Verifier-in-the-Loop LLM Solving of Heap Constraints for Concolic Execution

Shaoran Xia
Fudan University
Shanghai, China
xiasr24@m.fudan.edu.cn

Leyi Cheng
Soochow University
Suzhou, China
chengleyi69@gmail.com

Caihua Dong
Fudan University
Shanghai, China
25213050148@m.fudan.edu.cn

Dongdong She
Hong Kong University of Science and Technology
Hong Kong, China
dongdong@cse.ust.hk

Bo Wang
Beijing Jiaotong University
Beijing, China
wangbo_cs@bjtu.edu.cn

Xin Peng
Fudan University
Shanghai, China
pengxin@fudan.edu.cn

Zhen Dong*
Fudan University
Shanghai, China
zhendong@fudan.edu.cn

Abstract

Concolic execution systematically explores program paths by solving symbolic constraints, but in object-oriented software its practical effectiveness is often limited by *heap constraints* over objects and heap structure, such as nullness, aliasing, runtime types, and field reachability. These constraints form a systematic bottleneck in real-world software because satisfying them requires constructing a concrete heap model rather than merely computing primitive valuations, which existing SMT-based concolic engines and symbolic heap techniques do not handle effectively. We present COHEC, a hybrid concolic architecture that decomposes each target path condition into a primitive component solved by an SMT solver and a heap component handled by a *Heap Constraint Solver* (HCS). Given extracted heap constraints, HCS uses an LLM to propose candidate heap models and initialization code, while a trusted verifier executes and verifies the generated code, admitting only models whose initialization code constructs states that satisfy all constraints and keeping the LLM outside the trusted computing base. We implement COHEC on top of JDart/JPF and evaluate it on 6,750 APIs from ten Java libraries, together with a stratified sample from `commons-lang3` against the LLM-assisted baseline CONCOLLMIC. The results show that COHEC substantially improves path exploration and concrete valuation generation over native JDart, while achieving higher coverage than CONCOLLMIC at lower time and token cost. Overall, the findings indicate that treating heap constraint solving as verified heap model construction is a practical and effective way to extend concolic execution to the object-sensitive behaviors pervasive in modern software.

*Corresponding author.

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834410

CCS Concepts

• **Software and its engineering → Software verification and validation**; **Software testing and debugging**; *Automatic programming*.

Keywords

test generation, neuro-symbolic, heap constraints, large language models, SMT solving

ACM Reference Format:
Shaoran Xia, Leyi Cheng, Caihua Dong, Dongdong She, Bo Wang, Xin Peng, and Zhen Dong. 2026. Verifier-in-the-Loop LLM Solving of Heap Constraints for Concolic Execution. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3834410

1 Introduction

Background. Concolic execution has become a central technique for automated test generation, combining concrete execution with symbolic reasoning to systematically explore feasible program paths [2, 10, 20, 26, 46]. During execution, the engine accumulates path constraints and invokes an SMT solver such as Z3 to synthesize inputs that drive subsequent runs toward previously unexplored branches, enabling scalable bug finding and vulnerability discovery [8, 9, 11, 21, 42, 47, 49, 61]. This paradigm has been instantiated in a broad range of systems and extended to object-oriented settings through frameworks such as JPF, Symbolic PathFinder, JDart, and JBSE [5, 31, 38, 55]. For path conditions over primitive domains such as integers, booleans, and floating-point values, modern SMT technology is generally effective at producing satisfying assignments at a practical scale [18].

Bottleneck. The decisive bottleneck, however, lies in constraints that involve objects and heap structure. In object-oriented programs, many branch decisions depend not on numeric or boolean reasoning over primitive values but on relations among object-typed variables—whether an object reference is `null`, whether two

variables alias, whether a runtime object satisfies an `instanceof` test, whether a field dereference is realizable, or which concrete method a virtual call dispatches to. We call this class of object- and heap-sensitive conditions *heap constraints*. Unlike primitive constraints, these constraints are existential over heap structures: satisfying them requires constructing a concrete object graph that simultaneously realizes aliasing, runtime typing, field accessibility, and reachability conditions, which is beyond what SMT solving and traditional symbolic heap expansion are designed to produce directly. More fundamentally, satisfying heap constraints requires not just a symbolic valuation but a *heap model*—concrete object identities, runtime classes, relevant field bindings, and initialization actions that construct a concrete state. We refer to this outstanding challenge as the *heap constraint solving problem*.

Existing work. Existing solutions address this challenge only partially. Classical concolic engines such as JDart work well when path constraints can be discharged by mature SMT technology [18, 31], but SMT solving alone does not construct the concrete heap state needed to realize an object-sensitive path. Heap-oriented techniques—including lazy initialization, JBSE, Precise Lazy Initialization, and separation-logic-guided symbolic execution [5, 16, 25, 40]—improve the representation and expansion of symbolic object references, yet stop short of treating heap constraint solving for a target path as a dedicated heap model construction problem. As a result, mature Java symbolic execution frameworks such as JBSE and SPF often struggle on large, feature-rich real-world libraries: severe path explosion, rapidly growing heap/state spaces, and incomplete modeling of JVM or library behaviors (e.g., reflection, native methods, dynamic loading, and environment-dependent effects) frequently lead to unsupported executions, unstable exploration, or prohibitive analysis cost. These limitations ultimately lead to missed object-sensitive branches in practice.

Significance. The practical impact of unresolved heap constraints is substantial. In our manual study of 60 randomly selected APIs from `commons-lang3`, native JDart completed on 87.5% of the APIs but achieved full branch coverage on only 42.5%; for the remaining 45% that terminated without full coverage, manual inspection identified unresolved heap constraints as the dominant recurring cause of the missed branches. At benchmark scale, across 6,750 APIs from ten Java libraries, native JDart explores only 15,954 paths and produces 13,295 concrete valuations, leaving many branches guarded by heap constraints unexplored. Together, these findings show that heap constraint solving is a systematic bottleneck rather than an isolated engineering issue. If heap constraints can be made explicit to the solver and their solutions executable, there is substantial room to improve both coverage and exploration depth.

Core idea. We treat heap constraint solving as the *verified construction of heap models*, and use LLMs to propose candidate models that are structurally plausible under the program's type and code context. Rather than asking the solver to enumerate heap states symbolically, COHEC (**CO**ncolic execution with **HE**ap **C**onstraints) frames the task as candidate construction followed by trusted verification.

This formulation is a natural fit for LLMs. Satisfying an object-sensitive path typically requires constructing an object graph, choosing runtime types, and generating initialization actions that jointly realize nullness, aliasing, field access, and dynamic dispatch [22, 30, 32, 44, 56]—a construction problem whose search space grows combinatorially and is poorly matched to exhaustive symbolic or search-based enumeration. LLMs, by contrast, can leverage learned priors over code structure and type relationships to propose promising heap models directly.

Importantly, COHEC differs from prior LLM-assisted concolic systems such as CONCOLLMIC, which use LLMs to guide exploration or generate test inputs. Instead, COHEC confines the LLM to a fixed, well-scoped constraint-solving role and keeps correctness entirely on the trusted side through a fail-closed verifier. This design is embodied in the *Heap Constraint Solver* (HCS), which admits only semantically valid heap models whose initialization code constructs a valid concrete state.

Approach overview. COHEC realizes this idea through a hybrid architecture for target path re-execution. Once the engine selects a target path, it decomposes the recorded path condition into primitive constraints and heap constraints. An SMT solver solves the primitive constraints, while HCS receives the heap constraints together with the selected program and heap context and uses the LLM to propose candidate heap models and initialization code. The trusted verifier accepts only executable, semantically consistent candidates, and accepted heap models are then composed with the SMT valuation into concrete program states for re-execution.

Evaluation. We evaluate COHEC on 6,750 APIs from ten Java libraries. Compared with native JDart, COHEC increases explored paths from 15,954 to 28,757 (+80.2%) and concrete valuations from 13,295 to 23,248 (+74.9%). On a stratified sample of 30 APIs from `commons-lang3`, COHEC achieves higher line, instruction, and branch coverage than the state-of-the-art LLM-assisted concolic baseline CONCOLLMIC, while consuming 65.1% less wall-clock time and 98.8% fewer tokens. An ablation study shows that heap state constraints account for the largest share of the aggregate gain. Overall, the results demonstrate that isolating heap constraints behind a solving boundary with trusted verification is a practical and effective strategy for extending concolic execution, as demonstrated empirically on Java libraries.

Our main contributions are as follows.

- We formulate *heap constraint solving* as heap model construction for target paths, exposing it as a standalone constraint-solving problem with a well-defined solver interface.
- We propose COHEC, a verifier-in-the-loop hybrid architecture that decomposes each path condition into PC_{prim} and PC_{heap}, solves PC_{prim} with SMT, and resolves PC_{heap} through HCS under a fail-closed trust model.
- We implement the approach as an extension to JDart/JPF together with HCS, which collects heap constraint atoms, selects the relevant program context and a bounded collection of heap objects starting from variables constrained by PC_{heap}, and accepts only heap models that pass the trusted verifier.
- We evaluate the approach on 6,750 APIs from ten Java libraries and a stratified sample of 30 APIs from `commons-lang3`. COHEC explores 80.2% more paths and produces 74.9% more concrete valuations than native JDart, while achieving higher coverage with 65.1% less wall-clock time and 98.8% fewer tokens than CONCOLLMIC.

```java
// Simplified from org.jfree.chart rendering pipeline
public interface Dataset {
    int getSeriesCount();
}
public class IntervalDataset implements Dataset {
    public Dataset baseline;
    public int seriesCount;
    public int getSeriesCount() { return seriesCount; }
}
public static boolean canOverlay(Dataset ds1, Dataset ds2) {
    if (ds1 == null || ds2 == null) return false;
    if (!(ds1 instanceof IntervalDataset)) return false;
    IntervalDataset ids = (IntervalDataset) ds1;
    if (!(ids.baseline instanceof IntervalDataset))
        return false;
    if (ids.baseline == ds2) return false;
    if (ids.getSeriesCount() <= 0) return false;
    // --- target path ---
    return true;
}
```

Figure 1: Running example — A simplified JFreeChart path in which heap conditions determine feasibility before the final primitive branch condition.

2 Motivating Example

We illustrate this bottleneck with a concrete example drawn from JFreeChart. Figure 1 shows a simplified compatibility check from the dataset rendering pipeline. The method `canOverlay` determines whether one dataset can be safely overlaid on another during rendering. Although the code is small, it captures a common failure mode in object-oriented concolic execution: path feasibility is determined first by object-sensitive conditions, and only then by a primitive condition. In our formulation, this path is naturally decomposed into a pair $(PC_{\text{prim}}, PC_{\text{heap}})$, where PC_{heap} captures the heap constraints that determine executability.

To reach the target path, execution must realize a state in which `ds1` and `ds2` are non-`null`, `ds1` has runtime type `IntervalDataset`, its `baseline` field is also a non-`null` `IntervalDataset` distinct from `ds2`, and `seriesCount` is positive. The final predicate is primitive; all preceding requirements are *heap constraints* that determine whether the path is executable at all.

This asymmetry is the core difficulty. Once a concrete receiver object exists, an SMT solver can readily satisfy `getSeriesCount() > 0`. What it cannot provide directly is the heap model needed to satisfy the earlier constraints: which concrete implementation of `Dataset` should back `ds1`, whether `baseline` should point to a fresh object, and which object expressions must alias or remain distinct. In this setting, a satisfying input is therefore not merely a valuation over primitive variables, but a heap model with concrete object identities, runtime classes, relevant field bindings, and initialization actions.

Conventional concolic engines handle the primitive part of this path well, but the execution is blocked earlier. The obstacle is not the final integer branch condition; it is that the nullness, runtime type, field access, and object disequality facts governing the path are not exposed as an explicit solving problem for the target path. Without a mechanism for synthesizing a concrete heap that realizes this combination of heap constraints, the engine cannot generate a concrete input for the target path even though the remaining primitive predicate is easy.

This example motivates separating the final primitive branch condition from the heap feasibility conditions needed to realize the path. Section 3 formalizes this decomposition and the corresponding solving workflow.

3 Approach

COHEC extends concolic execution with a Heap Constraint Solver (HCS) that implements a three-stage, verifier-guided LLM workflow for heap model construction. Section 3.1 presents the overall framework, Section 3.2 details heap constraint solving, and Section 3.3 describes trusted verification and its soundness guarantees.

3.1 Overall Framework

Figure 2 presents the overall framework of COHEC. Its input is the recorded path condition PC of a selected target path, and its output is either a concrete program state for re-executing that path or a fail-closed solving status. The framework first decomposes PC into its primitive component PC_{prim} and heap component PC_{heap}. An SMT solver computes a primitive valuation for PC_{prim}, while HCS executes its three-stage, verifier-guided LLM workflow over PC_{heap} and the selected program context. Through staged generation and bounded refinement, the workflow produces a candidate model w. Only after w passes trusted verification is its realized heap state composed with the SMT valuation and returned to the concolic engine for re-execution.

This hybrid design is necessary because primitive and heap constraints require different kinds of solutions. Primitive constraints require scalar valuations, whereas heap constraints require an executable object graph that realizes runtime types, nullness, aliasing, and field relations. A uniform solving strategy would either leave the heap conditions unresolved or require the symbolic engine to explore a large space of possible heaps. COHEC instead delegates PC_{prim} and PC_{heap} to solvers suited to their respective value domains and combines their results only at the concrete state boundary.

Constraint decomposition. For a selected target path with recorded path condition PC, COHEC decomposes it as

$$PC = PC_{\text{prim}} \wedge PC_{\text{heap}},$$

where PC_{prim} consists of constraints over primitive symbolic variables, and PC_{heap} is a quantifier-free Boolean combination of heap atoms. Satisfaction of PC_{prim} is determined by the primitive valuation σ_{prim}, whereas satisfaction of PC_{heap} is determined by the heap state s. When a primitive predicate uses a value obtained through an

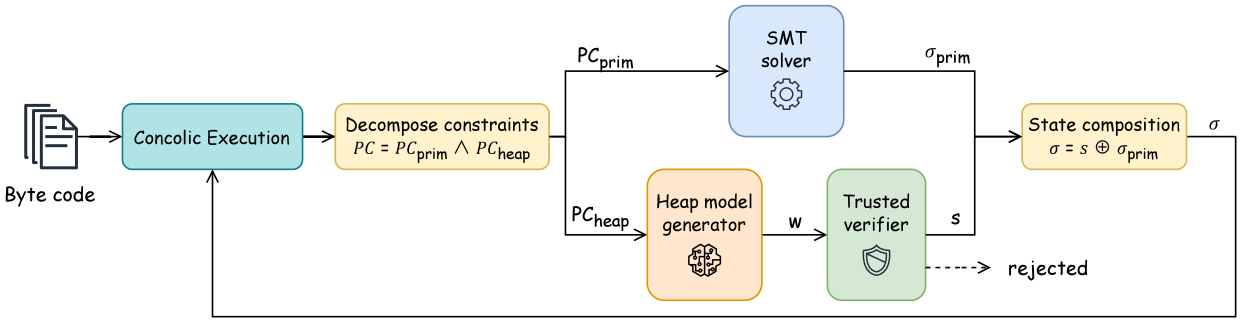

Figure 2: Overview of COHEC. The engine decomposes the recorded path condition into PC_{prim} and PC_{heap}, solves PC_{prim} with SMT, and delegates PC_{heap} to HCS with the selected context. Only heap models accepted by the trusted verifier are used to construct concrete heap states, which are composed with the SMT valuation into concrete program states for re-execution; all other outcomes remain fail-closed.

object access, the predicate belongs to PC_{prim}, while PC_{heap} records the nullness, type, and reachability conditions required to reproduce that access.

The decomposition operates on the path condition recorded by the underlying concolic executor; COHEC does not recover aliasing or other heap constraints omitted during constraint collection.

Solution composition. Solving PC_{prim} yields a primitive valuation σ_{prim}, whereas solving PC_{heap} yields a verified heap state s. These component solutions are defined as follows:

- $\sigma_{prim} : Var_{prim} \to Val_{prim}$ assigns values to the primitive symbolic variables (integers, Booleans, floating-point) introduced along that path;
- $s = (\Gamma, \rho, H)$ represents the concrete heap state. For the object variables Var_{obj} in the recorded constraints, Γ records runtime types, ρ maps variables to locations or null, and H maps allocated locations to pairs (C, F), where C is a runtime class tag and F is a finite field store. The components satisfy the following well-formedness invariant: for every $x \in Var_{obj}$, if $\rho(x) = \ell \neq \text{null}$, then $H(\ell) = (C, F)$ for some C, F and $\Gamma(x) = C$; if $\rho(x) = \text{null}$, then $\Gamma(x)$ is undefined.

The composition operator maps these component solutions to a concrete program state for re-execution:

$$\sigma = \text{Compose}(\sigma_{prim}, s) = s \oplus \sigma_{prim}.$$

During symbolic execution, the engine records the concrete slot associated with each primitive symbolic variable α. Composition writes $\sigma_{prim}(\alpha)$ to each recorded slot while preserving the object bindings, runtime types, object-valued fields, and allocated locations established by s. Consequently, if $\sigma_{prim} \models PC_{prim}$ and $s \models PC_{heap}$, then

$$\text{Compose}(\sigma_{prim}, s) \models PC_{prim} \wedge PC_{heap} = PC.$$

The running example (Figure 1) illustrates this composition concretely. HCS constructs a heap state s satisfying PC_{heap}: `ds1` and `ds2` are non-`null`, `ds1` and `ds1.baseline` have the required runtime types, and `ds1.baseline` is distinct from `ds2`. Independently, SMT solves the sole primitive condition in PC_{prim}, corresponding to `ids.getSeriesCount() > 0`, and produces a positive value for the symbolic variable associated with `seriesCount`. Composition then writes this value to the corresponding primitive slot without changing the heap structure. Consequently, the resulting concrete program state σ satisfies the original target path condition.

3.2 Heap Constraint Solving

Solving PC_{heap} requires a heap model that can be realized as an executable program state, not merely a valuation of symbolic variables. To support this construction, COHEC represents the path requirements as heap constraints over bounded object expressions and selects the relevant program context. Using these inputs, HCS executes a three-stage LLM workflow with bounded verifier-guided refinement to construct the runtime types, object graph, and initialization code forming a candidate model.

Heap-constraint representation. COHEC represents PC_{heap} as a quantifier-free Boolean formula over heap atoms. Each atom is defined over an object expression that is either a root symbolic variable or a bounded field path rooted at one, such as `ds1.baseline`. This representation captures four kinds of object facts: nullness, aliasing, subtype membership, and exact runtime type. Field access requires no separate atom because field paths are explicit object expressions, with the non-nullness of each dereferenced base recorded separately. In the running example, the null guards become negated nullness atoms, the two `instanceof` guards become subtype atoms, and `ids.baseline != ds2` becomes a disequality atom. The final `getSeriesCount() > 0` guard does not appear in PC_{heap} because it is primitive and remains in PC_{prim}. Equality guards in other paths introduce aliasing atoms, while exact-type atoms record runtime classes fixed by virtual dispatch.

The four atomic forms are:

(1) **Nullness atoms** $\text{IsNull}(p)$ and their negations encode null checks.
(2) **Aliasing atoms** $\text{Alias}(p, q)$ and $\text{Diseq}(p, q)$ encode reference equality and disequality.
(3) **Subtype atoms** $\text{InstanceOf}(p, T)$ encode subtype tests.
(4) **Exact type atoms** $\text{ExactType}(p, D)$ record the runtime class required by virtual dispatch.

Heap atom semantics. For a heap state s, $[\![p]\!]_s$ denotes the result of resolving an object expression p under ρ and H. For a root variable x, $[\![x]\!]_s = \rho(x)$; a field path is resolved by following its fields through H. We write $s \models \psi$ when a heap constraint ψ holds in s; for the basic forms, the relation is defined as follows:

$$\begin{aligned}
s \models \mathsf{IsNull}(p) &\iff [\![p]\!]_s = \mathsf{null},\\
s \models \neg\mathsf{IsNull}(p) &\iff [\![p]\!]_s \neq \mathsf{null},\\
s \models \mathsf{Alias}(p,q) &\iff [\![p]\!]_s = [\![q]\!]_s \neq \mathsf{null},\\
s \models \mathsf{Diseq}(p,q) &\iff [\![p]\!]_s \neq [\![q]\!]_s,\\
s \models \mathsf{InstanceOf}(p,T) &\iff [\![p]\!]_s = \ell \neq \mathsf{null}\\
&\qquad \wedge\ H(\ell) = (C,F)\ \wedge\ C \leq_{CT} T,\\
s \models \mathsf{ExactType}(p,D) &\iff [\![p]\!]_s = \ell \neq \mathsf{null}\\
&\qquad \wedge\ H(\ell) = (D,F).
\end{aligned}$$

The satisfaction relation for PC_{heap} follows the usual Boolean semantics for conjunction, disjunction, and negation. To define field path resolution for every heap state, resolution returns null when an intermediate dereference cannot be completed. For a target path without exceptions, PC_{heap} requires every intermediate object to be non-null, so every satisfying state resolves all observed field paths without relying on this convention.

To illustrate how this representation captures the requirements of a concrete target path, consider the running example in Figure 1. PC_{heap} requires `ds1` and `ds2` to be non-null, `ds1` and `ds1.baseline` to satisfy the two subtype tests, and `ds1.baseline` to differ from `ds2`:

$$\begin{aligned}
PC_{\text{heap}} = {} & \neg\mathsf{IsNull}(\texttt{ds1}) \wedge \neg\mathsf{IsNull}(\texttt{ds2})\\
& \wedge \mathsf{InstanceOf}(\texttt{ds1}, \texttt{IntervalDataset})\\
& \wedge \neg\mathsf{IsNull}(\texttt{ds1.baseline})\\
& \wedge \mathsf{InstanceOf}(\texttt{ds1.baseline}, \texttt{IntervalDataset})\\
& \wedge \mathsf{Diseq}(\texttt{ds1.baseline}, \texttt{ds2}).
\end{aligned}$$

Context selection. The heap atoms specify what a satisfying object graph must look like, but not how the graph can be constructed through the program's available classes and APIs. For each PC_{heap}, the engine therefore selects program and heap information relevant to its object expressions and supplies it as input context to the LLM-based construction stages in HCS.

The selected context is deliberately limited in scope. More context may expose useful construction operations, but it also increases prompt size, token cost, and irrelevant alternatives; too little context may prevent HCS from finding a realizable proposal. Context selection therefore affects practical completeness—insufficient context may lead to UNKNOWN—but never relaxes the verifier's acceptance criteria and hence does not affect soundness.

The selected context contains:

(1) the compile-time type of each variable and field path that appears in PC_{heap};
(2) the inheritance relationships and concrete classes needed to determine which runtime types are allowed;
(3) a bounded collection of heap objects selected by traversing from the variables constrained by PC_{heap}, together with their relevant object-valued field bindings;
(4) selected source code showing how objects of those classes can be created and updated, including constructors, factory and builder methods, setters, and other mutators.

The first two components identify admissible runtime classes, the third describes the path-relevant objects already available in the heap, and the fourth tells HCS how those objects can be created or updated.

History retrieval. Many target paths diverge only after executing a common prefix. Before solving a new PC_{heap}, the engine may therefore retrieve accepted solving records from earlier paths that share an execution prefix with the target path. Because these paths encountered the same object-sensitive decisions before their divergence, their accepted solutions provide plausible starting points for the new construction task.

The retrieved records summarize the earlier heap constraints and the construction decisions that produced an accepted model, including runtime types, object bindings and field links, and initialization operations. HCS uses these records only as optional examples when constructing a new candidate; it does not treat their decisions as constraints on the current path. Every candidate must still satisfy the current PC_{heap} and pass the complete trusted verification pipeline. History retrieval can therefore improve search efficiency and practical completeness, but it does not change the acceptance condition or the soundness guarantee.

Candidate model construction. Given PC_{heap}, the selected context, and any retrieved history records, HCS generates an untrusted candidate that assigns runtime types, object identities, and field links to the observed object expressions, together with initialization code intended to realize them. Producing executable code is necessary because a declaratively consistent object graph may not be realizable using the construction operations available in the target program, whereas concolic re-execution requires a concrete program state.

Definition 3.1 (Candidate Model). A *candidate model* is a triple

$$w = (\beta, \mathcal{H}_w, \mathit{code}),$$

where β maps each object expression occurring in PC_{heap} to a location or null, $\mathcal{H}_w$ is a finite partial heap model, and *code* is initialization code intended to realize $(\beta, \mathcal{H}_w)$. No consistency or executability is assumed at this stage; these properties are established by trusted verification.

HCS constructs each candidate through three dependent stages. Each stage produces an explicit intermediate artifact that fixes part of the candidate and constrains the choices available in subsequent stages:

(1) **Runtime type assignment.** For each object expression, HCS determines the admissible concrete runtime classes, subject to its declared type, subtype constraints, and exact-type constraints induced by dynamic dispatch. It then selects a consistent runtime-type assignment. This assignment restricts which existing objects may be reused and which classes may be instantiated during object graph construction. If no consistent assignment exists, the candidate is rejected before reasoning about object identities or initialization operations.
(2) **Object graph construction.** Given the runtime-type assignment, HCS constructs the bindings β and the finite partial heap

model $\mathcal{H}_w$. It decides which expressions are null, which non-null expressions refer to the same object, and whether each required object can be reused from the current heap or must be created as a fresh object. It then adds the field links needed to resolve the object expressions in PC_{heap}. The resulting graph must respect the nullness, aliasing, disequality, and type constraints in PC_{heap}, while containing only the objects and fields relevant to re-execution.

(3) **Initialization code synthesis.** Given the target object graph, HCS synthesizes an executable sequence of initialization operations intended to realize $(\beta, \mathcal{H}_w)$. The generated code may reuse existing objects, allocate fresh ones through constructors or factory methods, and establish the required field values and links through builders, setters, or other mutators. The output is concrete setup code that can be executed and subsequently checked by the trusted verifier.

These stages form a refinement pipeline: the type assignment bounds the admissible object graphs, and the selected graph defines the target that initialization code must realize. When a stage fails, HCS routes the structured failure reason to the earliest affected stage and reruns only that stage and its downstream dependents.

The synthesized operations may also establish library-specific invariants that are not represented explicitly in the partial object graph; regardless of the operations used, the resulting state must pass the same trusted verification checks.

The result remains a proposal rather than a heap solution: it is accepted only if it passes the trusted verification described next.

3.3 Trusted Verification

Because the candidate model $w = (\beta, \mathcal{H}_w, \mathit{code})$ produced by the staged LLM workflow is not assumed to be consistent or executable (Definition 3.1), it cannot be used directly for re-execution. Accepting an invalid candidate as a model of PC_{heap} would compromise the soundness of a SAT result. COHEC therefore treats w as untrusted and accepts it only after the trusted verifier confirms that w is well formed, $(\beta, \mathcal{H}_w)$ satisfies PC_{heap}, and *code* realizes $(\beta, \mathcal{H}_w)$ as an executable heap state.

These three obligations correspond to four ways in which a candidate can be invalid: it may contain malformed or out-of-scope references, define $(\beta, \mathcal{H}_w)$ that violates PC_{heap}, include initialization code that does not compile, or compile successfully but fail to realize the proposed heap model. The verifier addresses these failure modes through four ordered checks. Each check consumes the artifact admitted by the preceding checks and discharges one verification obligation:

(1) **Schema and context check.** The verifier parses w against the candidate model schema and checks that every referenced object expression, runtime type, field, constructor, and method is present and permitted by the selected context. Malformed entries, unresolved identifiers, and out-of-scope references are rejected before the verifier reasons about constraint satisfaction or code behavior.

(2) **Constraint satisfaction check.** Using β and $\mathcal{H}_w$, the verifier resolves each object expression and recovers its nullness, object identity, runtime class, and relevant field links. It then re-evaluates every atom and Boolean connective in PC_{heap} according to the semantics defined above. The candidate passes this check only if the complete formula evaluates to true.

(3) **Compilation check.** The verifier compiles *code* against the target program and its available dependencies. This check rejects invalid syntax, unresolved program elements, illegal member access, and type errors, and produces the compiled initialization code consumed by the final check.

(4) **Heap realization check.** The verifier conservatively analyzes the compiled initialization code and compares its inferred post-state with $(\beta, \mathcal{H}_w)$, projected onto the object expressions relevant to PC_{heap}. The analysis must establish the required runtime types, nullness, aliasing relations, and field links. If it finds a mismatch or cannot establish a required fact, the candidate is rejected rather than assumed valid.

The first two checks establish that the declarative candidate is well scoped and satisfies PC_{heap}; the last two establish that *code* is well typed and realizes the candidate as an executable heap state. Together, they form the trust boundary of COHEC.

When a check rejects a candidate, the verifier returns a structured diagnostic to the corresponding construction stage. Within a bounded refinement budget, that stage revises the runtime type assignment, object graph, or initialization code, and the workflow reruns any downstream stages that depend on the revised artifact. Diagnostics localize the revision, but every revised candidate must pass the complete verification pipeline before acceptance. Thus, refinement can improve the chance of finding a model without weakening the trust boundary.

Solver outcomes. HCS returns one of three statuses for PC_{heap}:

(1) **SAT.** HCS has produced a candidate model w that passes all four checks, and executing its initialization code has successfully produced a heap state s.

(2) **UNSAT.** UNSAT is returned only for contradictions established by deterministic checks over the supported heap atoms, such as incompatible exact type requirements for the same object expression. Failure to generate a candidate or pass verification does not imply UNSAT.

(3) **UNKNOWN.** The constraints may be feasible, but search or budgeted refinement did not produce an accepted candidate.

The hybrid solver reports SAT iff SMT returns SAT for PC_{prim} and HCS returns SAT for PC_{heap}. It reports UNSAT only when a trusted procedure proves either PC_{prim} or PC_{heap} infeasible. All remaining cases—including generation failure, verifier rejection, budget exhaustion, and SMT uncertainty—are reported as UNKNOWN. This fail-closed policy requires an accepted model for every SAT result.

Soundness of reported SAT results. The constraint satisfaction check establishes that the candidate interpretation satisfies PC_{heap}, while the heap realization check ensures that any heap state produced by the accepted initialization code agrees with this interpretation on every object expression used by PC_{heap}. Consequently, successful execution of an accepted candidate produces a heap state s such that $s \models PC_{\text{heap}}$.

For every reported SAT result, SMT additionally provides a valuation σ_{prim} satisfying PC_{prim}. Composition writes this valuation

into the corresponding primitive slots without changing the verified heap facts. Therefore, the resulting concrete program state $\sigma = s \oplus \sigma_{\text{prim}}$ satisfies $PC_{\text{prim}} \wedge PC_{\text{heap}} = PC$, establishing the soundness of every reported SAT result relative to the path condition recorded by the underlying concolic engine. For heap reasoning, this guarantee covers the properties explicitly represented in PC_{heap}.

4 Implementation

We implement COHEC by extending JDart/JPF with an external Heap Constraint Solver (HCS). The extension decomposes path conditions, sends PC_{prim} to the existing SMT backend, and sends PC_{heap} with its selected context to HCS. HCS returns SAT only for a verified model and otherwise fails closed. This design preserves JDart's primitive solver and path exploration while enforcing the boundary of Section 3.

Heap constraint collection. We extend JDart's instruction factory to intercept the branch-forming bytecodes that contribute heap atoms. Null checks produce IsNull atoms or their negations; reference comparisons produce Alias or Diseq atoms; `instanceof` bytecodes produce InstanceOf atoms; and receiver choices at virtual call sites produce ExactType atoms. Object-valued field accesses are represented as the bounded field paths used by PC_{heap}, while JDart continues to place predicates over primitive values in PC_{prim}. Each atom retains its originating object expression and the corresponding program type information so that later stages can resolve it against the selected context.

Before invoking HCS, the extension prunes alternatives that cannot reach the target path, deduplicates nullness atoms, and detects deterministic contradictions such as incompatible exact runtime types for one object expression. Only such contradictions can produce UNSAT without LLM inference; generation or verification failure cannot. Admissible virtual-call receivers are enumerated from the class hierarchy and cached per call site, stabilizing their mapping to exact-type atoms across refinements. For ablation, *structural heap constraints* comprise IsNull, Alias, Diseq, and InstanceOf atoms, while *dispatch-type constraints* comprise dispatch-induced ExactType atoms.

Context selection. The engine constructs the selected context from the object expressions in PC_{heap}. It first records their declared types, relevant field schemas, inheritance relationships, and admissible concrete classes. It then performs breadth-first traversal from the constrained root variables over object-valued fields. Configurable depth and object budgets limit only the collection of heap objects included in the request; a reference that crosses either limit is retained as a boundary reference but is not recursively expanded. This preserves the relationship between an observed field and the object beyond the selection boundary without claiming that the complete reachable heap has been exported.

The HCS request contains PC_{heap}, root and field bindings, selected objects, and relevant source excerpts covering constructors, factories, builders, setters, and other mutators (Section 3.2). With history retrieval enabled, it also includes accepted records from paths sharing an execution prefix. These optional examples summarize earlier constraints, runtime types, object bindings, field links, and initialization operations without constraining the current candidate.

Candidate construction and refinement. HCS realizes the three stages of Section 3.2. Runtime type assignment selects a concrete class for each non-null expression; object graph construction consumes that assignment to determine nullness, identities, reuse or allocation, and required field links; and initialization synthesis translates the graph into Java setup code using admitted operations. Each stage emits a separate intermediate artifact. A verifier diagnostic therefore revises the earliest affected artifact and its downstream dependents while retaining accepted upstream runtime-type and graph decisions. This avoids complete regeneration during bounded refinement.

The synthesizer initializes strings, lists, and maps through public API templates that preserve representation requirements overlooked by naive allocations and field assignments. A failed construction or exhausted refinement budget produces UNKNOWN.

Trusted verification and re-execution. The verifier implements the four ordered checks defined in Section 3.3. The schema and context check validates the candidate representation and rejects object expressions, types, fields, or operations not admitted by the selected context. The constraint satisfaction check resolves the proposed bindings and field links and re-evaluates the complete Boolean structure of PC_{heap}. The compilation check invokes standard `javac` against the target program and its dependencies. Finally, the heap realization check uses Soot to analyze the compiled initialization code and establish the required runtime types, nullness, aliasing relations, and field links. Any inconclusive required fact is treated as a verification failure.

Only a candidate that passes all four checks is executed to produce the concrete heap state s. The JDart/JPF extension then writes the SMT valuation σ_{prim} into the primitive slots collected for PC_{prim} while retaining the verified object bindings and heap structure, yielding $\sigma = s \oplus \sigma_{\text{prim}}$ for re-execution. Candidates rejected by any check are never installed into JDart/JPF, preserving the fail-closed behavior of the overall implementation.

5 Evaluation

5.1 Research Questions

Our evaluation is organized around four questions:

(1) **RQ1 (Exploration Effectiveness):** How effectively does COHEC explore new paths across a large-scale Java API benchmark?
(2) **RQ2 (Comparison with SOTA):** How does COHEC perform compared to the state-of-the-art LLM-assisted baseline CONCOLLMIC?
(3) **RQ3 (Trusted Verification and Recovery):** How often do candidate models fail trusted verification, and to what extent can bounded refinement recover from these failures to produce accepted candidate models?
(4) **RQ4 (Ablation):** How do different components of COHEC contribute to its overall performance?

Table 1: Subject programs and analyzed workload (APIs).

Library	Version	LOC	#APIs	Stars	Notes
commons-lang3	3.13.0	91.9K	1091	2.9k	utility methods
guava	32.1.2	3.6M	1002	51.5k	core utilities/collections
iCal4j	3.2.11	27.7K	371	827	iCal/calendar
jfreechart	1.5.4	141K	1308	1.4k	charting
jgrapht	1.3.1	127K	479	2.8k	graph algorithms
joda-time	2.12.5	88.7K	552	5k	date/time
threetenbp	1.6.8	51.4K	466	563	date/time
Time4J (base)	5.9.1	169K	703	472	date/time
SIS-Utility	1.3	43.5K	652	118	utility methods
XChart	3.8.7	25.0K	126	1.6k	charting
Total	—	—	6750	—	10 libraries

5.2 Experimental Setup

Experiment Subjects. We select 10 Java libraries used in prior studies [29], requiring each library to be both highly starred on GitHub and actively maintained. We evaluate COHEC on 6,750 APIs spanning utility functions, collections, graph algorithms, date/time processing, and charting (Table 1). Each API is wrapped as a single-entry harness and analyzed independently. All configurations start from the same initial seeds and use the same per-API setup, so differences in outcomes can be attributed to the solver configuration rather than to benchmark construction.

Metrics. For RQ1 and RQ4, we evaluate COHEC using three primary metrics: *Paths*, *Valuations*, and *Exceptions*. *Paths* counts explored executions, including both normal and exceptional runs. *Valuations* counts complete concrete inputs produced by the solver and accepted for re-execution. *Exceptions* counts executions that terminate exceptionally; it does not count unique faults. For RQ2, we additionally report the numbers of lines, instructions, and branches covered by JaCoCo, together with wall-clock time and cost, to align with prior LLM-assisted testing work. For RQ3, we record the number of candidate models that fail trusted verification, the number of failures recovered by bounded refinement, and the resulting recovery rate.

Baselines. We use native JDart [31] as the main baseline throughout the evaluation (hereinafter referred to as Base) and compare COHEC against CONCOLLMIC [32] in RQ2 as the LLM-assisted baseline. For component analysis in RQ4, we compare full COHEC with three ablations: COHEC without history retrieval, COHEC without heap state constraints, and COHEC without type constraints. For RQ2, we also report results from two earlier improved JDart variants from SV-COMP 2020 and 2021 [36, 37]. We do not treat them as direct peers because they rely on different JDart revisions and solver settings. We do not include heap-aware symbolic execution frameworks such as JBSE [5], Symbolic PathFinder [38], or Java StarFinder [39, 40] as baselines because they are *symbolic execution* frameworks, not concolic engines. Their reliance on fine-grained JVM modeling—library methods, native calls, reflection, and class loading semantics—leads to frequent unsupported bytecodes, incomplete environment models, or state space explosion on the large-scale libraries in our benchmark. In preliminary experiments, both JBSE and SPF failed to produce stable results on the majority of the 6,750 APIs, making a controlled comparison infeasible; we therefore restrict baselines to tools in the concolic execution family that share JDart's execution model.

Settings. The default proposer on the full benchmark is DeepSeek-Chat v3.2, while RQ2 uses Claude Sonnet 4.5 to align with CONCOLLMIC. We use conservative accounting: if Base times out on an API, that API is *skipped* for all configurations; if heap constraint solving fails on a path, the run falls back to the baseline result while still charging the incurred token cost. Fallback runs count as baseline outcomes, so reported gains are lower bounds. In total, only 50 of 6,750 APIs are skipped due to Base timeout.

API sampling. RQ2 compares COHEC with CONCOLLMIC on a stratified sample of 30 APIs from `commons-lang3`. Sampling is necessary because CONCOLLMIC requires whole-library LLM instrumentation before per-API execution; we attempted three libraries (total cost $194.34), but instrumentation completed only for `commons-lang3`. APIs are stratified by structural, input, and dependency complexity with a 30%/40%/30% low/medium/high allocation. We retain the least favorable completed library for COHEC, making the comparison conservative.

5.3 Results

RQ1: Exploration Effectiveness. We compare COHEC with Base and two historical variants (SV-COMP 2020 and 2021) on the full datasets. As shown in Table 2, we report *Paths*, *Valuations*, and *Exceptions*.

Results & Analysis. (1) *Overall effectiveness.* COHEC improves over Base by 80.25% in *Paths*, 74.86% in *Valuations*, and 265.89% in *Exceptions*, and achieves higher aggregate totals than the two historical JDart variants across all three metrics. This suggests that COHEC explores more symbolic opportunities and turns a large fraction of them into concrete executions, including executions that terminate exceptionally. (2) *Path exploration.* COHEC explores more paths than Base on all 10 libraries, with especially large gains on `jfreechart`, `sis-utility`, and `threetenbp`. JDart-2020 explores more paths on `ical4j`; however, COHEC produces more *Valuations*

Table 2: RQ1: Overall exploration effectiveness of COHEC against Base and historical variants.

Metrics	Tool	Library										Total
		lang3	guava	ical4j	jfreechart	jgrapht	joda-time	sis-utility	threetenbp	time4j	xchart	
Paths	Base	7709	2787	340	1258	380	645	1266	503	991	75	15954
	JDart-2020	4716	1991	**1527**	1207	380	650	1059	524	995	72	13121
	JDart-2021	1410	961	398	1132	380	560	907	266	531	43	6588
	COHEC	**8789**	**4147**	917	**4375**	**996**	**1398**	**3980**	**2047**	**1953**	**155**	**28757**
Relative to Base		+14.01%	+48.80%	+169.71%	+247.77%	+162.11%	+116.74%	+214.38%	+306.96%	+97.07%	+106.67%	+80.25%
Valuations	Base	5422	2689	340	1252	380	593	1097	498	949	75	13295
	JDart-2020	2672	1608	365	1098	380	526	842	414	658	70	8633
	JDart-2021	687	782	266	1020	380	428	588	209	426	41	4827
	COHEC	**6180**	**3612**	**751**	**3652**	**724**	**1167**	**3527**	**1837**	**1686**	**112**	**23248**
Relative to Base		+13.98%	+34.33%	+120.88%	+191.69%	+90.53%	+96.80%	+221.51%	+268.88%	+77.66%	+49.33%	+74.86%
Exceptions	Base	760	352	76	117	0	71	178	88	291	2	1935
	JDart-2020	628	297	44	104	0	71	129	89	294	2	1658
	JDart-2021	248	109	32	55	0	36	52	28	74	2	636
	COHEC	**1023**	**1072**	**383**	**1867**	**275**	**356**	**585**	**802**	**692**	**25**	**7080**
Relative to Base		+34.61%	+204.55%	+403.95%	+1495.73%	-	+401.41%	+228.65%	+801.12%	+137.80%	+1150.00%	+265.89%

Table 3: RQ2: Comparison between COHEC and CONCOLLMIC on 30 sampled `commons-lang3` APIs.

Metric	CONCOLLMIC	COHEC	Improve
Covered Lines	355	405	↑14.08%
Covered Instructions	1,829	2,004	↑9.57%
Covered Branches	174	201	↑15.52%
Time (s)	11,167	3,894	↓65.13%
Input Token	25,528,037	171,939	↓99.33%
Output Token	277,746	133,061	↓52.09%
Total Token	25,805,783	305,000	↓98.82%
Cost ($)	80.75	2.51	↓96.89%

and exceptional executions on this library. *(3) Exceptional executions.* COHEC records more exceptional executions than Base on all 10 libraries, and the aggregate increase is larger than the corresponding increase in *Paths*. These counts represent exceptional executions rather than unique faults.

Answer to RQ1: COHEC substantially improves exploration effectiveness over baselines. Heap constraint solving with trusted verification enables concrete executions that remain blocked under SMT-only concolic execution.

RQ2: Comparison with SOTA. We compare COHEC with CONCOLLMIC on a stratified sample of 30 APIs from `commons-lang3`. We report the numbers of lines, instructions, and branches covered according to JaCoCo, together with wall-clock time, token usage, and monetary cost in Table 3.

Results & analysis. (1) *Effectiveness.* COHEC covers 14.08% more lines, 9.57% more instructions, and 15.52% more branches than CONCOLLMIC (Table 3). (2) *Efficiency.* COHEC reduces total execution time by 65.13%. (3) *Cost.* COHEC reduces total tokens by 98.82% and monetary cost by 96.89% while covering more program elements, suggesting that COHEC obtains higher testing effectiveness with far less computational and monetary overhead.

Answer to RQ2: Compared to the SOTA baseline CONCOLLMIC, COHEC covers more lines, instructions, and branches while requiring significantly less time and fewer tokens.

RQ3: Trusted Verification and Recovery. To assess the practical role of trusted verification, we analyze how often candidate models are rejected by the trusted verifier and how often bounded refinement recovers these rejected candidates across all 10 libraries. Figure 3 reports the number of verifier rejections and bounded refinement recoveries for each library.

Results & analysis. Across all libraries, 13,879 candidate models are rejected by trusted verification, and bounded refinement recovers 9,267 (66.77%). This confirms that invalid candidates are common and justifies the trust boundary, while showing that many failures are repairable. Recovery rates vary from 27.93% (`xchart`) to 90.34% (`threetenbp`), indicating that unresolved cases are library-dependent.

Answer to RQ3: Trusted verification is necessary in practice: candidate models are frequently rejected by the trusted verifier, and bounded refinement recovers a substantial fraction of these rejected candidates as accepted candidate models.

RQ4: Ablation. To assess the contribution of each component in COHEC, we compare full COHEC with three ablated variants: COHEC without history retrieval, COHEC without type constraints, and COHEC without heap state constraints. The two constraint

Table 4: RQ4: Ablation results.

Metrics	Tool	Library										Total	Δ
		lang3	guava	ical4j	jfreechart	jgrapht	joda-time	sis-utility	threetenbp	time4j	xchart		
Paths	COHEC	**8789**	**4147**	**917**	**4375**	**996**	1398	3980	**2047**	1953	155	**28757**	
	w/o History	8721	3887	750	3057	855	1442	**4052**	1960	**2575**	**167**	27466	-4.49%
	w/o Type	8722	3882	757	2997	896	**1453**	2247	1912	1837	159	24862	-13.54%
	w/o Heap	7940	3159	356	1838	383	692	1512	507	1014	80	17481	-39.21%
Valuations	COHEC	**6180**	**3612**	**751**	**3652**	**724**	1167	3527	**1837**	1686	112	**23248**	
	w/o History	6153	3319	591	2562	644	**1203**	**3602**	1640	**2226**	**128**	22068	-5.08%
	w/o Type	6162	3307	608	2520	682	1193	1829	1613	1541	124	19579	-15.78%
	w/o Heap	5419	2844	341	1582	382	599	1169	489	958	77	13860	-40.38%
Exceptions	COHEC	**1023**	**1072**	**383**	**1867**	**275**	356	**585**	**802**	692	25	**7080**	
	w/o History	993	861	251	1020	203	**365**	577	645	**1259**	**35**	6209	-12.30%
	w/o Type	996	845	262	1018	229	361	579	633	572	32	5527	-21.94%
	w/o Heap	776	499	80	447	2	81	205	76	307	4	2477	-65.01%

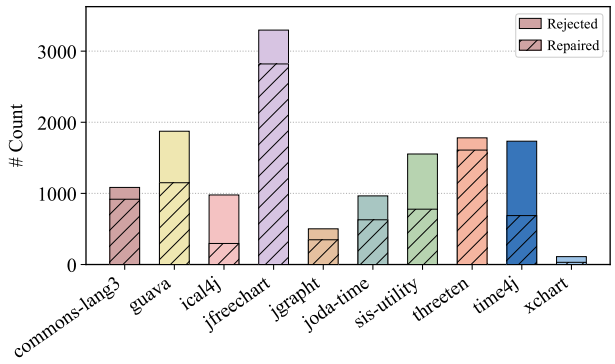

Figure 3: RQ3: verifier rejections and bounded refinement recoveries across libraries

categories align with the construction stages of Section 3: *type constraints* correspond to the runtime type assignment stage and cover dispatch-induced ExactType atoms, while *heap state constraints* correspond to the object identity and heap shape construction stage and cover nullness, aliasing, and subtype atoms. All variants are evaluated using *Paths*, *Valuations*, and *Exceptions* in Table 4.

Results & analysis. All three components contribute. Removing history retrieval reduces total *Paths*, *Valuations*, and *Exceptions* by 4.49%, 5.08%, and 12.30%; removing type constraints causes larger drops of 13.54%, 15.78%, and 21.94%; and removing heap state constraints causes the largest drops of 39.21%, 40.38%, and 65.01%. Some ablated variants are slightly better on individual libraries, likely because history and type guidance are heuristic, but full COHEC remains the best configuration overall.

Answer to RQ4: All three components contribute to COHEC's overall performance, and heap state constraints contribute the most.

5.4 Threats to Validity

Internal validity. LLM nondeterminism is the primary internal threat, mitigated by schema-constrained communication, same-seed initial states, per-API budgets, and trusted-verifier-based acceptance. RQ1 uses DeepSeek-Chat v3.2 (for cost reasons at 6,750-API scale) while RQ2 uses Claude Sonnet 4.5 (to align with CONCOLLMIC); each comparison is internally controlled but the two are not cross-comparable on model strength.

External validity. We wrap every target Java library API in a single-entry harness, so the results may not transfer to other languages, applications with complex setup, or scenarios requiring whole-program analysis. RQ2 is further limited to 30 APIs from a single library (`commons-lang3`) because instrumentation for the other candidate libraries did not complete. More broadly, the ten subject libraries come from prior work and may be biased toward well-maintained utility-style code; highly stateful, reflective, concurrent, or native-heavy systems may require richer context construction or different heap model construction strategies.

Construct validity. We use *Paths*, *Valuations*, and *Exceptions* as the primary full-scale metrics. *Paths* directly captures COHEC's ability to enter executions previously blocked by unresolved heap constraints, while *Valuations* and *Exceptions* capture successful re-execution and exceptional outcomes on those paths. Exceptional executions do not necessarily correspond to unique faults. These metrics therefore measure exploration and re-execution success rather than end-to-end bug-finding ability, and the JaCoCo covered element counts in RQ2 remain only standard proxies for testing effectiveness.

6 Discussion

Trusted but Incomplete Solving. COHEC is fail-closed but incomplete. If candidate construction and bounded refinement fail, HCS returns UNKNOWN and the driver retains the Base result. Every reported SAT result passes the four trusted checks, keeping the LLM outside the trusted computing base. However, the guarantee is relative to the recorded path condition and covers only properties

represented in PC_{heap}; it does not establish full semantic equivalence beyond those path-relevant properties.

Verification Failure Modes. Compilation failure accounts for over half of the 13,879 rejections. Initializers for deep object graphs must correctly sequence constructors, resolve overloads, and respect access control; one incorrect type, cast, or inaccessible constructor invalidates the result. Precise `javac` diagnostics make bounded refinement most effective for these failures, whereas schema, constraint-satisfaction, and heap-realization failures often require revisiting earlier construction stages. Incremental compilation during synthesis may further improve acceptance.

Applicability Boundary. The design works best when the required heap is constructible from selected program context and a bounded collection of objects. It is less effective for long-range invariants, reflection, native code, concurrency, or environment-dependent behavior outside the supported heap atoms. Thus, our results demonstrate practicality for many library-style APIs, not general heap reasoning.

Cost and Deployment Tradeoffs. Across the benchmark, HCS is invoked 11,047 times and consumes 93.0M tokens; tokens per accepted valuation range from 1,813 on `commons-lang3` to 14,563 on `xchart`. Cost grows with both invocation frequency and the heap context exposed. Selective invocation and bounded context keep easier workloads affordable, while harder cases may require better templates or tighter policies.

7 Related Work

Symbolic and concolic execution. Symbolic execution originates with King [26]; DART and CUTE established the concolic pattern [20, 46]; and EXE, KLEE, and later surveys mark the field's maturation [2, 8–10]. In the Java ecosystem, JPF, SPF, and JDart provide bytecode-level concolic analysis [31, 38, 55]; Pex targets .NET [53]; and synthetic symbolic execution applies concolic analysis to Android by synthesizing models for framework interactions [19]. A parallel thrust improves throughput and scalability: SAGE and Mayhem combine symbolic reasoning with whitebox fuzzing [11, 21]; Driller invokes concolic execution selectively when fuzzing stalls [52]; QSYM, SymCC, and SYMSAN redesign the engine for speed [12, 42, 61]; temporal-logic-guided greybox fuzzing directs exploration toward behavioral properties [34]; while S2E, Cloud9, angr, and Manticore offer versatile multi-environment platforms [7, 15, 35, 49]. COHEC does not compete on path scheduling or throughput; it targets the heap constraint bottleneck in object-oriented code, where branch feasibility depends on nullness, aliasing, runtime types, field reachability, and dynamic dispatch.

Heap reasoning and structured input construction. Lazy initialization defers heap decisions until field accesses force a commitment [25]; JBSE extends this with Heap EXploration Logic for complex Java heap inputs [5]; Precise Lazy Initialization tightens nondeterminism with structural invariants [16]; and POSE achieves path-optimal symbolic execution by eliminating the combinatorial trace explosion of lazy initialization [4]. Separation-logic-guided symbolic execution, as in Java StarFinder, uses inductive heap predicates to generate fully initialized inputs [39, 40]. TestEra, Korat, and SuSLik treat heap structure as something that must be explicitly built from declarative or separation logic specifications [3, 33, 43]. These approaches keep heap construction inside the symbolic engine or a bounded formalism. COHEC instead isolates PC_{heap} for a target path and delegates it to HCS, accepting only heap models that survive trusted verification.

Machine learning for symbolic execution. Green reuses constraints across analyses [54]; STASE guides exploration with static analysis [47]; MLBSE and Learch learn search strategies from coverage feedback [6, 23]; NeuEx approximates hard-to-model behaviors with neural constraints [48]; and SmarTest uses a language model to steer smart contract symbolic exploration [50]. The proposer/verifier split in COHEC also echoes CEGIS [51]. These techniques are complementary: they target search order or approximate solving, whereas COHEC confines its untrusted proposer to proposing candidate heap models for the target path's PC_{heap}, with semantic commitment on the verifier side.

LLM-based test generation and program reasoning. CodaMosa queries an LLM when search-based generation stalls [27]; TestPilot and ChatUniTest generate unit tests end-to-end from signatures and context [13, 45]; MuTAP augments prompts with surviving mutants [17]; CoverUp and HITS refine tests via coverage feedback or method slicing [1, 56]; SymPrompt and PANTA combine LLMs with program analysis [22, 44]; and recent studies evaluate these approaches systematically [24, 59]. Other work uses LLMs to partition the input space of library APIs [29], extract formal specifications from documents for test automation [28], or generate carved test cases and assertions in industrial practice [60]. LLMs have also been applied to property-based testing [58], invariant inference [41, 57], and hybrid analysis. Closer to symbolic execution, AutoBug uses LLM reasoning over decomposed paths [30], while CONCOLLMIC embeds LLMs in instrumentation, summarization, and solving [32]. COHEC is more conservative: it confines the LLM to proposing candidate heap models for PC_{heap}, and every result must survive compilation, execution, and semantic re-verification under a fail-closed trust model.

8 Conclusion

We presented COHEC, a verifier-in-the-loop solver for heap constraints in concolic execution of object-oriented programs. COHEC separates primitive constraints for SMT solving from heap constraints handled by HCS and accepts only verified heap models, keeping the LLM outside the trusted computing base. Across 6,750 APIs from ten Java libraries, COHEC substantially improves over JDart and offers a better coverage–cost tradeoff than CONCOLLMIC, demonstrating the effectiveness of verified heap model construction for object-sensitive behaviors.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant No. 62472100).

Data Availability Statement

The source code, experimental subjects, evaluation scripts, precomputed logs, generated test suites, and data underlying the results in this paper are publicly available in the COHEC artifact on Figshare [14]. The versioned archive includes documentation and a Docker configuration for reproducing the experiments.

References

[1] Juan Altmayer Pizzorno and Emery D Berger. 2025. CoverUp: Effective High Coverage Test Generation for Python. In *Proceedings of the ACM on Software Engineering*, Vol. 2. ACM New York, NY, USA, 2897–2919. doi:10.1145/3729398

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3 (2018), Article 50. doi:10.1145/3182657

[3] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated Testing Based on Java Predicates. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. Association for Computing Machinery, New York, NY, USA, 123–133. doi:10.1145/566172.566191

[4] Pietro Braione, Giovanni Denaro, and Luca Guglielmo. 2026. Path-Optimal Symbolic Execution of Heap-Manipulating Programs. In *Proceedings of the 33rd IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 957–968. doi:10.1109/SANER67736.2026.00112

[5] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. 2016. JBSE: A Symbolic Executor for Java Programs with Complex Heap Inputs. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. Association for Computing Machinery, New York, NY, USA, 1018–1022. doi:10.1145/2950290.2983940

[6] Lei Bu, Yongjuan Liang, Zhunyi Xie, Hong Qian, Yi-Qi Hu, Yang Yu, Xin Chen, and Xuandong Li. 2021. Machine Learning Steered Symbolic Execution Framework for Complex Software Code. *Formal Aspects of Computing* 33 (2021), 301–323. doi:10.1007/s00165-021-00538-3

[7] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel Symbolic Execution for Automated Real-World Software Testing. In *Proceedings of the 6th ACM European Conference on Computer Systems (EuroSys)*. Association for Computing Machinery, New York, NY, USA, 183–198. doi:10.1145/1966445.1966463

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Berkeley, CA, USA, 209–224. https://www.usenix.org/conference/osdi-08/klee-unassisted-and-automatic-generation-high-coverage-tests-complex-systems

[9] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2008. EXE: Automatically Generating Inputs of Death. *ACM Transactions on Information and System Security* 12, 2 (2008), 10:1–10:38. doi:10.1145/1455518.1455522

[10] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (2013), 82–90. doi:10.1145/2408776.2408795

[11] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP)*. IEEE, Los Alamitos, CA, USA, 380–394. doi:10.1109/SP.2012.31

[12] Ju Chen, Wookhyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. 2022. SYMSAN: Time and Space Efficient Concolic Execution via Dynamic Data-Flow Analysis. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*. USENIX Association, Berkeley, CA, USA, 2531–2548. https://www.usenix.org/conference/usenixsecurity22/presentation/chen-ju

[13] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion)*. Association for Computing Machinery, New York, NY, USA, 572–576. doi:10.1145/3663529.3663801

[14] Leyi Cheng and Shaoran Xia. 2026. COHEC. Figshare. Software, version 1. doi:10.6084/m9.figshare.31868686.v1

[15] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems* 30, 1 (2012), 2:1–2:49. doi:10.1145/2110356.2110358

[16] Juan Manuel Copia, Facundo Molina, Nazareno Aguirre, Marcelo F. Frias, Alessandra Gorla, and Pablo Ponzio. 2023. Precise Lazy Initialization for Programs with Complex Heap Inputs. In *Proceedings of the 34th IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Los Alamitos, CA, USA, 752–762. doi:10.1109/ISSRE59848.2023.00080

[17] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2024. Effective Test Generation Using Pre-trained Large Language Models and Mutation Testing. *Information and Software Technology* 171 (2024), 107468. doi:10.1016/j.infsof.2024.107468

[18] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science, Vol. 4963)*. Springer, Berlin, Heidelberg, 337–340. doi:10.1007/978-3-540-78800-3_24

[19] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android Testing via Synthetic Symbolic Execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. doi:10.1145/3238147.3238225

[20] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, New York, NY, USA, 213–223. doi:10.1145/1065010.1065036

[21] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Commun. ACM* 55, 3 (2012), 40–44. doi:10.1145/2093548.2093564

[22] Sijia Gu, Noor Nashid, and Ali Mesbah. 2025. LLM Test Generation via Iterative Hybrid Program Analysis. arXiv:2503.13580 doi:10.48550/arXiv.2503.13580

[23] Jingxuan He, Gishor Sivanrupan, Petar Tsankov, and Martin Vechev. 2021. Learning to Explore Paths for Symbolic Execution. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Association for Computing Machinery, New York, NY, USA, 2526–2540. doi:10.1145/3460120.3484813

[24] Zongze Jiang, Ming Wen, Jialun Cao, Xuanhua Shi, and Hai Jin. 2024. Towards Understanding the Effectiveness of Large Language Models on Directed Test Input Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, New York, NY, USA, 1408–1420. doi:10.1145/3691620.3695513

[25] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science, Vol. 2619)*. Springer, Berlin, Heidelberg, 553–568. doi:10.1007/3-540-36577-X_40

[26] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. doi:10.1145/360248.360252

[27] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, Los Alamitos, CA, USA, 919–931. doi:10.1109/ICSE48619.2023.00085

[28] Hui Li, Zhen Dong, Siao Wang, Hui Zhang, Liwei Shen, Xin Peng, and Dongdong She. 2025. Extracting Formal Specifications from Documents Using LLMs for Test Automation. In *Proceedings of the 33rd IEEE/ACM International Conference on Program Comprehension (ICPC)*. doi:10.1109/ICPC66645.2025.00039

[29] Jiageng Li, Zhen Dong, Chong Wang, Haozhen You, Cen Zhang, Yang Liu, and Xin Peng. 2025. LLM Based Input Space Partitioning Testing for Library APIs. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. 1436–1448. doi:10.1109/ICSE55347.2025.00153

[30] Yihe Li, Ruijie Meng, and Gregory J. Duck. 2025. Large Language Model Powered Symbolic Execution. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2, Article 385 (Oct. 2025), 29 pages. doi:10.1145/3763163

[31] Kasper Luckow, Marko Dimjasevic, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamaric, and Vishwanath Raman. 2016. JDart: A Dynamic Symbolic Analysis Framework. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science, Vol. 9636)*. Springer, Berlin, Heidelberg, 442–459. doi:10.1007/978-3-662-49674-9_26

[32] Zhengxiong Luo, Huan Zhao, Dylan Wolff, Cristian Cadar, and Abhik Roychoudhury. 2026. Agentic Concolic Execution. In *Proceedings of the 2026 IEEE Symposium on Security and Privacy (SP)*. IEEE, Los Alamitos, CA, USA, 1–19. doi:10.1109/SP63933.2026.00003

[33] Darko Marinov and Sarfraz Khurshid. 2001. TestEra: A Novel Framework for Automated Testing of Java Programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)*. IEEE, Los Alamitos, CA, USA, 22–31. doi:10.1109/ASE.2001.989787

[34] Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. 2022. Linear-time Temporal Logic Guided Greybox Fuzzing. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE)*. doi:10.1145/3510003.3510082

[35] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Los Alamitos, CA, USA, 1186–1189. doi:10.1109/ASE.2019.00133

[36] Malte Mues and Falk Howar. 2020. JDart: Dynamic symbolic execution for Java bytecode (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 398–402. doi:10.1007/978-3-030-45237-7_28

[37] Malte Mues and Falk Howar. 2021. JDart: Portfolio solving, breadth-first search and SMT-Lib strings (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 448–452. doi:10.1007/978-3-030-72013-1_30

[38] Corina S. Păsăreanu, Willem Visser, David H. Bushnell, Jaco Geldenhuys, Peter C. Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: Integrating Symbolic Execution with Model Checking for Java Bytecode Analysis. *Automated Software Engineering* 20 (2013), 391–425. doi:10.1007/s10515-013-0122-2

[39] Long H. Pham, Quang Loc Le, Quoc-Sang Phan, Jun Sun, and Shengchao Qin. 2018. Testing Heap-Based Programs with Java StarFinder. In *Proceedings of the 40th ACM/IEEE International Conference on Software Engineering: Companion (ICSE-Companion)*. Association for Computing Machinery, New York, NY, USA, 268–269. doi:10.1145/3183440.3194964

[40] Long H. Pham, Quang Loc Le, Quoc Sang Phan, Jun Sun, and Shengchao Qin. 2019. Enhancing Symbolic Execution of Heap-Based Programs with Separation Logic for Test Input Generation. In *Proceedings of the 17th International Symposium on Automated Technology for Verification and Analysis (ATVA) (Lecture Notes in Computer Science, Vol. 11781)*. Springer, Berlin, Heidelberg, 209–227. doi:10.1007/978-3-030-31784-3_12

[41] Muhammad A. A. Pirzada, Giles Reger, Ahmed Bhayat, and Lucas C. Cordeiro. 2024. LLM-Generated Invariants for Bounded Model Checking Without Loop Unrolling. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, New York, NY, USA, 1395–1407. doi:10.1145/3691620.3695512

[42] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic Execution with SymCC: Don't Interpret, Compile!. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*. USENIX Association, Berkeley, CA, USA, 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[43] Nadia Polikarpova and Ilya Sergey. 2019. Structuring the Synthesis of Heap-Manipulating Programs. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 72:1–72:30. doi:10.1145/3290385

[44] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A Study of Coverage Guided Test Generation in Regression Setting using LLM. In *Proceedings of the ACM on Software Engineering*, Vol. 1. ACM New York, NY, USA, 951–971. doi:10.1145/3643769

[45] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. doi:10.1109/TSE.2023.3334955

[46] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Association for Computing Machinery, New York, NY, USA, 263–272. doi:10.1145/1081706.1081750

[47] Md Shafiuzzaman, Achintya Desai, Laboni Sarker, and Tevfik Bultan. 2024. STASE: Static Analysis Guided Symbolic Execution for UEFI Vulnerability Signature Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, New York, NY, USA, 1783–1794. doi:10.1145/3691620.3695543

[48] Shiqi Shen, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, and Prateek Saxena. 2019. Neuro-Symbolic Execution: Augmenting Symbolic Execution with Neural Constraints. In Proceedings of the Network and Distributed System Security Symposium (NDSS). doi:10.14722/ndss.2019.23530

[49] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, Los Alamitos, CA, USA, 138–157. doi:10.1109/SP.2016.17

[50] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*. USENIX Association, Berkeley, CA, USA, 1361–1378. https://www.usenix.org/conference/usenixsecurity21/presentation/so

[51] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodik, and Kemal Ebcioglu. 2005. Programming by Sketching for Bit-streaming Programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, New York, NY, USA, 281–294. doi:10.1145/1065010.1065045

[52] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Internet Society. doi:10.14722/ndss.2016.23368

[53] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex – White Box Test Generation for .NET. In *Tests and Proofs (TAP) (Lecture Notes in Computer Science, Vol. 4966)*. Springer, Berlin, Heidelberg, 134–153. doi:10.1007/978-3-540-79124-9_10

[54] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE). doi:10.1145/2393596.2393665

[55] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. 2004. Test Input Generation with Java PathFinder. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. Association for Computing Machinery, New York, NY, USA, 97–107. doi:10.1145/1007512.1007526

[56] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1258–1268. doi:10.1145/3691620.3695501

[57] Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. 2024. LLM Meets Bounded Model Checking: Neuro-symbolic Loop Invariant Inference. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, New York, NY, USA, 406–417. doi:10.1145/3691620.3695014

[58] Yiheng Xiong, Ting Su, Jue Wang, Jingling Sun, Geguang Pu, and Zhendong Su. 2024. General and Practical Property-based Testing for Android Apps. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, New York, NY, USA, 53–64. doi:10.1145/3691620.3694986

[59] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. 2024. On the Evaluation of Large Language Models in Unit Test Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3691620.3695529

[60] Haozhen You, Zhen Dong, Jingjing Wang, Qiang Li, and Xin Peng. 2026. Industrial Practice of LLM-based Test Case Carving and Assertion Generation. In *Proceedings of the 35th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. doi:10.48550/arXiv.2607.24000

[61] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*. USENIX Association, Berkeley, CA, USA, 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

Received 2026-03-26; accepted 2026-06-18