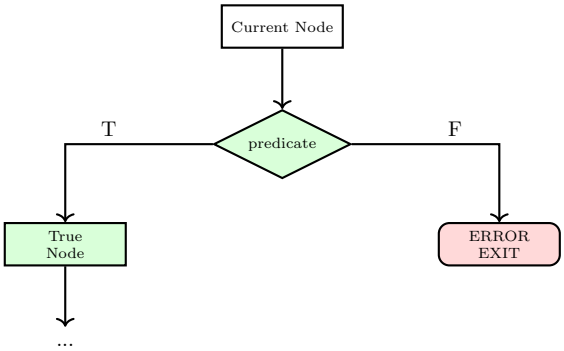


Fig. 18: Require statement insertion. The builder creates a CONDITION NODE with true edge to a node and false edge to the ERROR EXIT

C 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.

C.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.

C.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:** $\widehat{\mathsf{Enum}}(E) = \{[\ell, u] \mid 0 \leq \ell \leq u \leq |E| - 1\} \cup \{\bot, \top_E\}$

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

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, $M : \mathbb{N}_{\mathrm{fin}} \rightharpoonup \widehat{\tau}$ stores observed indices
- **Mappings:** $\widehat{\mathsf{Map}}(\kappa \Rightarrow \tau) = (\hat{d}, M)$ with default $\hat{d}$ and finite map M for observed keys

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

C.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)
  - 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.

C.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.

C.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 §??.2
- **Abstract stores:** $\hat{\sigma} \in \widehat{\Sigma} = \mathsf{Var} \rightharpoonup \widehat{\mathsf{Val}}$

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 $\widehat{\mathsf{Abort}}$