ARTIFACT EVALUATED
USENIX
AVAILABLE

Towards Generality: Task-Adaptive Binary Analysis via Semantic Retrieval and Verifiable Reasoning

Yuzhe Liu[†*], Zhijie Liu[‡*], Zhengmin Yu[†], Shu Wang[§◇], Ling Jiang[¶]
Sen Nie[¶], Shi Wu[¶], Zhanyong Tang[§◇✉], Yuan Zhang[†✉]
[†] *Fudan University, China* [‡] *ShanghaiTech University, China*
[§] *Northwest University, China* [¶] *Tencent Security Keen Lab, China*
[◇] *Xi'an Key Laboratory of Advanced Computing and Software Security, China*

Abstract

Stripped binaries dominate real-world security analysis: COTS software, firmware, and malware are released without symbols, obscuring program semantics. Recent LLM-based approaches target narrowly predefined tasks, limiting applicability in intent-driven analysis. Toward general binary analysis, agentic LLMs provide a natural direction, yet achieving generality remains challenging: query intent is hard to ground in stripped binaries, and effective tool-driven analysis is difficult.

We present BINREX, the first agentic framework for general, fully automated static binary analysis. BINREX employs a dual-encoder pipeline to enrich stripped functions with semantic context, paired with hierarchical planning to decompose intents into executable subtasks. It leverages code synthesis to translate subtasks into IDAPython scripts, executed with iterative validation to produce outputs aggregated into the final report.

We evaluate BINREX on BinREval, a benchmark of 72 tasks across seven COTS categories with machine-checkable oracles. BINREX achieves 83.3% success rate, outperforming baselines (Codex: 27.8%, Codex with IDA: 43.1%) while reducing total analysis time from 87.9 to 36.1 hours. On domain benchmarks (Juliet, NYU CTF), BINREX is competitive with task-specific methods on their respective evaluation settings. Human study and industrial deployment confirm practical value: BINREX identifies 395 unknown malware samples with 96× average efficiency gain over experts.

1 Introduction

Stripped binaries dominate real-world security analysis. Commercial off-the-shelf (COTS) software, firmware images, and malware samples are routinely distributed without symbols, removing debug information and high-level semantics [28, 34, 35, 63]. Consequently, security analysts must frequently reason about large-scale stripped binaries when performing diverse tasks such as vulnerability discovery, malware analysis, backdoor detection, and reverse engineering. These tasks vary widely in intent, granularity, and required evidence, creating a strong demand for general, fully automated binary analysis systems capable of handling diverse analyst queries without task-specific adaptation or manual intervention.

Unfortunately, existing binary analysis approaches fall short of this goal. Task-specific methods are tailored to predefined objectives: Rule-driven approaches focus on specific vulnerability patterns (e.g., Emtaint [13], Arbiter [55], SaTC [7]) or baseband analysis (e.g., BaseSpec [37]), whereas emerging LLM-based works [52] automate distinct sub-tasks such as taint analysis (e.g., LATTE [41], LARA [70]), CWE-based [14] vulnerability detection (e.g., ClearAgent [11], VulBinLLM [30]), and understanding stripped binaries (e.g., SymGen [34], DeGPT [28]). Their acceptance criteria are implicitly embedded in prompts (e.g., CWE templates) or heuristics and cannot adapt to general analysis intents. General-purpose frameworks such as Angr [58], Ghidra [1], and IDA Pro [22] provide powerful low-level primitives but still rely heavily on manual scripting and expert guidance. As a result, these approaches struggle to support general, fully automated analysis across diverse real-world binaries.

These limitations motivate the need for a new approach to achieve general, fully automated binary analysis. A natural direction is to adopt an agentic paradigm, where an LLM-driven agent plans, executes, and iteratively refines analysis steps in response to diverse user intents. Recent LLM-augmented systems (e.g., CTFAgent [32], IDA-MCP [46]) attempt to automate parts of the workflow but remain limited to narrow domains, fixed exploration strategies, or ad-hoc reasoning, and thus fail to scale to diverse real-world tasks on stripped binaries. For instance, detecting a hardcoded backdoor in a stripped `sshd` binary (detailed in §2.4) requires identifying the authentication function among thousands of unnamed `sub_XXXX` entries and verifying suspicious control-flow de-

---

[*] The authors contribute equally to this paper.
[✉] Corresponding authors.
Part of Yuzhe Liu's work was done at Northwest University and Tencent Security Keen Lab.

viations, a task that neither CWE-specific tools nor generic LLM prompting can accomplish. More fundamentally, directly applying agentic LLMs to binaries exposes two key **Challenges**:

C1: Effectively modeling general analysis intents over large stripped binaries, where the absence of symbols and high-level abstractions forces the agent to navigate thousands of low-level functions without guidance, making it difficult to identify intent-relevant code and decompose tasks in an effective and scalable manner.

C2: Producing precise and verifiable analysis results under uncertainty, as LLM reasoning and tool interactions are inherently probabilistic and error-prone, and the semantic carriers and evidence forms in binaries vary across implementations. Binary analysis, however, requires deterministic and independently checkable artifacts (e.g., control-flow slices), making it challenging to ensure correctness without explicit validation and error repair.

We formalize these challenges and detail our key insights in §2.5–2.6. Without addressing them, agentic binary analysis remains brittle and unreliable.

To this end, we present BINREX, a general framework for fully automated static analysis through a structured agentic design. At a high level, BINREX first builds a semantic map of the target binary via retrieval, then hierarchically decomposes the analyst's intent into executable subtasks, and finally fulfills each subtask through verifiable code synthesis with iterative validation.

Concretely, BINREX operates in two phases. ① *Solution to C1:* BINREX employs **Semantic Retrieval** together with a **Hierarchical Planning Strategy** to enrich stripped binaries with semantic context and model analysis intents. Instead of forcing the LLM to infer function semantics directly from low-level code, BINREX uses a dual-encoder pipeline [42, 59, 60] to retrieve semantically similar functions from a large-scale corpus of 57M open-source functions [54, 62]. These retrieval-based function hints serve as external semantic knowledge that guides hierarchical planning, enabling the agent to efficiently navigate large stripped binaries and decompose analysis queries into executable subtasks under constrained context and time budgets.

② *Solution to C2:* BINREX performs **Verifiable Reasoning** to execute and validate the derived subtasks. Rather than generating free-form scripts or textual conclusions, the agent leverages code synthesis to translate each subtask into specification-driven IDAPython scripts, which are deterministically executed to produce concrete artifacts such as addresses, call-graph slices, and data-flow facts. A dedicated validation-and-fixing loop iteratively checks intermediate results against explicit predicates and repairs failures through regeneration and re-execution. Eventually, BINREX progressively integrates planning decisions and verified execution results within an agentic loop to produce the final analysis outcome.

We evaluate BINREX on *BinREval*, a new task-oriented benchmark for agentic static binary analysis that we curate and release. *BinREval* comprises 72 tasks (binary-query pairs) spanning seven categories of real-world COTS analysis scenarios, including BYOVD vulnerability discovery, backdoor detection, ransomware analysis, CTF reverse engineering, cryptographic function localization, string obfuscation analysis, and hard-coded secret extraction. Each task pairs a stripped binary with a natural-language query and a machine-checkable evaluation oracle. On *BinREval*, BINREX achieves a task success rate of *83.3%*, outperforming strong tool-augmented baselines (Codex [49]: 27.8%, Codex with IDA [22]: 43.1%), while reducing total analysis time from 87.9 hours to 36.1 hours. We compare against end-to-end agent baselines with equivalent tool access rather than task-specific methods [11, 30, 32, 41], which target narrow objectives. On the benchmarks introduced by those task-specific methods (§5.3), BINREX remains competitive on their respective evaluation settings: on NYU CTF [53], it solves 8 of 38 Pwn challenges and 9 of 51 Reverse challenges, compared with CTFAgent's [32] 3 and 6; on the Juliet Test Suite [5], it achieves >99% accuracy on four CWE categories, comparable to LATTE [41] and VulBinLLM [30] on their reported splits (baseline numbers are cited from the respective papers, not re-run under our setup; see §5.3). Beyond automated evaluation, a human study shows accuracy gains and speedups over security experts on the evaluated malware analysis tasks. In large-scale industrial deployments, BINREX successfully identifies *395* previously unknown real-world malware samples and achieves an average efficiency improvement of *96*× over human experts (up to 169×).

Contributions. In summary, we make the following contributions:

- We propose BINREX, a general agentic framework for fully automated static binary analysis that combines: (i) *Semantic Retrieval* with *Hierarchical Planning* for efficient navigation and task decomposition in stripped binaries, and (ii) *Verifiable Reasoning* for producing deterministic, checkable analysis artifacts.
- We build and release *BinREval*, the first task-oriented benchmark for evaluating agentic static binary analysis on real-world COTS binaries, featuring diverse analysis intents and machine-checkable evaluation oracles.
- We conduct an extensive evaluation on *BinREval*, where BINREX achieves *83.3%* task success rate, outperforming all baselines while reducing analysis time from 87.9 to 36.1 hours. BINREX is also competitive with task-specific methods on their respective benchmarks (NYU CTF, Juliet Test Suite), confirming that its generality does not preclude task-specific competitiveness.
- Through a human study and large-scale industrial deployment, we demonstrate that BINREX achieves accuracy

gains and efficiency improvements relative to security experts on real-world malware analysis, identifying *395* previously unknown malware samples with an average *96*× efficiency gain over human analysts.

2 Preliminary and Related Work

2.1 Reverse Engineering and Binary Analysis

Binary reverse engineering analyzes binary code to understand functionality [1, 6, 10, 22, 32–35, 42, 50, 63, 65]. Traditional approaches transform executable binaries into more abstract representations, enabling human analysts and automated algorithms to detect security issues. Classic techniques include disassemblers and decompilers [1, 22] that lift machine code to intermediate representations, symbolic execution engines for path exploration [58], and taint-tracking frameworks for data flow analysis [8, 9]. These methods underpin modern binary analysis pipelines and remain widely used in vulnerability research, malware analysis, and software security tasks. However, these techniques often struggle to capture deep semantics or scale to complex or stripped binaries [16, 28, 34, 35, 69], motivating LLM-assisted binary analysis.

2.2 LLM-Assisted Binary Analysis

Recent work applies LLMs to binary analysis along multiple directions [52]: taint analysis (LATTE [41]), decompiler-output refinement (DeGPT [28], VulBinLLM [30]), agentic exploration over multiple representations (ClearAgent [11]), function-name recovery for stripped binaries (SYMGEN [34]), and CTF-style problem solving (CTFAgent [32]). These approaches share three *generality limitations*. **(i) Limited task coverage**: they target predefined objectives (e.g., specific CWEs) whose implicit acceptance criteria cannot translate arbitrary analyst intents into completion predicates. **(ii) Fixed workflows**: slicing granularities, prompt templates, and exploration order are hard-wired, so when task-relevant concepts manifest through different variables, functions, or control structures, these systems cannot re-plan from evidence gaps and waste effort on irrelevant code. **(iii) Insufficient semantic understanding**: without explicit decomposition of stripped binaries into named functional units, they cannot bridge arbitrary intents to concrete program behaviors. Table 1 contrasts BINREX with state-of-the-art approaches across four dimensions: task generality, adaptive exploration, verifiable evidence, and semantic navigation. Unlike prior works that trade generality for domain precision (LATTE, CTFAgent) or offer component enhancements without automation (DeGPT, SYMGEN), BINREX simultaneously addresses all dimensions. We argue that integrating *Semantic Retrieval* and *Verifiable Reasoning* is essential: retrieval addresses the Modeling Gap (localization in stripped binaries), while verification addresses the Verifiability Gap (trustworthy stopping conditions).

Table 1: Comparison of BINREX with representative LLM-assisted binary analysis approaches. ✓: Full; ○: Partial; ✗: None.

Method	Task Gen.	Adap. Expl.	Verif. Evid.	Sem. Nav.	Domain	Key Limitation
LATTE [41]	✗	○	○	✗	Taint	Fixed templates; no intent decomposition
VulBinLLM [30]	✗	✗	✗	✗	CWE	CWE classification only; no validation
ClearAgent [11]	○	○	○	✗	CWE	Framework only; no specification
CTFAgent [32]	✗	✓	○	○	CTF	Domain-specific; relies on dynamic interaction
DeGPT [28]	✗	✗	○	✗	Readability	Representation only; not task-driven
SYMGEN [34]	✗	✗	✗	✓	Symbols	Component only; no analysis
BINREX (Ours)	✓	✓	✓	✓	General	–

Beyond academic prototypes, commercial IDE assistants such as Binary Ninja Sidekick [56] provide LLM-driven hints alongside interactive reverse engineering; BINREX's focus on fully automated, oracle-graded analysis is orthogonal.

2.3 Problem Formalization

We formalize intent-driven binary analysis as iterative goal decomposition and execution. A goal is a pair

$$\mathcal{G} = \langle B, I \rangle, \tag{1}$$

where B is the target stripped binary and I is a coarse-grained natural-language intent (e.g., *locate the backdoor*); the objective is a verified result R grounded in evidence extracted from B. A planner $\mathcal{P}$ maps the intent and the current analysis context $\mathcal{C}$ (explored regions, candidate functions, accumulated evidence) into a partially-ordered subtask set

$$\mathcal{P}(B, I, \mathcal{C}) \to \mathcal{T} = \{t_1, \ldots, t_n\}, \tag{2}$$

where each t_i carries a localized objective, required artifacts (functions, basic blocks, CFG slices, cross-references), and an expected output for validation. An executor $\mathcal{E}$ runs tool-grounded procedures yielding deterministic artifacts

$$\mathcal{E}(B, t_i) \to o_i. \tag{3}$$

A satisfaction predicate

$$\Phi(\mathcal{G}, \{o_i\}) \in \{\mathit{true}, \mathit{false}\} \tag{4}$$

checks whether accumulated outputs answer the intent; if not, the system merges o_i into $\mathcal{C}$ and re-invokes $\mathcal{P}$. The global objective is to construct a dynamically growing subtask set

$$\mathcal{T}^{\star} \triangleq \{t_1, \ldots, t_m\}, \tag{5}$$

whose collective outputs establish a verified R, i.e., $\Phi(\mathcal{G}, \{o_1, \ldots, o_m\}) = \mathit{true}$, before a timeout fires. *Generality*, in this work, is the system's ability to keep $\Phi = \mathit{true}$ as the joint distribution of (B, I) varies, without task-specific adaptation.

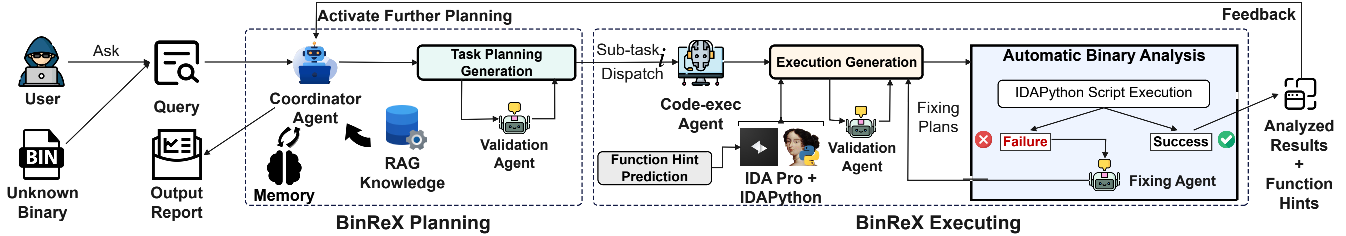

Figure 1: The workflow of BINREX.

2.4 Motivating Example

Scenario: SSHD Backdoor. Consider a security analyst tasked with detecting potential backdoors in a stripped `sshd` binary. Listing 1 shows backdoor logic in `auth_password`. The normal path (Lines 10–13) validates credentials via system calls; the backdoor (Lines 4–8) bypasses validation if the password matches `OZm7HF`, logging to a hidden file.

```
1  int auth_password(Authctxt *authctxt,
    ↪ char *password) {
2      // [BACKDOOR PATH]
3      // Hardcoded check before normal
        ↪ validation
4      if (strcmp(password, "OZm7HF") ==
        ↪ 0) {
5          // Log backdoor usage to
            ↪ hidden file
6          log_to_file("/etc/lps/lps",
            ↪ ...);
7          return 1; // Direct success
8      }
9
10     // [NORMAL PATH]
11     // Standard system authentication
12     if (sys_auth_passwd(authctxt,
        ↪ password)) {
13         return 1;
14     }
15     return 0;
16 }
```

Listing 1: Simplified pseudocode of the SSHD backdoor logic.

Why Existing Works Fail. State-of-the-art approaches struggle due to: **(i) Lack of task completion constraints**: tools like LATTE [41] and VulBinLLM [30] rely on fixed CWE-based workflows and cannot translate open-ended intents (e.g., "find backdoor") into completion predicates or verifiable reports. **(ii) Insufficient semantic understanding**: in stripped binaries with thousands of `sub_XXXX` functions, local code recovery alone cannot identify which function handles authentication versus logging, leading to aimless exploration.

2.5 Challenges

C1: Task-Adaptive Modeling Under Limited Context (Modeling Gap). Given an arbitrary intent, the agent must recover semantics for large stripped binaries and decompose into executable subtasks under constrained context and time. The key difficulty is *semantic navigation*: thousands of `sub_XXXX` placeholders provide no guidance to security-critical logic. LLMs re-deriving semantics from low-level code exhaust context budgets, causing wasted exploration or missed targets. Bridging this gap requires efficient semantic cues guiding toward task-relevant regions.

C2: Precision-Guided Execution Against Uncertainty (Verifiability Gap). Semantic carriers and evidence forms shift across binaries due to varying implementations and compilation. LLM outputs are probabilistic and contain errors; tool interactions accumulate uncertainty. Binary analysis demands deterministic artifacts (addresses, CFG slices, data-flow predicates) that are independently checkable. Therefore, an expert agent must constrain subtasks to produce verifiable evidence, validate tool outputs, and repair errors through re-generation.

2.6 Key Insights

S1: Semantic Retrieval for Efficient Navigation (Addressing C1). Rather than letting the LLM re-derive function semantics from low-level code, we provide *retrieval-based semantic hints*: a dual-encoder model [42, 59, 60] maps each stripped function into a high-dimensional semantic space, and multi-vector retrieval [19] against a 57M-function corpus from 10,000+ open-source projects [54, 62] returns high-confidence candidates as semantic cues. Working at function granularity keeps the search space tractable and aligns with how IDA and Ghidra index code; retrieval (rather than generation) gives $>10\times$ lower latency, SOTA recall, and a calibrated confidence measure. The hints form a *navigable semantic map* that lets the agent jump to security-critical regions instead of blindly exploring.

S2: Verifiable Reasoning via Constrained Code Synthesis (Addressing C2). We treat tool interaction as *constrained code synthesis*: subtasks compile into specification-driven IDAPython scripts producing deterministic artifacts (CFG paths, cross-references) as checkable evidence, enforced by a

validation-and-fixing loop. Emitting structured IR rather than raw code separates probabilistic reasoning from deterministic execution and guarantees syntactic correctness; with syntax fixed, residual failures are semantic misses that the validation loop can diagnose and repair. The workflow grounds outputs in executable proof rather than hallucinated text.

2.7 Scope and Assumptions

BINREX assumes the target binary is *unpacked* and loadable by IDA Pro; packed binaries (e.g., UPX [47]) need external unpacking, and VM-based protection needs external deobfuscation. Moderate obfuscation (inlining, control-flow flattening) is tolerated. BINREX targets *static* security-analysis intents derivable from a binary's code and data without runtime observation; intents that fundamentally require dynamic execution (race conditions, timing or cryptographic side channels, environment-dependent triggers) are out of scope and would require dynamic instrumentation or symbolic execution. We assume the LLM is *honest-but-probabilistic* (faithful to instructions but error-prone), and we sanitize decompiled output to mitigate prompt injection from malicious binaries, an emerging attack surface. Knowledge-base poisoning is out of scope: retrieved entries are name/summary hints the planner reasons over rather than code it executes, and the corpus is public upstream sources disjoint from the evaluation targets.

3 BINREX Design

3.1 Overview

BINREX is an agentic static binary analysis system built around an interactive decompiler environment [11, 32]. As illustrated in Fig. 1, it runs a two-stage loop: a *hierarchical planning* stage that decomposes a user query into subtasks (Section 3.2), and an *executing* stage that performs tool-grounded analysis backed by semantic retrieval (Section 3.3), specification-driven IDAPython synthesis (Section 3.4), and a validation–repair loop (Section 3.5). The system outputs a structured report with intermediate artifacts for reproducibility; Fig. 3 walks through a stripped SSHD backdoor end-to-end.

3.2 Hierarchical Planning

The hierarchical planner $\mathcal{P}$ (Section 2.3) translates user intent I and context $\mathcal{C}$ into a subtask set $\mathcal{T}$ whose elements each carry a localized objective, required artifacts, and a local verification predicate contributing to the global Φ.

Knowledge base. We build a knowledge base of analysis strategies, tool-invocation patterns, and security-relevant behavior signatures. The coordinator retrieves relevant strategies via RAG [21, 23] based on I and combines them with function hints to generate the initial plan.

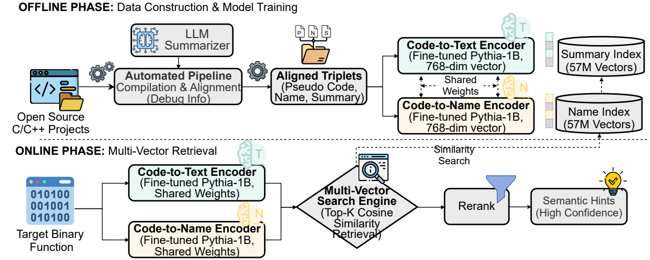

Figure 2: Function Hint Prediction pipeline. Offline: a fine-tuned Pythia-1B model embeds decompiled pseudocode into the same vector space as function names and summaries. Online: multi-vector retrieval and reranking translate an unknown stripped function into semantic hints.

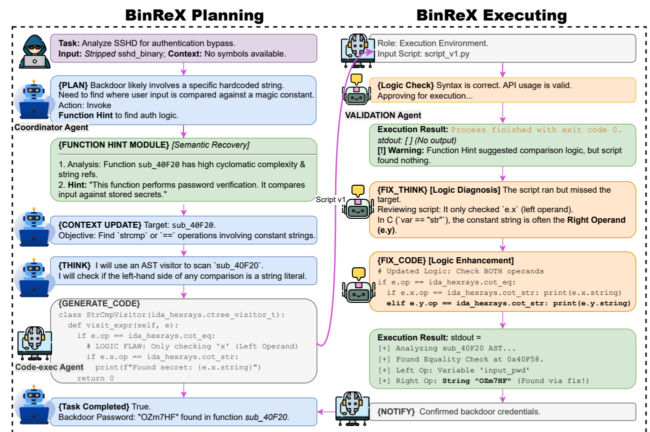

Figure 3: End-to-end workflow of BINREX on a stripped SSHD backdoor-detection task: function hints localize the authentication routine; an initial script misses the target, the fixing loop repairs and re-executes it, and the verified evidence is summarized into the final report.

Context-driven re-planning. When a subtask output o_i fails its local predicate, BINREX merges o_i into $\mathcal{C}$ and re-invokes $\mathcal{P}(B, I, \mathcal{C})$ to generate supplementary subtasks, adjusting strategy based on the observed evidence gap rather than on speculation. Our planner runs on recent general-purpose LLMs (GPT-5 [48], Gemini-3-Pro [24], DeepSeek-V3 [40]); separating planning from execution keeps high-level reasoning focused on what to analyze while delegating execution to downstream modules.

3.3 Function Hint Prediction

To enable BINREX to effectively perform binary analysis tasks, we introduce the *Function Hint Prediction* module, as shown in Fig. 2. By automatically generating high-quality hints for every function, we allow the LLM to focus on higher-level reasoning rather than re-deriving basic semantics, substantially improving analysis performance.

We train two complementary embedding models: a *code-to-text* model and a *code-to-function name* model. These two embeddings capture function semantics from distinct perspectives, natural-language summaries and concise symbolic abstractions, forming a multi-vector representation that improves retrieval coverage during hint retrieval. Both models are based on the pre-trained `Pythia-1B` [4], a widely adopted LLM, and are fine-tuned via contrastive learning.

Dataset Construction. We use an automated compilation pipeline to collect aligned function-level pairs of source and pseudo code: the Ubuntu package manager batch-acquires and compiles open-source C/C++ projects. For each project, we compile both stripped and unstripped binaries under fixed environments (e.g., `GCC v13.2, -O3, x86-64`). Debug information from unstripped binaries allows us to map decompiled pseudo code back to its corresponding source function. This process yields a large corpus of matched pairs, each consisting of a function's source code and its corresponding pseudo code representation, forming the foundation for our downstream code-to-text and code-to-function name alignment tasks.

Dual Embedding Models. We train two complementary embedding models using the same architecture and contrastive objective. The code-to-text model aligns pseudo code with natural-language summaries generated by `Qwen3-32B` [66], capturing behavioral semantics. The code-to-function-name model aligns pseudo code with function names (e.g., `VerifySignature`, `ParseHeader`), capturing symbolic intent. Both models are fine-tuned from `Pythia-1B` [4] by extracting final hidden states h as embeddings and optimizing the contrastive loss:

$$\mathcal{L} = -\log \frac{\exp(\text{sim}(h_i^c, h_i^t)/\tau)}{\sum_j \exp(\text{sim}(h_i^c, h_j^t)/\tau)}, \tag{6}$$

where positive pairs (matched code and text/name) are drawn closer while negatives are pushed apart.

Multi-Vector Search. During the hint prediction phase, every binary function is encoded by both the code-to-text and code-to-function name models. The two resulting vectors are jointly used to query two separate repositories, one built from natural-language summaries and the other from function names, through a multi-vector retrieval process. We follow prior work [54, 62] to collect 10,134 diverse open-source C/C++ projects from GitHub [25] and the GNU/Linux community [26], resulting in a repository of 57M unique functions with corresponding textual summaries and names. For each pseudo function, we retrieve the top-K matches and subject them to a cosine-similarity threshold ($\theta = 0.75$) across both spaces, returning the merged high-confidence candidates as *semantic hints* [19]. These hints are subsequently consumed by BINREX to accelerate interpretation and guide higher-level reasoning for the tasks.

Hint Integration. Function hints are attached to each pseudo function as auxiliary fields to guide navigation and candidate selection.

Design Rationale. We emphasize the key design choice in our hint prediction strategy. Instead of using a generative model to directly produce hints, we adopt a retrieval-based approach. While generative models can synthesize fluent textual summaries, their performance on out-of-distribution binary code is highly unstable and difficult to constrain [34]. Generative outputs also lack a reliable confidence estimate, making it challenging to filter or verify results.

3.4 Execution Generation

BINREX treats IDA Pro interaction as a compilation problem: each subtask is encoded as an intermediate representation (IR) that a deterministic synthesizer lowers into a logic-checked IDAPython script, which then runs to collect evidence.

Specification. We formalize the IDAPython API surface[1] into a machine-readable spec defining admissible API calls, type signatures, and well-formedness rules (required arguments, allowed value domains).

Intermediate representation. The IR is a *typed, closed-vocabulary* S-expression encoding inputs (e.g., function addresses), operations (enumerate functions, resolve cross-references, traverse call chains, extract control-flow slices), and outputs in a machine-checkable schema; every operator, argument, and output field is drawn from a fixed schema, sitting strictly above IDAPython (no Python syntax or version-specific APIs) yet strictly below natural language. Listing 2 shows a representative IR for "report whether the binary enables ASLR, DEP, and stack canaries". Each `:var` binding makes a value available to later operations via `$var` references; each `:out` field carries an `:evidence` back-pointer so the validation agent can independently recompute the conclusion from the same artifact.

```
1  (spec
2    :intent "Report ASLR / DEP / stack-canary"
3    :ops ((query_pe_header :field
       ↪ dll_characteristics :var dc)
4          (bit_test :val $dc :mask 0x0040 :var aslr)
5          (bit_test :val $dc :mask 0x0100 :var dep)
6          (enum_funcs :filter (name_contains "
             ↪ __security_check_cookie")
7                      :var ck)
8          (any :set $ck :var canary))
9    :out ((aslr   :bool :evidence $dc)
10         (dep    :bool :evidence $dc)
11         (canary :bool :evidence $ck)))
```

Listing 2: Compact IR for the ASLR / DEP / stack-canary subtask. Each op is defined in the specification; `$var` references resolve through the data-flow bindings introduced by earlier `:var` clauses.

Deterministic synthesizer. A rule-based, type-checked compiler lowers the IR into IDAPython against the specifi-

[1] https://python.docs.hex-rays.com/index.html

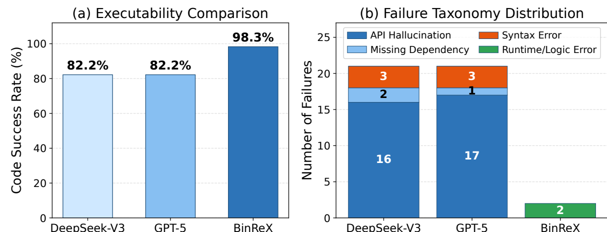

Figure 4: Code generation reliability. (a) Executability: BINREX 98.3% vs. direct LLM 82.2%. (b) Baselines fail on API hallucinations and syntax errors; IR compilation eliminates these structural faults, leaving only rare logic errors.

cation, giving *structural correctness* by construction: no missing imports, wrong argument orders, or invalid object types. Continuing the example, `query_pe_header` maps to a verified `idautils.peutils_t` access at offset `0x5E`, each `bit_test` becomes a bitwise mask, `enum_funcs` lowers to an `idautils.Functions` iteration filtered by `ida_funcs.get_func_name`, and all referenced imports are auto-injected; a pre-execution logic check then confirms specification conformance before the script runs. The public artifact (Section 7) provides the full EBNF grammar, the typed compilation rules, and a side-by-side direct-LLM hallucination on the same intent. On 118 representative subtasks, direct prompting of DeepSeek-V3/GPT-5 plateaus at 82.2% Code Success Rate while BINREX achieves 98.3% (Fig. 4); error taxonomies are in Appendix A.3/Table 8.

3.5 Validation and Fixing Loop

All executions are wrapped by an automatic validation loop. After each run, the validation agent checks whether artifacts satisfy subtask requirements and whether evidence supports the conclusion. If validation fails (e.g., missing artifacts or evidence mismatch), the fixing agent generates a repair plan [12, 43] and triggers re-execution. On success, verified artifacts and evidence are accumulated and summarized into the output report.

3.6 Implementation

We implement BINREX as a modular orchestration pipeline with explicit interfaces across planning, execution, and validation. The coordinator maintains an artifact-driven analysis state (hypotheses, explored regions, candidate functions, executed subtask traces) and updates it only from tool-generated outputs for auditability and reproducibility.

Execution roles and message flow. The executing stage adopts three roles (Fig. 1) inspired by ReAct [67]: a code-execution agent compiles subtasks into IR and invokes deterministic synthesis; a validation agent enforces pre-execution logic checks and post-execution evidence checks; a fixing agent triggers targeted repairs when evidence is missing or inconsistent.

Tool interface and sandboxing. BINREX interacts with IDA Pro [22] via IDAPython in an isolated environment. Each run loads the target binary, executes the synthesized script, and exports deterministic artifacts to a dedicated output directory.

4 Benchmark

We curate and release **BinREval**, a task-oriented benchmark [27, 64, 68] for fully automated binary analysis on real-world binaries.

4.1 Task Formulation and Categories

Each BinREval sample is a tuple: $\langle$`binary`, `query`, `task_category`, `rubric`, `ground_truth`$\rangle$. The `query` reflects analyst intents (e.g., “find the backdoor trigger”); the `rubric` specifies required fields and evidence requirements (provided in our public artifact release; see Section 7). We grade these fields against `ground_truth` and identify binaries by *SHA-256* for stability.

Task categories. BinREval covers seven categories (72 tasks in total; per-category counts in parentheses): (*i*) BYOVD vulnerability analysis (13), (*ii*) backdoor detection (13), (*iii*) ransomware analysis (8), (*iv*) CTF reverse-engineering challenges (9), (*v*) crypto function location (10), (*vi*) string obfuscation analysis (9), and (*vii*) hard-coded secret extraction (10). These categories stress complementary capabilities: precise localization (crypto/obfuscation), multi-step navigation (backdoors/ransomware), and input-trigger reasoning (BYOVD/CTF). They align with real-world analyst workflows documented in prior work on datasets [3, 15, 20, 29, 31, 36, 38, 39, 44, 45, 47, 57, 61]. Required fields per category and a per-category mapping to key analysis dimensions (multi-step navigation, trigger reasoning, evidence extraction, precise localization) are provided in the public artifact release (Section 7).

4.2 Grading Protocol and Ground Truth

For each sample, we maintain ground truth for the rubric’s required fields, including locations, triggers, and artifacts. A run succeeds only if all fields are present *and* backed by checkable evidence; the full rubric is released alongside the benchmark (Section 7). This rubric-style grading reduces ambiguity: no partial credit is given for ungrounded explanations.

Ground-truth labeling. We derive ground truth from authoritative references (e.g., CTF write-ups, advisories, threat reports) and manual reverse engineering, storing labels in a structured format aligned with the rubric. We audit labels by verifying that cited evidence resolves to claimed behavior. Multiple acceptable variants (e.g., equivalent locations) are recorded; a run is correct if it matches any variant.

4.3 Data Collection and Validity

BinREval draws samples from public sources [54, 62], including open-source software builds, CTF challenges, and malware/driver samples. For reproducibility, BINREX exports intermediate artifacts (addresses, CFG slices, script logs) for third-party audit.

Sourcing by category. We prioritize samples with *statically* verifiable behavior. CTF tasks use publicly released problems with well-defined answers; crypto/obfuscation/secret tasks use open-source builds and real-world samples from public reports; backdoor/ransomware/BYOVD tasks use binaries from threat intelligence or vulnerability reports.

Validity and leakage prevention. Public benchmarks like Juliet or aged CTF challenges risk being memorized during LLM pre-training [18, 51]. BinREval mitigates this by sourcing recent, real-world COTS binaries, firmware, and in-the-wild malware paired with custom intent-driven queries and verifiable oracles. We further use disjoint development/test splits stratified by task category, split by project (for open-source) or malware family, strip project identifiers from prompts, and keep the retrieval knowledge base disjoint from the evaluation targets.

5 Evaluation

5.1 Evaluation Tasks

We evaluate BINREX on *BinREval*, our task-oriented benchmark for fully automated binary analysis (Section 4). Each sample provides an unknown target binary (identified by SHA-256) and a natural-language query from one of seven task categories in Table 2.

Task queries. Queries mirror analyst intents by requiring verifiable evidence beyond simple classification. For instance, BYOVD and backdoor tasks demand precise locations of triggers and vulnerable paths, while crypto and obfuscation tasks require identifying algorithms and decoding schemes with concrete artifacts. These evidence requirements (e.g., function addresses, trigger conditions, data-flow traces) are enforced by a per-category rubric, applied uniformly across all task categories.

5.2 Evaluation Setup

BINREX. We use GPT-5 [48] as the base LLM (with a 400K-token context window) for planning and code generation. We set temperature to 0 to stabilize outputs and improve reproducibility, and use a fixed prompt template per task category (released with the artifact; see Section 7).

Baselines. We include two end-to-end baselines aligned with BINREX in *tool access*, *time budget*, and *stopping rule*. Unless otherwise stated, all baselines use the same base LLM (GPT-5) and temperature setting as BINREX. Each run is capped at two hours of wall-clock time (timeouts are counted as failures). All methods are instructed to produce a structured report following the same task-specific prompt template; missing required fields are treated as failures in TSR. We exclude task-specific agents (e.g., LATTE [41], ClearAgent [11], VulBinLLM [30], DeGPT [28], CTFAgent [32]) as they are designed for narrow objectives (e.g., taint analysis or CTF only) and cannot handle the diverse, general-purpose tasks in BinREval.

• *Codex (Generic Agent).* A general-purpose, ReAct-style coding agent [49] with standard terminal tools (e.g., `strings`, `objdump`, `readelf`). This represents a strong autonomous reasoning baseline without binary-specific analysis mechanisms.

• *Codex (Generic Agent) with IDA.* Extends Codex with IDA Pro [22] access via IDAPython to extract control-flow and call-graph information. The design is inspired by IDA-MCP [46], enabling access to decompiler-grounded artifacts.

Evaluation Metric. We report (i) task success rate (TSR) and (ii) time-to-completion (TTC). TSR compares reported key findings (e.g., locations, triggers, evidence) against ground truth; TTC measures wall-clock time from query-in to report-out, including tool execution and retries, capped at two hours. For diagnosis, we additionally report (iii) *field completion rate* (fraction of required fields filled) and (iv) *evidence validity rate* (fraction of filled fields whose evidence matches ground truth under the rubric), which help separate navigation misses from ungrounded claims. We categorize failures into three types: *(i) navigation failures* (insufficient field completion), *(ii) evidence failures* (filled fields with mismatched evidence), and *(iii) execution failures* (tool/script errors or timeouts). This taxonomy isolates bottlenecks in search, verification, or tool reliability.

Behavioral Trace Metrics. To characterize agent behavior beyond binary success/failure, we track four behavioral metrics per run: *(i) Execution Rounds*: count of tool or script executions, proxying exploration cost; *(ii) Script Operations (Script ops)*: sum of script generation and repair actions, reflecting verifiable evidence density; *(iii) Planning Steps*: count of planner reasoning steps; and *(iv) Actions*: aggregate count of all operations. To assess efficiency fairly, we report these metrics as *per-success costs*. Unless stated otherwise, we average costs over successful runs; for ablations we also report *failure-inclusive* per-success costs (total actions normalized by successful task count) to reflect failure-induced overhead.

Evaluation Environment. All experiments run on a Linux server with Ubuntu 20.04, AMD EPYC 7K62 48-Core Processor, 1TB RAM, and 8 Nvidia A100 GPUs.

5.3 RQ1: Performance of BINREX

Table 2 reports task success rate (TSR) across all seven task categories. BINREX achieves an overall TSR of **83.3%**, outperforming both baselines: Codex (27.8%) and Codex with

Table 2: Task success rate (TSR, %) on task categories.

Method	Task Category							Avg.
	BYOVD Vulnerability	Backdoor Detection	Ransomware Analysis	CTF Challenge	Crypto Function Location	String Obfuscation	Hard-coded Secret Extraction	
Codex	23.1	23.1	50.0	22.2	30.0	22.2	30.0	27.8
Codex with IDA	23.1	30.8	75.0	55.6	40.0	55.6	40.0	43.1
BINREX (ours)	**76.9**	**84.6**	**87.5**	**88.9**	**90.0**	**77.8**	**80.0**	**83.3**
*Pairwise comparison (*BINREX *vs. baseline; arrows show TSR change)*								
BINREX vs. Codex	↑+53.8	↑+61.5	↑+37.5	↑+66.7	↑+60.0	↑+55.6	↑+50.0	↑+55.5
BINREX vs. Codex with IDA	↑+53.8	↑+53.8	↑+12.5	↑+33.3	↑+50.0	↑+22.2	↑+40.0	↑+40.2

Deltas are absolute TSR changes (percentage points).

Table 3: Field Completion Rate (FCR) and Evidence Validity Rate (EVR) across methods (%).

Method	FCR (%)	EVR (%)
Codex	90.5	43.0
Codex with IDA	85.8	61.2
BINREX (ours)	84.5	**82.9**

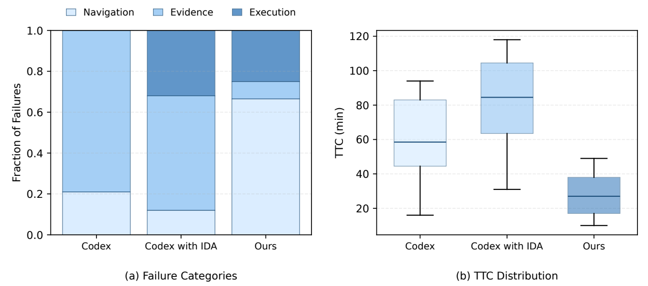

Figure 5: Diagnostic Analysis of BINREX. (a) Failure categories across methods. (b) Time-to-Completion: BINREX stays in 10–49 min while Codex with IDA spreads toward the 2-hour cap.

IDA (43.1%). The improvement is robust across all task categories, ranging from 76.9% (BYOVD) to 90.0% (crypto function location), where semantic retrieval enables rapid localization of task-relevant code regions.

To understand *why* BINREX succeeds where baselines fail, we examine FCR and EVR in Table 3. Codex achieves high FCR (it readily generates text for most fields) but exhibits low EVR, indicating that many filled fields contain hallucinated or incorrect evidence, a manifestation of the *Modeling Gap*. Codex with IDA yields lower FCR but higher EVR by leveraging IDA artifacts; however, its EVR remains limited due to the lack of a systematic verification mechanism to filter invalid evidence. In contrast, BINREX achieves the highest EVR while maintaining comparable FCR: semantic retrieval guides the agent toward task-relevant functions, and the verifiable reasoning loop ensures that extracted evidence is checked against executable scripts before being reported. This combination closes the gap between filling fields and filling fields with valid evidence.

Figure 5(a) breaks down failures into three categories: navigation, evidence, and execution. Codex failures are dominated by *evidence failures*: it often locates relevant code but cannot ground the claim with checkable artifacts. Codex with IDA shifts the failure distribution toward *execution failures*: its ad-hoc IDAPython scripts frequently crash or timeout, reflecting the fragility of unstructured code generation. BINREX's residual failures are smaller and concentrate on *navigation* for heavily obfuscated or packed binaries where semantic retrieval precision degrades, validating that the verifiable-reasoning loop suppresses evidence and execution failures by construction. Figure 6 shows that successful BINREX runs use fewer execution rounds (median ≈3) than Codex (median ≈5) and Codex with IDA (median ≈7), indicating analysis completion rather than exhaustive trial-and-error. BINREX's script-op counts are consistently non-zero on successful runs (median ≈4) while the baselines show near-zero script-related activity, confirming that successes come from running evidence-extraction scripts rather than emitting plausible-sounding prose. The combined *Actions* metric shows that BINREX invests in script operations for verification while still minimising wasteful execution rounds, consistent with semantic retrieval enabling targeted exploration.

BINREX demonstrates robust performance across different base LLMs (Section 5.7), on the NYU CTF Benchmark [53] (8 Pwn, 9 Reverse, matching specialized agents), and on the Juliet Test Suite [5] (over 99% accuracy across four major CWE categories). See Appendices A.2, A.5, A.6 for details.

Negative-case behavior on unsolvable inputs. To validate BINREX's behavior on inputs where the queried intent has no positive answer, we ran negative-case experiments on three settings: three clean SSHD binaries compiled from upstream OpenSSH (backdoor-hunt intent), ten benign drivers (BYOVD intent), and 3,680 Juliet good-case binaries across four CWE categories (Appendix A.6). BINREX returned empty

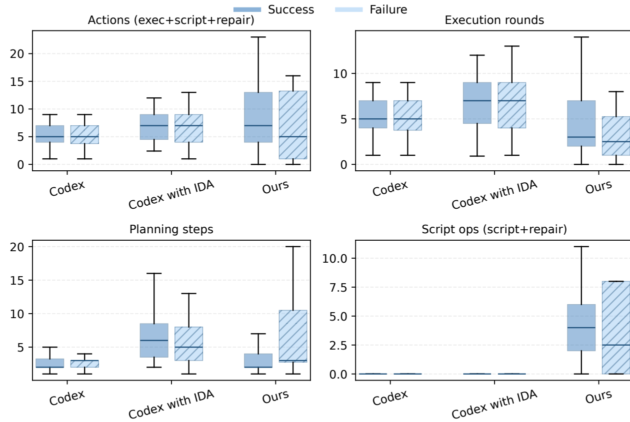

Figure 6: Behavioral trace distributions across methods and outcomes. Metrics: execution rounds (*Exec*), script ops (generation + repair), *Actions* (exec+script+repair), and *Plan* steps.

Table 4: Runtime statistics on the test set. Methods synced with the TSR table; Min capped at ≥10 min, Max capped at ≤2 hours.

Method	Min (min)	Max (min)	Total (h)
Codex	16	94	73.9
Codex with IDA	31	118	87.9
BINREX (ours)	**10**	**49**	**36.1**

conclusions in every setting, with zero false positives on the SSHD and driver sets and a >99% true-negative rate on Juliet, because the validation-and-fixing loop rejects any finding without script-extracted evidence. Complementary precision evidence on positive inputs (zero confirmed false positives across the 395 industrial findings) is reported in Section 5.6.

5.4 RQ2: Efficiency of BINREX

Motivation. A natural question is whether the higher TSR of BINREX is achieved simply by spending more time or performing more retries. RQ2 examines whether the performance gains are *efficient*, i.e., achieved with comparable or lower resource consumption.

Table 4 summarizes runtime statistics. BINREX completes runs in 10–49 minutes (36.1 hours total), reducing analysis time by 51.1% over Codex and 58.9% over Codex with IDA. This stems from semantic retrieval narrowing the search space early and verifiable reasoning avoiding ungrounded exploration. Figure 5(b) shows that BINREX's TTC concentrates in the 10–49 minute range, reflecting predictable analysis. Codex with IDA, despite equivalent tool access, spreads to the 2-hour cap, indicating frequent stalls. This confirms that tool access alone is insufficient without principled navigation and verification.

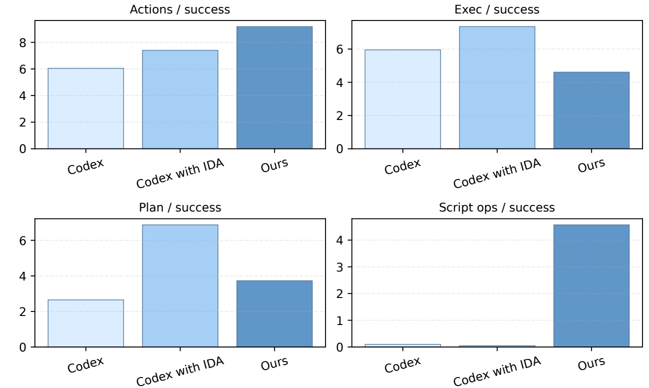

Figure 7: Per-success efficiency diagnostics. Each bar is a mean over successful runs only: *Actions* = exec+script+repair, *Exec* = execution rounds, *Plan* = planning steps, *Script ops* = script generation+repair.

Figure 7 reports per-success cost. BINREX incurs higher total actions per success (9.18) than baselines (6.05–7.40), driven primarily by *script operations* (4.57 vs. 0.05–0.10) for verifiable evidence extraction. Conversely, BINREX requires fewer planning steps (3.73) than Codex with IDA (6.87), as semantic retrieval enables more direct problem solving.

5.5 RQ3: Human-BINREX Comparison

Motivation. To quantify how BINREX compares to human analysts on realistic stripped-binary tasks, we ran a head-to-head study on two representative intents requiring deep semantic understanding: the *SSHD Backdoor* (Logic flaw) and *BYOVD Driver Analysis* (Vulnerability Discovery).

Study Setup. We recruited eight Senior Security Researchers (>3 years of reverse-engineering experience, proficient in IDA Pro). Each was asked to identify the malicious logic (backdoor trigger or vulnerable IOCTL) within a flexible timeframe; BINREX performed the same tasks in fully automated mode. We recorded wall-clock time and the breakdown of analysis phase.

Results. Figure 8 reports the wall-clock comparison: on the BYOVD driver, the expert needed about 15.5 hours (dominated by locating the dispatch logic among thousands of unnamed functions) while BINREX finished in 5.5 minutes (169×); on the SSHD backdoor, the expert needed about 5.5 hours while BINREX finished in 13.5 minutes (24×).

BYOVD driver. The expert's 15.5 hours included a dynamic-debugging setup and iterative IOCTL probing to confirm the vulnerable dispatch path. BINREX's semantic retrieval identified the IOCTL dispatch table in ∼30 seconds; the remaining time was spent verifying the exploitable parameter path through scripted control-flow extraction.

SSHD backdoor. The expert spent ∼1.5 hours just locating the authentication loop before reasoning about the hardcoded

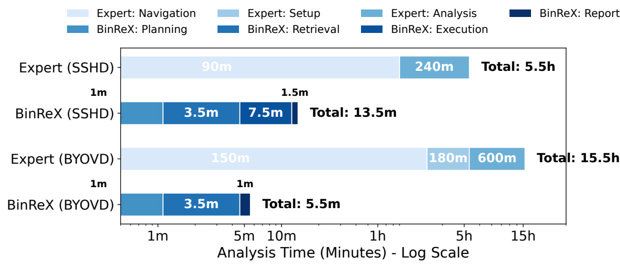

Figure 8: Time-to-Completion comparison (log scale): expert vs. BINREX on the BYOVD and SSHD-backdoor tasks.

check. BINREX's retrieval surfaced the candidate authentication function in ∼2 minutes; automated AST traversal then identified the magic-string comparison and produced the verified evidence.

Takeaway. For stripped binaries, the primary bottleneck for humans is *Navigation*: locating the relevant logic among thousands of unnamed functions. BINREX's semantic retrieval converts this open-ended search into a short verification task, bridging the gap between manual expertise and automated reasoning.

5.6 RQ4: Case Analysis

Scenario. BYOVD attacks abuse signed-but-flawed drivers for kernel privilege; distinguishing exploitable vulnerabilities from legitimate features in stripped drivers stresses BINREX's adaptive planning and evidence extraction. We ran BINREX on three stripped driver samples with the same generic query `Analyze for BYOVD vulnerabilities`; the execution adapted into three different paths based on intermediate evidence (Figure 9, Listing 3).

Case 1: Monitoring Driver. For sample `0b2ad...`, semantic retrieval revealed a lack of typical IOCTL logic but high callback registrations. BINREX switched strategy to behavioral analysis, located `ObReferenceObjectByName` and `PsSetCreateProcessNotifyRoutine` (Listing 3, top), correctly classifying it as a monitoring driver rather than a BYOVD candidate. This is the negative case the agent must recognise to avoid a false positive on a security tool that happens to look "driver-like".

Case 2: Traditional WDM Driver. For sample `00d978...`, BINREX successfully identified the `DriverEntry` and the `IRP_MJ_DEVICE_CONTROL` handler, traced dispatch function `sub_120E4`, and extracted a set of private IOCTLs. It then verified that IOCTL `0x222018` directly called `MmMapIoSpace` with user-controlled parameters and other handlers used raw `in`/`out` instructions (Listing 3, middle), establishing a high-severity BYOVD vulnerability allowing arbitrary physical memory access.

Case 3: WDF Driver and Failure Recovery. Sample

```
; Case 1 (0b2ad...): Monitoring driver (not vulnerable)
11B84:  mov  [rsp+18h], rbx        ; prologue
11BC2:  call cs:ObReferenceObjectByName

; Case 2 (00d978...): WDM physical-memory access (BYOVD)
12B3A:  in   al, dx                ; raw port input
12B3B:  mov  [ebp-1], al
12B6A:  out  dx, al                ; raw port output

; Case 3 (342c...): WDF MSR manipulation (BYOVD)
1672C:  rdmsr                      ; read MSR
1672E:  sbb  [rbp+...], edx        ; use MSR value
166D8:  wrmsr                      ; write MSR
```

Listing 3: Key evidence extracted from the three adaptive paths.

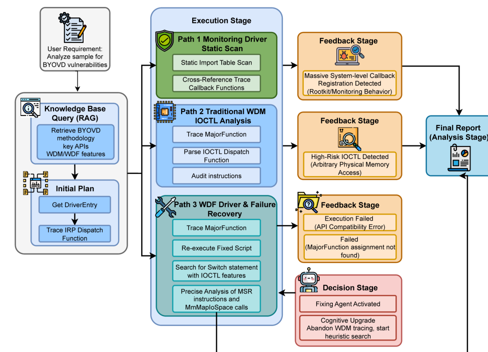

Figure 9: Adaptive analysis paths for the three drivers: (1) stop at monitoring behavior, (2) trace WDM IOCTLs, (3) recover from WDF mismatch via the fixing agent.

`342c...` initially broke: the WDM-style script the planner first emitted assumed an `IRP_MJ_DEVICE_CONTROL` entry, but the driver uses the Windows Driver Framework (WDF), which hides dispatch behind `WdfDeviceInitSetIoInCallerContextCallback`. The execution-verification module flagged the API mismatch instead of returning empty. The fixing agent regenerated the IR to walk the WDF dispatch table, resolve concrete IOCTL callbacks, and check each for sensitive instructions; the revised run reached handlers issuing `rdmsr`/`wrmsr` (Listing 3, bottom), confirming arbitrary MSR access. This case shows the planner-fixer loop salvaging a run that a single-shot agent would silently drop.

Boundary Case: a string-obfuscation failure we could not fix. Not all StringObf tasks recover. On one malware sample, retrieval pointed to the correct decoder function, but the decoding routine itself crossed several basic blocks and carried state in a per-thread buffer between them. Our `extract_string_refs` op only reads one block at a time, and none of the fixing agent's retries (a wider slice, then cross-reference lifting, then byte-window scanning) produced bytes that passed the entropy check. The task ended up flagged as an evidence failure. The fix really needs a richer IR that models decoder state across blocks; we discuss that in Section 6.

Real-world Impact. Beyond individual case studies, BINREX has been deployed in the threat analysis pipelines of partnering security vendors. In a large-scale malware analysis, BINREX successfully identified **295** real-world malicious samples exploiting BYOVD vulnerabilities and **100** variants of SSHD backdoors. All 395 findings were independently re-validated by three senior reverse engineers at the deploying vendor, who confirmed **zero false positives**. The vendors provided positive feedback on the ability of BINREX to automate evidence extraction for complex logic:

- "BINREX *significantly accelerated our response time. It autonomously pinpointed the subtle IOCTL trigger conditions in massive driver samples, helping us capture 295 BYOVD exploits that traditional scanners missed.*" — Senior Threat Researcher, Vendor A.
- "*The explainability is a game-changer. For a complex sample where two of our experts spent 16 person-hours to locate the backdoor,* BINREX *completed the analysis in just 10 minutes with verifiable evidence, which we directly used for attribution.*" — Lead Analyst, Vendor B.

5.7 RQ5: Ablation Study & Component Analysis

We evaluate four ablated variants of BINREX on the same BinREval test split (2-hour timeout, GPT-5 backbone, identical toolset): *w/o Planning* replaces the hierarchical planner with a flat ReAct loop; *w/o Semantic Retrieval* disables function-hint prediction and falls back to generic decompilation reading and string search; *w/o Evidence Extraction* emits the report directly after locating the target, skipping verification scripts; *w/o Fixing/Validation* reports the first execution result without self-correction.

Effectiveness. Removing any component degrades TSR (Figure 10a). Overall TSR drops from 83.3% to 56.9% (w/o Planning, −26.4 pp), 58.3% (w/o Retrieval, −25.0 pp), 58.3% (w/o Evidence, −25.0 pp), and 61.1% (w/o Fixing, −22.2 pp). The worst category-level hits expose where each component carries its weight: removing retrieval collapses StringObf to 44.4% (vs. 77.8%) because string-decoder localisation depends on semantic hints; removing fixing knocks CTF down to 33.3% (vs. 88.9%) because CTF tasks rely on retry-after-failure to refine probes; removing evidence extraction drops Ransomware to 50.0% (vs. 87.5%) because evidence chains discriminate ransomware behavior from generic file I/O.

Failure Mode Analysis. Figure 10c decomposes failures into navigation, evidence, and execution categories. w/o Retrieval is dominated by *navigation failures* (96.7%): without semantic hints the agent exhausts its budget searching irrelevant regions. w/o Evidence Extraction exhibits exclusively *evidence failures* (100%): the agent locates the target logic but cannot produce the verifiable artifacts the rubric requires. w/o Fixing/Validation shifts to *execution failures* (85.7%, mostly unrepaired script crashes) and *evidence failures* (7.1%). This confirms that retrieval closes the Modeling Gap and fixing closes the Verifiability Gap; they are not redundant.

Cost. Figure 10b analyzes per-success action cost. w/o Planning blows up to 11.80 executions and 9.49 script ops per success, indicating that the agent thrashes without goal decomposition. w/o Retrieval pays 9.24 in navigation cost as it manually inspects functions. w/o Fixing spends 5.32 script ops with no recovery. Full BINREX balances at 3.82 executions and converts effort into targeted verification rather than exhaustive trial-and-error.

Runtime Overhead Analysis. Figure 10d shows the TTC distribution. Full BINREX achieves the lowest median TTC (29.2 min, Max <49 min). w/o Retrieval, Planning, and Fixing all spread higher (medians 64.7, 56.9, and 93.8 min), with w/o Fixing showing the highest variance because unrepaired scripts often run to the 2-hour timeout.

Component Evaluation. Intrinsic retrieval and summarization metrics (Appendix A.1) confirm the quality of semantic retrieval: at $\theta \geq 0.75$, our retriever reaches P/R/F1 of 68.87/76.34/73.52% (vs. SymGen's 31.88/31.27/31.43 [34]), and our summaries outperform MiSum [71] on BLEU/ROUGE-L/METEOR (13.38/26.57/22.35 vs. 6.71/18.23/15.52), giving the planner more informative cues at the input boundary.

Cross-Model Evaluation. We further instantiate BINREX with Gemini-3-Pro [24], GPT-5 [48], and DeepSeek-V3 [40]; all three reach high TSR across BinREval categories (Appendix A.2), with DeepSeek-V3 competitive at lower token cost. These backends span context windows from ≈128K to ≈1M tokens, yet TSR stays stable across them, indicating that context-window size is not the bottleneck once semantic retrieval narrows the search space. The framework's effectiveness comes from the architecture, not a single proprietary model.

Robustness to Obfuscation. We rebuilt the source-available BinREval subset (Crypto and StringObf, the two categories where we have the source) with OLLVM [36], in four configurations: instruction substitution, bogus control flow, control-flow flattening, and all three together. Table 5 shows the resulting TSR. Instruction substitution and bogus control flow leave TSR unchanged on both categories: BINREX mostly keys off signals OLLVM does not touch (header bits, API names, cross-references), so this kind of syntactic mangling alone does not hurt it. Control-flow flattening is the mode that bends StringObf, dropping it from 77.8% to 66.7% as the decoder routines become harder to read in any single block; Crypto is unaffected at this stage. Stacking all three together drops Crypto from 90.0% to 80.0%, traced to one task where the AES round was inlined and got mis-tagged, while StringObf stays at 66.7%. VM-based protections are still out of scope and require an external deobfuscation step (Section 2.7).

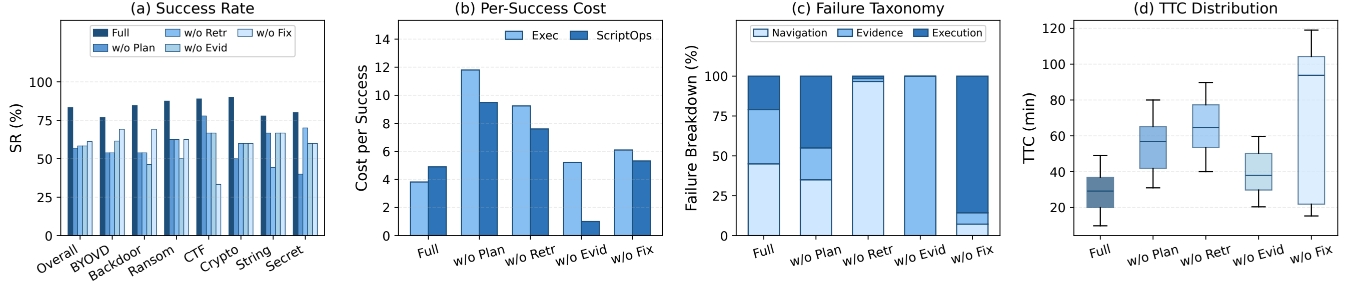

Figure 10: Ablation: (a) per-category TSR for each variant, (b) per-success action cost, (c) failure-mode breakdown, (d) TTC distribution.

Table 5: TSR under OLLVM modes on the source-available subset of BinREval (Crypto, StringObf).

OLLVM mode	Crypto TSR	StringObf TSR
Clean build (baseline)	90.0%	77.8%
Instruction substitution	90.0%	77.8%
Bogus control flow	90.0%	77.8%
Control-flow flattening	90.0%	66.7%
All three combined	80.0%	66.7%

6 Discussion

Generalizability and Cost-Efficiency. BINREX is evaluated on BinREval (Section 4), which covers seven major categories of real-world security tasks (Table 2), effectively establishing its performance and generality across diverse analysis scenarios. Regarding economic cost, while agentic workflows using LLMs incur token expenses, they offer a high return on investment compared to human labor. As demonstrated in our evaluation, BINREX reduces analysis time from days to minutes (Table 4), making it a cost-effective solution for scalable threat analysis. With the continuous decline in inference costs and stable model performance, we anticipate that such systems will become increasingly accessible for industrial deployment. Table 6 reports the per-run deployment cost across three LLM backends; pre-training the dual-encoder takes one week on 8 A100 GPUs as a one-off setup cost, and retrieval over the 57M-function corpus runs at 4.7 s per 1 000 functions using ≈27 GB of storage.

Table 6: Per-run deployment cost across LLM backends (averages per task; API pricing as of 2026-04). Pre-training and retrieval setup costs are reported in text.

LLM backend	Avg tokens	Cost / run
GPT-5	299 K	$4.50
Gemini-3-Pro	430 K	$1.30
DeepSeek-V3	902 K	$0.40

Limitations. The reasoning of BINREX relies on the fidelity of the underlying decompiler (currently IDA Pro [22]); inherent errors in decompilation [2], such as incorrect function boundaries or incomplete code recovery, can propagate to the analysis performed by the agent. While our validation-and-fixing loop (Section 3.5) mitigates this by verifying extracted evidence, the upper bound of the system is partially constrained by the quality of decompilation. For scope-level boundaries (packed and VM-obfuscated binaries, dynamic-execution intents), see Section 2.7.

Future Work. We plan to extend the *Execution Generation* module to support open-source backends such as Ghidra [1] and binary analysis frameworks such as Angr [58], broadening the accessibility of the system. To address packed malware, we intend to integrate a sandbox-based preprocessing pipeline that executes samples and dumps unpacked memory regions for static analysis. We also plan to extend the IR vocabulary to express cross-basic-block decoder state machines, addressing the evidence-failure boundary surfaced by the string-obfuscation case in Section 5.6. Finally, we will continue to expand BinREval (Section 4) to cover more diverse architectures (e.g., IoT firmware [13]) and subtler logic vulnerabilities, further advancing the boundaries of automated binary analysis.

7 Conclusion

We have presented BINREX, the first agentic framework for general, fully automated static binary analysis on stripped binaries. By integrating *Semantic Retrieval* to bridge the modeling gap and *Verifiable Reasoning* to ensure execution reliability, BINREX enables effective semantic navigation and produces deterministic, checkable evidence for arbitrary analysis intents. To standardize evaluation, we curated *BinREval*, a diverse benchmark covering seven real-world task categories with machine-checkable oracles. Our evaluation demonstrates that BINREX outperforms strong tool-augmented baselines, achieving an 83.3% success rate while reducing analysis time by over 50%. On domain benchmarks (NYU CTF Benchmark, Juliet Test Suite), BINREX is competitive with task-specific methods on their respective evaluation settings, confirming that generality does not compromise specialization. Human

studies and industrial deployments further confirm its practical value, achieving competitive efficiency relative to human experts and identifying 395 previously unknown malware samples. We hope BINREX serves as a foundation for future autonomous binary analysis, bridging the gap between high-level analyst intent and low-level binary semantics.

Acknowledgement

We thank the anonymous reviewers for the helpful comments and feedback. The work was supported in part by the National Natural Science Foundation of China (U2436207, 62372373) and the Shanghai Pilot Program for Basic Research — Fudan University 21TQ1400100 (21TQ012).

Ethical Considerations

Human Participant Study. The user study in Section 5.5 was reviewed and approved by our affiliation's Institutional Review Board (IRB); all participants gave written informed consent after a briefing on objectives, data collection, and privacy protections. Collected data (time logs, interaction traces) were stripped of any personally identifiable information and stored on encrypted systems under GDPR.

Vulnerability Disclosure and Safety. New vulnerabilities identified by BINREX (e.g., BYOVD drivers) were reported to the affected vendors with the generated evidence under standard responsible disclosure. All malware analysis ran in isolated sandboxes with no network egress. Our LLM API usage complied with provider policies: only stripped binary artifacts were transmitted, no personal data or proprietary source code.

Stakeholder Analysis. We identify five stakeholder groups: security researchers and tool builders gain a public benchmark and reproducible agent framework (but a reusable framework can be adopted by adversarial tool builders); vendor incident-response teams gain faster triage of unknown samples (with misclassification risk absent human-in-the-loop review); end users of analyzed software see only indirect benefit through faster patches and no user data is collected; LLM and model providers are unaffected (no provider data harvested, costs use public pricing); and malware authors could in principle repurpose BINREX for offensive automation, but well-resourced actors already possess equivalent in-house tooling.

Dual-Use and Mitigations. Weighing publication along the Menlo Report's beneficence axis [17]: marginal offensive uplift is low because state-actor and well-resourced criminal groups already employ in-house static/dynamic analysis tooling of comparable sophistication, while defensive uplift is concrete (expert-hours to minutes, Section 5.5; 395 previously unknown malware samples triaged, Section 5.6). Mitigations: the benchmark defines analysis tasks rather than ready-to-run exploits; artifact release follows the policy in Section 7; malware samples are distributed only by SHA-256 hash (VirusTotal/VirusShare/MalwareBazaar standard); and BYOVD vulnerabilities were reported through coordinated disclosure before submission.

Open Science

What is publicly released. *(i)* The BinREval benchmark: SHA-256 identifiers for every binary, natural-language queries, machine-checkable task-specific oracles, ground-truth labels, grading scripts, and prompt templates; the accompanying README documents step-by-step retrieval of malware samples via SHA-256 from VirusTotal, VirusShare, and MalwareBazaar (the established practice in malware research). *(ii)* The core BINREX framework code, comprising the hierarchical planner, the IR specification, the deterministic code synthesizer, and the validation-and-fixing loop. *(iii)* The dual-encoder training pipeline, including the loss function, the hyperparameters, the data construction recipe, and the training scripts. *(iv)* The evaluation harness, including metric computation and the prompt templates used in all reported experiments. All released artifacts are deposited on a public Zenodo record: https://doi.org/10.5281/zenodo.20340098.

Proprietary artifacts. Three artifacts remain proprietary under a non-disclosure agreement (NDA) with our industrial partner: the pre-trained dual-encoder weights, the 57M-function industrial training corpus, and the production orchestration layer that integrates BINREX with vendor-specific infrastructure. The corpus contains licensed proprietary code and aggregated function metadata; the weights inherit those restrictions; the orchestration layer is deployment-specific glue with no replication value to independent researchers. This split is permitted under the USENIX Call for Artifacts allowance for NDA-restricted material.

Reproducibility without the proprietary artifacts. Independent researchers can retrain equivalent dual-encoder weights on any public corpus (for example, Ubuntu upstream packages) using the released training pipeline; together with the framework code and the BinREval benchmark, this enables end-to-end reproduction of the reported methodology and comparable results.

References

[1] National Security Agency. Ghidra software reverse engineering framework, 2025. https://github.com/NationalSecurityAgency/ghidra.

[2] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An {In-Depth} analysis of disassembly on {Full-Scale} x86/x64 binaries. In *25th USENIX security symposium (USENIX security 16)*, pages 583–600, 2016.

[3] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *26th USENIX security symposium (USENIX Security 17)*, pages 1093–1110, 2017.

[4] Stella Biderman, Hailey Schoelkopf, Quentin Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, Aviya Skowron, Lintang Sutawika, and Oskar Van Der Wal. Pythia: a suite for analyzing large language models across training and scaling. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

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

[6] Jay Bosamiya, Maverick Woo, and Bryan Parno. {TRex}: Practical type reconstruction for binary code. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 6897–6915, 2025.

[7] Libo Chen, Yanhao Wang, Jiaqi Linghu, Qinsheng Hou, Quanpu Cai, Shanqing Guo, and Zhi Xue. Satc: Shared-keyword aware taint checking for detecting bugs in embedded systems. *IEEE Trans. Dependable Secur. Comput.*, 21(4):2421–2433, July 2024. doi:10.1109/TDSC.2023.3307430.

[8] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[9] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: fuzzing deeply nested branches. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 499–513, 2019.

[10] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4327–4343, 2022.

[11] Xiang Chen, Anshunkang Zhou, Chengfeng Ye, and Charles Zhang. ClearAgent: Agentic Binary Analysis for Effective Vulnerability Detection. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Language Models and Programming Languages*, pages 130–137, Singapore Singapore, October 2025. ACM. URL: https://dl.acm.org/doi/10.1145/3759425.3763397, doi:10.1145/3759425.3763397.

[12] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*, 2024. URL: https://openreview.net/forum?id=KuPixIqPiq.

[13] Kai Cheng, Yaowen Zheng, Tao Liu, Le Guan, Peng Liu, Hong Li, Hongsong Zhu, Kejiang Ye, and Limin Sun. Detecting Vulnerabilities in Linux-Based Embedded Firmware with SSE-Based On-Demand Alias Analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 360–372, Seattle WA USA, July 2023. ACM. URL: https://dl.acm.org/doi/10.1145/3597926.3598062, doi:10.1145/3597926.3598062.

[14] Steve Christey, J Kenderdine, J Mazella, and B Miles. Common weakness enumeration. *Mitre Corporation*, 2013.

[15] Savino Dambra, Yufei Han, Simone Aonzo, Platon Kotzias, Antonino Vitale, Juan Caballero, Davide Balzarotti, and Leyla Bilge. Decoding the secrets of machine learning in malware classification: A deep dive into datasets, feature extraction, and model performance. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 60–74, 2023.

[16] Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

[17] David Dittrich and Erin Kenneally. The menlo report: Ethical principles guiding information and communication technology research. *SSRN Electronic Journal*, 08 2012. doi:10.2139/ssrn.2445102.

[18] Yifan Dong, Yang Liu, Yang Ding, Yifan Zhou, Yining Wang, Xue Wang, et al. Data contamination in large language models: A survey. *arXiv preprint arXiv:2403.04811*, 2024.

[19] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. *arXiv preprint arXiv:2401.08281*, 2024.

[20] Evan Downing, Yisroel Mirsky, Kyuhong Park, and Wenke Lee. {DeepReflect}: Discovering malicious functionality through binary reconstruction. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3469–3486, 2021.

[21] Xueying Du, Geng Zheng, Kaixin Wang, Yi Zou, Yujia Wang, Wentai Deng, Jiayi Feng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag, 2025. URL: https://arxiv.org/abs/2406.11147, arXiv:2406.11147.

[22] Chris Eagle. *The IDA pro book*. no starch press, 2011.

[23] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024. URL: https://arxiv.org/abs/2312.10997, arXiv:2312.10997.

[24] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, et al. Gemini: A family of highly capable multimodal models, 2025. URL: https://arxiv.org/abs/2312.11805, arXiv:2312.11805.

[25] GitHub. Github: Let's build from here, 2024. https://github.com/.

[26] GNU. The gnu operating system and the free software movement. https://www.gnu.org/home.en.html, 2025.

[27] Haitao Hu, Peng Chen, Yanpeng Zhao, and Yuqi Chen. Agentsentinel: An end-to-end and real-time security defense framework for computer-use agents. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, CCS '25, page 3535–3549, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3719027.3765064.

[28] Peiwei Hu, Ruigang Liang, and Kai Chen. Degpt: Optimizing decompiler output with llm. In *Proceedings 2024 Network and Distributed System Security Symposium*, volume 267622140, 2024.

[29] Xin Hu, Kang G Shin, Sandeep Bhatkar, and Kent Griffin. {MutantX-S}: Scalable malware clustering based on static features. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 187–198, 2013.

[30] Nasir Hussain, Haohan Chen, Chanh Tran, Philip Huang, Zhuohao Li, Pravir Chugh, William Chen, Ashish Kundu, and Yuan Tian. VulBinLLM: LLM-powered Vulnerability Detection for Stripped Binaries, May 2025. arXiv:2505.22010 [cs]. URL: http://arxiv.org/abs/2505.22010, doi:10.48550/arXiv.2505.22010.

[31] Jiyong Jang, David Brumley, and Shobha Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 309–320, 2011.

[32] Zimo Ji, Daoyuan Wu, Wenyuan Jiang, Pingchuan Ma, Zongjie Li, and Shuai Wang. Measuring and Augmenting Large Language Models for Solving Capture-the-Flag Challenges, June 2025. arXiv:2506.17644 [cs]. URL: http://arxiv.org/abs/2506.17644, doi:10.48550/arXiv.2506.17644.

[33] Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. Binaryai: Binary software composition analysis via intelligent binary source code matching. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[34] Linxi Jiang, Xin Jin, and Zhiqiang Lin. Beyond classification: Inferring function names in stripped binaries via domain adapted llms. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*, 2025.

[35] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1631–1645, 2022.

[36] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-llvm–software protection for the masses. In *2015 ieee/acm 1st international workshop on software protection*, pages 3–9. IEEE, 2015.

[37] Eunsoo Kim, Dongkwan Kim, CheolJun Park, Insu Yun, and Yongdae Kim. BaseSpec: Comparative analysis of baseband software and cellular specifications for l3 protocols. In *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, Online, February 2021.

[38] Tencent Security Keen Lab. Binabsinspector: Vulnerability scanner for binaries, 2024. https://github.com/KeenSecurityLab/BinAbsInspector.

[39] Vector Guo Li, Matthew Dunn, Paul Pearce, Damon McCoy, Geoffrey M Voelker, and Stefan Savage. Reading the tea leaves: A comparative analysis of threat intelligence. In *28th USENIX security symposium (USENIX Security 19)*, pages 851–867, 2019.

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

[41] Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, Yuncheng Wang, Zhenyang Xu, Zhi Li, Peng Di, Yu Jiang, and Limin Sun. LLM-Powered Static

Binary Taint Analysis. *ACM Transactions on Software Engineering and Methodology*, 34(3):1–36, March 2025. URL: https://dl.acm.org/doi/10.1145/3711816, doi:10.1145/3711816.

[42] Zhijie Liu, Qiyi Tang, Sen Nie, Shi Wu, Liang Feng Zhang, and Yutian Tang. Keenhash: Hashing programs into function-aware embeddings for large-scale binary code similarity analysis. In *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2025.

[43] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: iterative refinement with self-feedback. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[44] Alessandro Mantovani, Simone Aonzo, Xabier Ugarte-Pedrero, Alessio Merlo, and Davide Balzarotti. Prevalence and impact of low-entropy packing schemes in the malware ecosystem. In *NDSS 2020, Network and Distributed System Security Symposium, 23-26 February 2020, San Diego, CA, USA*. Internet Society, 2020.

[45] Jaron Mink, Hadjer Benkraouda, Limin Yang, Arridhana Ciptadi, Ali Ahmadzadeh, Daniel Votipka, and Gang Wang. Everybody's got ml, tell me what else you have: Practitioners' perception of ml-based security tools and explanations. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2068–2085. IEEE, 2023.

[46] mrexodia. ida-pro-mcp: Ai-powered reverse engineering assistant that bridges IDA Pro with language models through MCP. https://github.com/mrexodia/ida-pro-mcp, 2025. GitHub repository, Version 1.4.0.

[47] Markus FXJ Oberhumer. Upx the ultimate packer for executables. *http://upx. sourceforge. net/*, 2004.

[48] OpenAI. Gpt-5, 2025. https://openai.com/.

[49] OpenAI. Openai codex, 2025. https://openai.com/index/introducing-codex/.

[50] Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, et al. "len or index or count, anything but v1": Predicting variable names in decompilation output with transfer learning. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4069–4087. IEEE, 2024.

[51] Michael Riddell, Ansong Ni, and Arman Cohan. Quantifying contamination in evaluating code generation capabilities of language models. *arXiv preprint arXiv:2403.04811*, 2024.

[52] Xiuwei Shang, Shaoyin Cheng, Guoqiang Chen, Yanming Zhang, Li Hu, Xiao Yu, Gangyang Li, Weiming Zhang, and Nenghai Yu. How far have we gone in binary code understanding using large language models. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12. IEEE, 2024.

[53] Minghao Shao, Sofija Jancheska, Meet Udeshi, Brendan Dolan-Gavitt, haoran xi, Kimberly Milner, Boyuan Chen, Max Yin, Siddharth Garg, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, and Muhammad Shafique. Nyu ctf bench: A scalable open-source benchmark dataset for evaluating llms in offensive security. In *Advances in Neural Information Processing Systems*, volume 37, pages 57472–57498, 2024.

[54] Wei Tang, Zhengzi Xu, Chengwei Liu, Jiahui Wu, Shouguo Yang, Yi Li, Ping Luo, and Yang Liu. Towards understanding third-party library dependency in c/c++ ecosystem. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[55] Jayakrishna Vadayath, Moritz Eckert, Kyle Zeng, Nicolaas Weideman, Gokulkrishna Praveen Menon, Yanick Fratantonio, Davide Balzarotti, Adam Doupé, Tiffany Bao, Ruoyu Wang, Christophe Hauser, and Yan Shoshitaishvili. Arbiter: Bridging the Static and Dynamic Divide in Vulnerability Discovery on Binary Programs. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022.

[56] Vector 35. Binary ninja sidekick. https://sidekick.binary.ninja/, 2025. Commercial AI assistant integrated with Binary Ninja for interactive reverse engineering.

[57] VirusTotal. Virustotal - free online virus, malware and url scanner, 2025. https://www.virustotal.com.

[58] Fish Wang and Yan Shoshitaishvili. Angr - The Next Generation of Binary Analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9, Cambridge, MA, USA, September 2017. IEEE. URL: https://ieeexplore.ieee.org/document/8077799/, doi:10.1109/SecDev.2017.14.

[59] Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xi Xiao. Clap: Learning transferable

binary code representations with natural language supervision. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 503–515, 2024.

[60] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. Jtrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–13, 2022.

[61] Chunlian Wu, Sen Chen, Jiaming Li, Renchao Chai, Lingling Fan, Xiaofei Xie, and Ruitao Feng. Beyond decision: Android malware description generation through profiling malicious behavior trajectory. *ACM Transactions on Software Engineering and Methodology*, 2025.

[62] Jiahui Wu, Zhengzi Xu, Wei Tang, Lyuye Zhang, Yueming Wu, Chengyue Liu, Kairan Sun, Lida Zhao, and Yang Liu. Ossfp: Precise and scalable c/c++ third-party library detection using fingerprinting functions. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 270–282. IEEE, 2023.

[63] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 4554–4568, 2024.

[64] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Osworld: benchmarking multimodal agents for open-ended tasks in real computer environments. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc.

[65] Xiangzhe Xu, Zhuo Zhang, Zian Su, Ziyang Huang, Shiwei Feng, Yapeng Ye, Nan Jiang, Danning Xie, Siyuan Cheng, Lin Tan, et al. Symbol preference aware generative models for recovering variable names from stripped binary. *arXiv preprint arXiv:2306.02546*, 2023.

[66] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL: https://arxiv.org/abs/2505.09388, arXiv:2505.09388.

[67] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL: https://openreview.net/forum?id=WE_vluYUL-X.

[68] Hanrong Zhang, Jingyuan Huang, Kai Mei, Yifei Yao, Zhenting Wang, Chenlu Zhan, Hongwei Wang, and Yongfeng Zhang. Agent security bench (ASB): Formalizing and benchmarking attacks and defenses in LLM-based agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL: https://openreview.net/forum?id=V4y0CpX4hK.

[69] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wenchuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 813–832. IEEE, 2021.

[70] Jiaxu Zhao, Yuekang Li, Yanyan Zou, Zhaohui Liang, Yang Xiao, Yeting Li, Bingwei Peng, Nanyu Zhong, Xinyi Wang, Wei Wang, and Wei Huo. Leveraging semantic relations in code and data to enhance taint analysis of embedded systems. In *Proceedings of the 33rd USENIX Conference on Security Symposium*, SEC '24, USA, 2024. USENIX Association.

[71] Kangchen Zhu, Zhiliang Tian, Shangwen Wang, Weiguo Chen, Zixuan Dong, Mingyue Leng, and XiaoGuang Mao. Misum: Multi-modality heterogeneous code graph learning for multi-intent binary code summarization. In *The ACM International Conference on the Foundations of Software Engineering (FSE)*, 2025.

A Appendix

A.1 Evaluation of Semantic Retrieval

To complement the end-to-end ablation study (RQ5), we report metrics for the semantic retrieval pipeline. We evaluate (i) *function hint retrieval quality* and (ii) *function summary quality* under the same retrieval threshold settings used by BINREX. Table 7 summarizes the results.

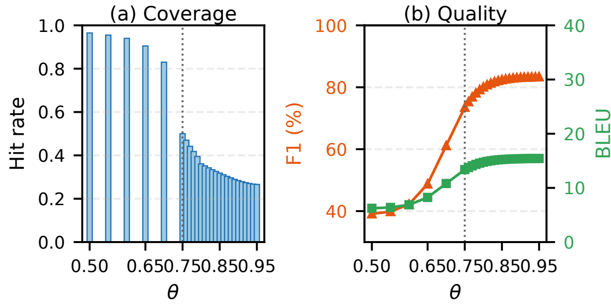

Figure 11: Coverage vs. quality as θ goes from 0.50 to 0.95. The curves bend near $\theta = 0.70$. (a) hit rate; (b) F1 (orange, left axis), BLEU (green, right axis). Dotted line: our $\theta = 0.75$.

Table 7: Semantic retrieval evaluation. Metrics at the operating threshold ($\theta \geq 0.75$) used by BINREX and at a relaxed threshold ($\theta < 0.75$) for reference; BINREX does not retrieve hints below $\theta = 0.75$.

Task	Method	Metric 1	Metric 2	Metric 3
Function hint retrieval ($\theta \geq 0.75$)	Ours	**68.87**	**76.34**	**73.52**
	SymGen	31.88	31.27	31.43
Function hint retrieval ($\theta < 0.75$)	Ours		—	
	SymGen	31.21	30.54	30.12
Function summarization ($\theta \geq 0.75$)	Ours	**13.38**	**26.57**	**22.35**
	Mi-Sum	6.71	18.23	15.52
Function summarization ($\theta < 0.75$)	Ours		—	
	Mi-Sum	5.35	17.89	15.01

Metric columns: for retrieval, (Precision, Recall, F1) in %; for summarization, (BLEU, ROUGE-L, METEOR).

Threshold Selection. We picked $\theta = 0.75$ by grid search on a held-out set. Figure 11 sweeps θ from 0.50 to 0.95 over 219,857 held-out functions. The curves bend around 0.70: below that, F1 on name retrieval and BLEU on summaries both drop fast because noisy candidates start dominating, and the hit rate (fraction of functions with at least one candidate above θ) is close to one anyway. Above 0.70, quality flattens (F1=73.52%, BLEU=13.38 at $\theta = 0.75$ per Table 7) while hit rate keeps falling. We put $\theta = 0.75$ one step into the safe side.

Task Sensitivity. Lowering θ from 0.75 to 0.65 affects categories very differently. Numbers below are TSR changes in percentage points (pp) relative to the $\theta = 0.75$ row of Table 2. Backdoor detection drops 21.0 pp and BYOVD drops 16.9 pp; both lean on the retriever's name signal to land on the right function. Crypto and String Obfuscation barely move (2.8 and 3.5 pp) because they latch onto API names and constants instead. The other three sit in between: CTF 8.9 pp, Ransomware and Secret 12.5 pp each.

Fallback Strategy. When a function falls below θ, BINREX has two backups. First, the base LLM produces a quick summary from the pseudocode. Second, we read call graphs, API usage, and cross-references to known constants (e.g., crypto magic numbers) for hints that don't depend on the retriever at all. About 15% of successful runs use at least one of these.

A.2 Performance across Base Models

We verify the robustness of BINREX by instantiating it with different base LLMs: Gemini-3-Pro, GPT-5, and DeepSeek-V3. Figure 12 shows the success rate across task categories. BINREX maintains high effectiveness across all three models, with Gemini-3-Pro and GPT-5 achieving comparable SOTA performance. DeepSeek-V3 also demonstrates strong capability, particularly in reasoning-heavy tasks, confirming that the effectiveness of our framework is not solely dependent on a specific proprietary model but generalizes across high-capability LLMs.

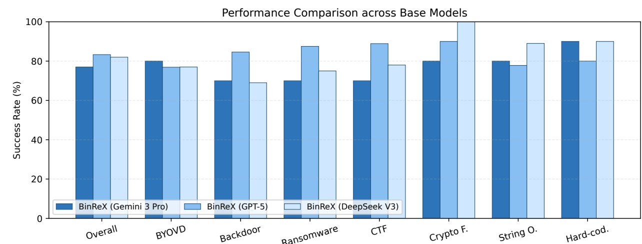

Figure 12: Success Rate comparison of BINREX instantiated with different base LLMs (Gemini-3-Pro, GPT-5, DeepSeek-V3). The framework proves effective across diverse high-capability models.

A.3 Case Study: Verifiable Code Generation

To illustrate the limitations of direct LLM generation and the efficacy of our IR compilation, Table 8 details concrete failure cases from our preliminary comparison (Section 3.4). It enumerates four representative failure modes: API hallucinations against version-incompatible IDAPython surfaces (Cases 1–2), missing imports left unresolved by the LLM (Case 3), and semantically wrong but syntactically valid code (Case 4). BINREX's deterministic synthesizer eliminates the first three classes by drawing from a verified API specification and auto-injecting required imports; the residual semantic class (Case 4) is caught downstream by the validation loop (Section 3.5).

A.4 IR Grammar and Compilation Rules

The formal IR grammar (EBNF), the typed compilation rule set mapping each operator to its verified IDAPython primitive, and the end-to-end IR-to-IDAPython walkthrough that mirrors Listing 2 (with a side-by-side direct-LLM hallucination

Table 8: Representative failure cases in direct LLM generation vs. BINREX. Direct generation suffers from API hallucinations and context errors, while BINREX produces correct, executable code.

ID	Analysis Intent	Baseline Error (LLM)	BINREX Outcome (IR Compilation)
1	Check for memory protections (ASLR/DEP/Canary)	**AttributeError**: `module 'idc' has no attribute 'INF_ASLR'` (DeepSeek-V3)	**Success**: Avoids hallucinated `INF_*` fields by emitting only version-compatible IDAPython calls and extracting evidence via supported loader/format metadata.
2	Find undetected callback functions	**AttributeError**: `module 'ida_idaapi' has no attribute 'get_inf_structure'` (GPT-5)	**Success**: Uses valid accessor functions to query binary information.
3	Detect infinite loops or deadlocks	**NameError**: `name 'idaapi' is not defined` (DeepSeek-V3)	**Success**: Automatically includes all required imports (`idaapi`, `idautils`, etc.).
4	Identify crypto functions with vulnerabilities	**Success** (GPT-5), but logic was hallucinated (flagged secure functions as vulnerable).	**Success**: Produced deterministic script that correctly listed crypto functions without false flags.

Table 9: Vulnerability detection performance on Juliet Test Suite (stripped binaries). Baseline metrics are reported in their original papers (not re-run under our filtered dataset, as official open-source implementations are not publicly available), while BINREX is evaluated on our filtered Juliet subset described in Appendix A.6.

Metric	CWE-78 (OS Cmd Injection)			CWE-134 (Format String)			CWE-190 (Int Overflow)			CWE-606 (Unchecked Loop)		
	LATTE	VulBin	**BINREX**	LATTE	VulBin	**BINREX**	LATTE	VulBin	**BINREX**	LATTE	VulBin	**BINREX**
Accuracy	96.46%	96.55%	**99.87%**	93.88%	99.74%	**99.90%**	62.1%	99.34%	**99.86%**	73.33%	74.54%	**99.48%**
Precision	100%	84.67%	**99.87%**	95.92%	98.97%	**99.90%**	52.04%	98.01%	**99.83%**	68.18%	92.78%	**99.48%**
F1 Score	96.33%	91.70%	**99.87%**	93.99%	99.48%	**99.90%**	62.05%	98.99%	**99.86%**	74.24%	85.4%	**99.48%**

VulBin = VulBinLLM. Our filtered subsets are balanced (equal bad/good); sizes: CWE-78 (1536), CWE-134 (1920), CWE-190 (3520), CWE-606 (384). LATTE and VulBinLLM numbers are cited from their papers (no released implementation) and are not directly comparable on this subset.

on the same intent) are released as part of the public artifact (Section 7); the compact summary in Section 3.4 suffices for the main results.

A.5 Evaluation on NYU CTF Benchmark

We further evaluate BINREX on the NYU CTF Benchmark [53], focusing on the *Pwn* (38 challenges) and *Reverse Engineering* (51 challenges) categories that recent specialized baselines also target. BINREX solves 8/38 Pwn and 9/51 Reverse challenges, compared with CTFAgent's [32] 3 and 6 on the full set, and ClearAgent's [11] 8 and 7 on its 19-task selected subset (8 Pwn, 11 Rev). Baseline numbers are cited verbatim from the original papers and not re-executed under our framework's constraints. The result indicates BINREX's semantic retrieval and planning framework is competitive on CTF tasks without specialized tuning or intermediate-representation lifting.

A.6 Evaluation on Juliet Test Suite

Dataset Setup. We evaluate BINREX on Juliet Test Suite v1.3 (C/C++) [5], following LATTE [41] and VulBinLLM [30] on four CWEs: *CWE-78* (OS Command Injection), *CWE-134* (Uncontrolled Format String), *CWE-190* (Integer Overflow), and *CWE-606* (Unchecked Loop Input). Test cases are GCC-compiled and stripped of symbols and debug information. We filter to external-input cases (sockets, files, env vars), yielding a balanced set of 7,360 binaries (equal bad/good).

Baselines. We compare against LATTE [41] (LLM-driven static taint analysis with program slicing) and VulBinLLM [30] (neural-decompilation-enhanced agent with memory augmentation).

Results. Table 9 presents the comparative results. BINREX reaches greater than *99%* accuracy and F1-score in every evaluated category. For instance, in CWE-190 (Integer Overflow), where static analysis often struggles with type inference, BINREX achieves *99.86%* accuracy, compared to 99.34% reported by VulBinLLM and 62.1% reported by LATTE on the same splits. For CWE-606 (Unchecked Loop Condition), BINREX reaches *99.48%* on this split, while the reported numbers for VulBinLLM and LATTE are 74.54% and 73.33%, respectively.

The gain comes from BINREX's validation loop, which plans and runs targeted IDAPython scripts to verify control- and data-flow conditions (e.g., whether a loop bound is tainted by external input), filtering out false positives that purely static or neural approaches miss.

Threats to validity. NYU CTF and Juliet likely overlap with LLM pre-training corpora [18, 51], mixing reasoning with memorization; the contamination-controlled BinREval (83.3%) addresses this.