A CORRECT Evaluation Framework

The CORRECT evaluation introduces rationale-based assessment to verify whether models identify vulnerabilities for the correct reasons: The rationale refers to the model's step-by-step reasoning process and results that lead to its vulnerability detection conclusion. CORRECT evaluates this rationale against ground-truth vulnerability information (CVE descriptions, patches, and commit messages) to determine correctness: The framework operates as follows:

(1) **For vulnerable code**: The model must correctly identify the ground-truth vulnerability in its rationale. If the rationale includes the actual vulnerability cause or key elements in vulnerabilities, it is deemed correct; otherwise, the detection is considered a false negative despite potentially correct binary labeling.

(2) **For patched code**: Two evaluation modes are employed:
- *Lenient Mode*: Accepts the model's results if it either correctly identifies the code as non-vulnerable, or if it reports a vulnerability but the rationale does not reference the original (now patched) vulnerability.
- *Strict Mode*: This mode extends Lenient Mode by adding an explicit iterative check. When the model flags a patched function as vulnerable but does not reference the ground-truth vulnerability, the framework instructs the model to disregard the previously reported issues and re-evaluate whether the function still contains a vulnerability. If during this iterative process the model again reports the ground-truth vulnerability, the case is marked as a false positive, since the model fails to recognize that the vulnerability has already been fixed.

These scenarios are summarized in Table 4, which contrasts the evaluation outcomes of Lenient and Strict modes across different prediction cases. In practice, this rationale assessment is implemented by another LLM acting as a judge, which takes the model's explanation as input and decides whether it correctly reflects the ground-truth vulnerability.

Table 4. Evaluation outcomes under Lenient and Strict modes for different prediction scenarios.

Ground Truth	Prediction	Rationale	Lenient	Strict
Vulnerable	Vulnerable	Correct	TP	TP
Vulnerable	Vulnerable	Incorrect	FN	FN
Vulnerable	Non-vulnerable	–	FN	FN
Patched	Non-vulnerable	–	TN	TN
Patched	Vulnerable	References original vuln	FP	FP
Patched	Vulnerable	Other issues only	TN	Feedback→TN/FP

Choice of Evaluation Mode. While CORRECT provides both Lenient and Strict modes, we adopt Lenient Mode in our evaluation based on empirical evidence and practical considerations.

Empirical results in the CORRECT paper, including extensive statistics reported in its appendix, show that the benefits of *Strict Mode* are marginal: only 10.3% of cases are modified after the first feedback round, dropping to 3.6% in round 2, and merely 1.5% by round 4. This diminishing return suggests that the iterative refinement process yields small improvements while significantly increasing computational costs.

In addition, applying *Strict Mode* to existing methods is particularly problematic. Training-based approaches such as **ReVD**, and multi-agent frameworks such as **VulTrial**, rely on carefully designed instructions. Forcing these models to ignore previously reported vulnerabilities and repeatedly re-analyze the code disrupts their prompt structures, often leading to poor performance. Supporting such feedback loops would require retraining models from scratch with feedback-aware objectives, which introduces substantial overhead in both data construction and training optimization.

Given these issues, we adopt *Lenient Mode* for all evaluations. This ensures a fair comparison across baselines while still keeping evaluation rigor through rationale correctness assessment.

B Prompt Template.

Prompt: System Prompt for General Specifications

A structured threat modeling analysis process where security experts conduct systematic security analysis based on provided information. The expert must:

1. Understand Code Context (within <understand> tags)

Thoroughly analyze and describe the system context without revealing the vulnerability itself:

System Identification

- **What system**: Clearly identify the software system, library, or application
- **Domain/Subsystem**: Specify the particular domain or subsystem where the code operates
- **Module/Component**: Identify the specific module, component, or functional unit

Functional Analysis

- **Core functionality**: Describe what this system/module is designed to do in detail: 1. 2. 3.

2. Security Domain Classification (within <classification> tags)

Classify vulnerabilities according to 10 core security domains:

Core Security Domains:

(1) **MEM**: Memory Safety [Buffer errors, pointer issues, use-after-free, allocation problems, etc.]
(2) **STATE**: State Management [Inconsistent states, object lifecycle, concurrency issues, etc.]
(3) **INPUT**: Input Validation [Parsing logic, data validation, type checking, encoding, etc.]
(4) **LOGIC**: Program Logic [Arithmetic errors, type confusion, logical mistakes, etc.]
(5) **SEC**: Security Features [Authentication, cryptography, permissions, policy enforcement]
(6) **IO**: I/O Interaction [Filesystem operations, networking, device interaction, etc.]
(7) **CONF**: Configuration Environment [Configuration parsing, environmental variables, etc.]
(8) **TIMING**: Timing & Concurrency [Race conditions, synchronization issues, TOCTOU, etc.]
(9) **PROTOCOL**: Protocol Communication [Message parsing/formatting, session handling, etc.]
(10) **HARDWARE**: Hardware & Low-level [Low-level interfaces, architectural specifics, etc.]

3. Security Specification (within <spec> tags)

Security Specification helps understand how vulnerable code violates developer's original constraints and how patches implement fixes.

Example:

Input Information

- **Repository**: ksmbd
- **Commit Message**: ksmbd: Fix dangling pointer in krb_authenticate
- **CVE Description**: In the Linux kernel, the following vulnerability has been resolved: ksmbd: Fix dangling pointer in krb_authenticate...
- **CWE Type**: CWE-416 (Use After Free)

Code Diff:

```
-if (sess->state == SMB2_SESSION_VALID)
+if (sess->state == SMB2_SESSION_VALID) {
     ksmbd_free_user(sess->user);
+    sess->user = NULL;
+}
```

Expected Output Format:

<understand>

System Identification

- **What system**: ksmbd - in-kernel SMB server implementation for Linux
- **Domain/Subsystem**: SMB/CIFS network file sharing protocol implementation

- **Module/Component**: Kernel component receives SMB requests, uses netlink IPC...

```
</understand>
<classification>
<primary>MEM.LIFECYCLE</primary>
<tags>[STATE.CONSISTENCY, SEC.AUTHENTICATION, PROTOCOL.SMB]</tags>
<reasoning>The root cause is the failure to manage the lifecycle...</reasoning>
</classification>
<spec>HS-MEM-001: Pointer release operations require atomic cleanup with immediate nullification</spec>
```

- Reasoning: Dangling pointer vulnerability → freed but not nullified → atomic release-nullification prevents use-after-free

---

Current Analysis Target:
Repository: {repository}
Commit Message: {commit_message}
CVE Description: {cve_description}
CWE Type: {cwe_type}
Vulnerable Code:

```
{vuln}
```

Solution:

```
{fixed}
```

Please conduct analysis following the above format.

Prompt: Detailed Vulnerability Cases in General Specifications

A structured threat modeling analysis process where security experts conduct systematic security analysis based on provided information.

Analysis Framework

1. System Understanding (provided context)

```
{understand}
```

2. Security Specifications (provided rules)

```
{specification}
```

3. System-Level Threat Modeling (within `<model>` tags)

Analyze vulnerability at system design level:

- **Trust Boundaries**: Identify where system components transition between trusted/untrusted states
- **Attack Surfaces**: Focus on realistic attack vectors that led to this specific vulnerability
- **CWE Analysis**: Trace complete vulnerability chain (e.g., initial CWE-X triggers subsequent CWE-Y, where at least one matches: `{cwe_type}`)

4. Code-Level Analysis

Vulnerability Context (within `<vuln>` tags)

Provide a granular, narrative explanation of the vulnerability:

(1) **Entry Point & Preconditions**: Describe how the attack is initiated and what system state is required
(2) **Vulnerable Code Path Analysis**: Step-by-step trace of execution flow, naming key functions and variables. Pinpoint **The Flaw** and its **Consequence**
(3) **Specification Violation Mapping**: Link code path steps to specific `HS-` specifications they violate

Fix Implementation (within `<solution>` tags)

Explain how the patch enforces security specifications:

- Specific code changes and their security impact
- How fixes restore compliance with violated specifications

Example Output Format:

```
<model>
```

- **trust_boundaries**: User-Kernel boundary during SMB2 session setup; Intra-kernel function contract violation

- **attack_surfaces**: Malicious SMB2 SESSION_SETUP request; Error path exploitation
- **cwe_analysis**: Primary CWE-416 (Use After Free) enabled by state management violation

`</model>`
`<vuln>`

(1) **Entry Point**: Privileged user sends Netlink message with crafted CIPSOV4 tags
(2) **Code Path**: Loop processes tags → **The Flaw**: Off-by-one error in bounds check → **Consequence**: Stack buffer overflow
(3) **Violations**: `HS-MEM-001` (incorrect bounds check), `HS-STATE-002` (incomplete initialization)

`</vuln>`
`<solution>`

Change 1: Bounds Check Correction

```
-if (iter > CIPSO_V4_TAG_MAXCNT)
+if (iter >= CIPSO_V4_TAG_MAXCNT)
```

Compliance: Changes exclusive to inclusive comparison, preventing array overflow

Change 2: Complete Array Initialization

```
-doi_def->tags[iter] = CIPSO_V4_TAG_INVALID;
+while (iter < CIPSO_V4_TAG_MAXCNT)
+    doi_def->tags[iter++] = CIPSO_V4_TAG_INVALID;
```

Compliance: Ensures all array elements initialized to safe values

`</solution>`

Input Information:

- **CVE**: `{cve_description}`
- **CWE**: `{cwe_type}`
- **Commit**: `{commit_message}`
- **Vulnerable Code**: `{vuln}`
- **Fixed Code**: `{fixed}`
- **Code Context**: `{code_context}`

Please conduct analysis following the above framework.

Prompt: VulInstruct Knowledge Scoring Mechanism

You are a security expert. Please evaluate the relevance between the following code and VulInstruct vulnerability cases.

Target Code

`{code_snippet}`

VulInstruct Cases to Evaluate

`{chr(10).join(cases_for_evaluation)}`

Please score the relevance of each case to the target code (1-10 points):

Scoring Criteria:

- **10 points**: Highly relevant, vulnerability type, trigger conditions, and code patterns are almost identical
- **8-9 points**: Strong relevance, main vulnerability features are similar, can provide valuable reference
- **6-7 points**: Moderate relevance, some features are similar, has certain reference value
- **4-5 points**: Weak relevance, only few similarities
- **1-3 points**: Very low relevance, basically no reference value

Please strictly follow the HTML format for output:

```
<vulinstruct_evaluation>
<case_1_score>6</case_1_score>
<case_1_reasoning>Scoring reason</case_1_reasoning>
<case_2_score>8</case_2_score>
<case_2_reasoning>Scoring reason</case_2_reasoning>
...
```

```
</vulinstruct_evaluation>
```

Prompt: Domain-specific Specification Extraction

You are a security expert. Analyze these related vulnerabilities and extract reusable security specifications.
Related Historical Vulnerabilities
`{chr(10).join(nvd_descriptions)}`
Task: Extract Attack-Derived Specifications
For each vulnerability pattern you identify:

(1) **Identify the recurring attack mechanism** across these CVEs
(2) **Convert it to a positive security specification** that would prevent such attacks
(3) **Format as defensive requirements** developers must implement

Output Format:

```
<attack_specifications>
  <specification_1>
    <attack_pattern>
      Description of recurring attack mechanism
      in cve-xxx and cve-xxx in detail
    </attack_pattern>
    <defensive_spec>
      AS-DOMAIN-001: Security rule that describes
      the code behavior that prevents this attack
    </defensive_spec>
    <implementation_hint>
      Specific checks or validations needed
    </implementation_hint>
  </specification_1>
  <specification_2>...</specification_2>
</attack_specifications>
```

Prompt: Vulnerability Detection

You are a senior code security expert. Please perform systematic multi-layer security analysis on the following code.
Analysis Mode: *[Determined by knowledge relevance scoring]*

- **Autonomous Analysis**: Low relevance with knowledge base, perform independent analysis
- **Knowledge-Assisted**: High relevance knowledge filtered through LLM evaluation as reference

Input Components:

- Code Snippet: `{code_snippet}`
- Code Context: `{code_context}`
- LLM-filtered Security Knowledge: `{selected_knowledge}`

Multi-Layer Vulnerability Analysis Framework
1. Surface Symptom Analysis Identify direct suspicious operations.
2. Root Cause Investigation Trace deeper causes that give rise to the surface symptoms, focusing on data/control flow, completeness of input validation, adequacy of error handling, and potential attacker exploitation paths.
3. Architectural & Contextual Analysis Examine broader design-level factors and domain-specific assumptions in the application logic.
Comprehensive Security Assessment. Based on the above LLM-filtered Security Knowledge and three-layer analysis mode framework, please provide your professional judgment:
(i) Analysis Process: [Please describe your three-layer analysis process in detail, including discovered issues and reasoning chains]
(ii) Key Findings: [List the most important security findings]
(iii) Final Conclusion:
Please strictly follow the format below for output:
Output Format:

Table 5. Failure analysis of baseline methods. Left: iteration distribution of Vul-RAG. Right: categorized failure cases of ReVD.

Iteration Range	Ratio	Vuln.	Secure
1–3	53.1%	12.5%	87.5%
4–6	22.6%	18.0%	82.0%
7–9	6.2%	27.5%	72.5%
10 (final)	5.4%	50.0%	50.0%
10 (no decision)	12.7%	0.0%	100%

(a) Vul-RAG iteration distribution where Vuln and Secure represents the final binary prediction.

Failure Type	Ratio
Zero reasoning	4%
Wrong vulnerability type	30%
Similar type confusion	20%
Mechanism misinterpretation	14%
Over-generalization	26%
Other errors	6%

(b) ReVD failure categories.

```
<vulnerability_assessment>\\
Please strictly follow the format below for output:
  <has_vulnerability>yes/no</has_vulnerability>
  <confidence>0-1</confidence>
  <suspected_root_cause>Core findings summary</suspected_root_cause>
</vulnerability_assessment>
```

Format Description:

- `has_vulnerability`: “yes” or “no”
- `confidence`: Confidence level between 0.0 and 1.0
- If a fixing solution has been applied, you may judge “no”
- Focus on analysis quality, avoid over-sensitivity

C Failure Analysis in Vul-RAG and ReVD

We conduct a targeted analysis of the key performance factors in three representative approaches: the retrieval mechanism in Vul-RAG, the reasoning capability in ReVD, and the specifications in VulInstruct.

Failure Analysis of Vul-RAG. Vul-RAG conducts vulnerability detection in an iterative retrieval manner. Given a target code snippet, the model first retrieves the top-k most relevant vulnerability–patch cases ($k = 10$ in our experiments). For each case, it checks (i) whether the same type of vulnerability exists in the target code and (ii) whether the corresponding patch has already been applied. The outcome is a binary signal: $(1, 0)$ indicates a vulnerability without a patch (*classified as vulnerable*), $(0, 1)$ indicates a patch without vulnerability (*classified as secure*), while $(0, 0)$ or $(1, 1)$ are inconclusive and trigger the next iteration. The process continues until a conclusive decision is made; if no decisive signal is found after all iterations, the output is treated as *no decision.* As shown in Table 5, over half of the samples (53.1%) terminate within the first three iterations, while 12.7% end without any decision.

As shown in Table 5(a), over half of the samples (53.1%) terminate within three iterations, while 12.7% never reach a decision. A common failure is prematurely classifying samples as secure once a single patch match is observed, even if other vulnerabilities remain. This reliance on cross-project case matching proves unreliable: few vulnerabilities recur in exactly the same form, resulting in only 3.4% reasoning correctness under CORRECT.

Failure Analysis of ReVD. We manually analyzed 50 vulnerable cases where ReVD produced the correct binary label but failed in reasoning. Four recurring error types emerged (Table 5(b)): **(i) Zero reasoning** where no explanation was provided, **(ii) Incorrect vulnerability categorization** with either wrong CWE families or confusion between closely related types, **(iii) Incomplete mechanism understanding** where the type was identified but the related code or the exploitation results was wrong, and **(iv) Over-generalization** where only vague statements like code quality concerns replaced specific root causes. Although ReVD leverages large-scale reasoning data, its generated explanations often lack the precision and depth which highlights the key limitation of reasoning-distillation approaches: they improve surface-level consistency but do not guarantee faithful identification of vulnerability root causes.

D Manual analysis of successful cases in VulInstruct

We analyze VulInstruct's specification-guided approach using DeepSeek-R1, examining how specifications enhance vulnerability detection. The method achieves 16.7% improvement in detecting actual vulnerabilities (FN→TP) and 8.0% reduction in false positives (FP→TN). To understand these improvements, we randomly sampled 10 cases from each category and performed a manual analysis, identifying four distinct repair mechanisms through which specifications enhance detection capabilities. Table 6 shows four recurring *repair mechanisms*: **(i) Missing Security Dimension (20% of cases)** which occurs when models lack awareness of entire vulnerabilities. For example, in CVE-2021-37848, the model only checked `strncmp` for logical errors. Our specification addressed this by mandating constant-time comparisons for security-sensitive operations. **(ii) Domain-Specific Blindness (30% of cases)** occurs when models detect potential defects but underestimate their severity in specific contexts. For instance, in CVE-2022-24214, the model identified an integer overflow in DNS code but classified it as low risk, failing to recognize that DNS TXT records are attacker-controlled and high-risk vectors. Our domain specification AS-DOMAIN-1 provided this essential context, upgrading the assessment from "possible issue" to "high-severity vulnerability." **(iii) Deep Reasoning Enhancement (25% of cases)** improves models' ability to connect isolated risks into complete analysis of vulnerability mechanism. We provide a detailed case study in Figure 6. **(iv) Secure Pattern Validation (25% of cases, exclusively FP→TN)** enables models to recognize secure implementations and avoid false positives. For example, in CVE-2022-21654, the model mistakenly flagged the use of `EVP_sha256` as insecure. Our specification HS-CRYPTO-003 confirmed this as correct cryptographic practice, reinforcing that knowledge of secure patterns is as critical as vulnerability detection.

Table 6. Repair mechanisms by which specifications improve vulnerability detection

Repair Mechanism	**FN→TP**	**FP→TN**	**Total**	**Primary Knowledge Sources**
Missing Security Dimension	4	0	4 (20%)	General + Domain-specific
Domain-Specific Blindness	3	3	6 (30%)	Domain-specific + Detailed cases
Deep Reasoning Enhancement	3	2	5 (25%)	Domain-specific + Detailed cases
Secure Pattern Validation	0	5	5 (25%)	General + Detailed cases

E Using VulInstruct Finding real-world vulnerability

Case Study: From CVE-2021-32056 to a New Access Control Bypass. We provide a detailed case study to illustrate how our specification-guided auditing framework can facilitate the discovery of new real-world vulnerabilities. Starting from CVE-2021-32056 in the Cyrus IMAP Server (`cyrusimap/cyrus-imapd`), a randomly selected vulnerability case from the CORRECT dataset, we successfully identified and reported a previously unknown high-severity flaw in the same codebase. CVE-2021-32056 allowed authenticated users to bypass intended access restrictions on server annotations, leading to potential replication stalls and service disruptions. The root cause was an improperly scoped permission check in `imap/annotate.c`, where the `maywrite` check was nested inside a conditional block and skipped when the mailbox pointer was `NULL`. From this case, our framework distilled three reusable security specifications, including the specification HS-SEC-001 (annotation write operations must always enforce strict privilege-based access control).

Guided by this specification, we constructed an auditing agent workflow that simulated the workflow of a manual security audit: starting from the known flaw, generalizing its pattern, and searching the codebase for other instances where security checks might be incorrectly scoped.

Our workflow first prompted a LLM to generate candidate `git grep` commands that could reveal similar patterns elsewhere in the repository. We employed Gemini-2.5-Pro, whose relatively large context window allowed us to supply extracted code fragments without requiring a more elaborate dynamic windowing design. The generated commands reflected the model's reasoning about how the extracted specification might be violated in other parts of the codebase. In particular, the model focused on core database operations such as `store` and `delete`, which are security-critical under HS-SEC-001, and combined them with contextual information from the repository's application context (e.g., session management, mailbox operations). In doing so, the model hypothesized potential validation points where access control checks might be inconsistently applied. The next stage of our workflow was to feed the retrieved code snippets back into the model for analysis under the extracted specifications. This allowed the model to reason about whether the conditional checks in each match were relevant to security enforcement. Among four generated queries produced, one was especially effective, as shown below:

```
git grep -p --all-match \
  -e 'if (mailbox)' \
  -e 'cyrusdb_store\|cyrusdb_delete' \
  -- '*.c'
```

This query rediscovered the already patched `write_entry` function in `annotate.c`, thereby validating the search strategy, while simultaneously surfacing additional candidate sites. Furthermore, the model dismiss some false positives, such as matches in `imap/tls.c`, where the conditionals guarded resource management logic rather than access control, and therefore did not violate HS-SEC-001. At the same time, the model highlighted `imap/mboxlist.c` as a high-risk location, noting that several conditional branches in this file surrounded critical database operations such as `cyrusdb_store` and `cyrusdb_delete`. To further investigate, we selected this candidate and applied our Automatic Context Extraction Tool to retrieve the surrounding function bodies together with a depth-3 call chain, ensuring that the model could reason about how access control checks were propagated across related functions. Given this enriched context, the model conducted a more detailed analysis and identified problematic control-flow paths. Guided by this assessment, we performed a closer inspection of the extracted functions—`mboxlist_update`, `mboxlist_update_entry_full`, and `mboxlist_renamemailbox`—which ultimately led to the discovery of a severe privilege bypass. The most critical finding emerged in `mboxlist_renamemailbox`, a function with complex multi-path control flow. Here, we discovered that a `goto` statement transferred execution directly to the database update section, bypassing any privilege checks:

```
if (mbentry->mbtype & MBTYPE_INTERMEDIATE) {
    // ... destination checks ...
    goto dbupdate; // BYPASSES permission check!
}

myrights = cyrus_acl_myrights(auth_state, mbentry->acl);
if (!isadmin && !(myrights & ACL_DELETEMBOX)) {
```

```
    return IMAP_PERMISSION_DENIED;
}
```

As a consequence, an authenticated user without ACL_DELETEMBOX rights could still rename intermediate mailboxes. This represents a direct violation of HS-SEC-001: security checks were present but misplaced, enabling a privilege bypass. Conceptually, this flaw mirrors CVE-2021-32056—the same fundamental security specification was violated—but here the error manifested through control-flow misordering (`goto`) rather than conditional scoping.

The potential consequences were significant: unauthorized mailbox renaming could disrupt shared mailbox hierarchies, shift shared folders into private namespaces, and cause denial-of-service for legitimate users. We responsibly disclosed the issue to the Cyrus IMAP development team, who confirmed the vulnerability, reproduced it via integration tests, and patched it by relocating the permission check before the special-case handling logic.

F Automatic Context Extraction Tool

We implement a unified Commit URL parser that normalizes repository and patch commit metadata across heterogeneous hosts. Using a small set of hand-written matching rules, the parser recognizes both modern and legacy interfaces (e.g., GitHub, GitLab, Bitbucket, as well as cgit/gitweb deployments such as kernel.org and GNU Savannah), and deterministically maps each commit URL to a canonical triple {`repo_name`, `commit_hash`, `clone_url`}. For example, kernel.org cgit links are remapped to stable GitHub mirrors to enable uniform downstream handling.

Building on this, we resolve—when available via the host's API—the canonical parent repository for each commit, thereby obtaining the repositories for both the patched and vulnerable versions. Following CORRECT, we derive target functions from the change lines and then use `joern` and `cflow` to extract the four context types described in Section 4.1.1. For large repositories (e.g., Linux, TensorFlow), we further augment the default flow with lightweight subsystem/component identification and module-boundary detection, guided by directory structure cues together with staged filtering (includes, calls, symbol-to-file mapping) and shallow depth limits. These heuristics make CPG construction incremental and tractable, while avoiding uncontrolled expansion.

Building on this, we resolve—when available via the host's API—the canonical parent repository for each commit, thereby obtaining the repositories for both the patched and vulnerable versions. Following CORRECT, we derive target functions from the change lines and then use `joern` and `cflow` to extract the four context types described in Section 4.1.1. For large repositories (e.g., Linux, TensorFlow), we further augment the default flow with lightweight subsystem/component identification and module-boundary detection, *seeded by a small set of manually curated anchors* (Linux: `net/fs/kernel`; TensorFlow: `core/python/compiler`) and simple boundary rules (`find_module_root`, `is_module_boundary`). The analysis employs staged filtering (header-include analysis → call-graph edges → symbol-to-file mapping) and adaptive file selection (targets+module, callers/callees, include neighbors, paired `.h`/`.c`), together with conservative caps (up to 3,000 files; depth ≤ 3) and graceful timeouts/recovery. These heuristics keep CPG construction incremental and tractable while preventing uncontrolled expansion.

Our final preprocessed dataset provides substantially richer context than PrimeVul, which only supplies the full file in which a vulnerable function resides. In contrast, our dataset captures the four distinct types of context described in Section 4.1.1, thereby enabling more fine-grained program analysis and downstream tasks.