Allegheny College
Computer And Information Science Department

Senior Thesis

Hybrid Analysis of Object Escapes

by

Grant Anderson

ALLEGHENY COLLEGE

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Project Supervisor: **Professor Gregory M. Kapfhammer**
Co-Supervisor: **Student Second Reader**

Abstract

Object escapes keep data alive past its intended lifetime, creating performance, stability, and correctness risks across runtimes. This paper presents Graphene-HA, a fused-evidence escape-analysis workflow that unifies static analysis with heap-growth tracking across Python, Java, JavaScript, Go, and Rust under a single command and report format. The system combines a Python CLI, a Rust orchestration core, and language-specific bridges that directly measure heap allocation deltas during execution. Each bridge uses native runtime introspection (tracemalloc for Python, MemoryMXBean for Java, process.memoryUsage for Node.js, runtime.MemStats for Go, and custom GlobalAlloc for Rust) to capture heap growth as evidence of object escape. We evaluate the workflow on 1,421 labeled targets spanning five languages. Overall accuracy improves from 61.2% with static analysis to 83.7% with combined analysis, while dynamic heap tracking achieves 100.0% accuracy on this benchmark, with an F1 score of 1.0. The largest combined gains are observed in Python, where F1 rises from 20.2% to 92.0%, and in Java, where F1 rises from 31.5% to 100.0%. Go and Rust remain constrained by static false-positive patterns when evidence is combined. These results suggest that direct heap measurement is a reliable dynamic signal for object escape detection, and that a unified combined pipeline can substantially improve correctness compared to static analysis alone. Future work will extend Graphene-HA to additional languages and to stronger detection patterns for edge cases in Go and Rust.

Table of contents

List of Figures

List of Tables

1 Introduction

This project addresses a bounded software-lifecycle problem in which objects remain alive after the function that created them has returned. It examines how these failures appear in production settings, including request pipelines, background processing, and long-running services. When escaped objects remain reachable longer than intended, they keep stale state in circulation and increase the chance of downstream correctness and stability issues. Escape analysis is therefore a memory-safety and lifecycle-integrity concern because it asks whether references are retained and reused only where intended. The following sections explain why object escapes matter in production systems, situate the framework against prior analysis approaches, state the project goals, and conclude with the ethical constraints on use.

1.1 Motivation

An object escape occurs when temporary data created for one task is stored in a longer-lived location. When the task ends, the data remains reachable.

The following example demonstrates the basic form of the problem. The `make_escape` function creates a local object, but instead of allowing it to end with the function, it appends the object to a global list. Each call to `make_escape` adds another object to that list, which means the memory associated with those objects is retained indefinitely.

```
global_store = []

def make_escape():
  local_obj = {"x": 1}
  global_store.append(local_obj)  # escape: stored in global scope
```

Every call to `make_escape()` is intended to complete and release its temporary state, but the object remains in `global_store`. If the function is invoked repeatedly, the container grows over time. In production systems, this appears as slowdowns, memory pressure, increased operational cost, and eventually crashes or restarts. Escape-related failures also represent a correctness problem. If stale state remains reachable, later work can read outdated data and make an incorrect decision. For users, this appears as inconsistent outputs. For operators, it appears as incidents that are difficult to reproduce and diagnose.

The same lifecycle pattern appears across infrastructure, where CI services, repository indexing, package analysis, and automation systems all split work into immediate and deferred phases. If the deferred path holds full request objects instead of smaller snapshots, escaped references accumulate quietly until instability becomes visible.

This pattern recurs across languages despite syntactic differences. In Python, a queued callable can keep a large object alive. In Java, deferred workers can retain request state after response completion. In JavaScript, timer callbacks can hold stale closures. In Go and Rust, background workers can preserve references through queue or channel paths. This cross-language consistency is one reason a unified detection workflow is useful because the engineering pattern changes less than the surface syntax suggests.

Graphene addresses this problem by structuring escape diagnosis as a repeatable, cross-language workflow. Findings can then be compared within one operational model rather than interpreted as isolated, tool-specific outputs. The static stage highlights likely escape paths. The dynamic stage tests whether objects are actually retained during execu-

tion through direct heap-allocation observation. Standardized session artifacts further support comparison, separate target-code escapes from tooling failures, and ground ownership-lifetime judgments in evidence rather than inference.

1.2 Current State of the Art

Object-escape reasoning has a substantial research base that spans compiler analysis, optimization, and runtime validation. Foundational work by Choi et al. established connection-graph methods and interprocedural summary strategies, making object-lifetime analysis practical for real compilers [1, 2]. Those ideas continue to shape engineering judgments about whether an object remains local to a scope or becomes reachable from a wider execution context.

Building on these foundations, later work expanded these ideas beyond all-or-nothing decisions. Partial and interprocedural techniques showed that lifetime behavior can depend strongly on path structure and call context, which improved precision in complex programs [10, 12]. At the same time, correctness-focused research warned that partial reasoning must be interpreted carefully and with explicit assumptions [7].

More recent work confirms that escape-analysis improvements remain relevant in modern toolchains. In the Go ecosystem, studies report measurable reduction in allocation pressure when lifetime precision improves [3, 11]. These results matter for Graphene because they show that object-lifetime analysis is not just historical compiler theory. It continues to deliver practical value in contemporary systems.

Beyond single-language optimization, cross-language analysis practice highlights another reality in which tools differ not only in precision, but also in reporting style and operational expectations [6]. Runtime-oriented validation tools reinforce this point by showing that executable evidence is often necessary to confirm how lifetime behavior unfolds under actual workloads [4]. Safety-critical studies also emphasize that lifetime misclassification can affect assurance and predictability, not only speed [5].

Collectively, the literature provides strong building blocks. A practical gap remains, however, for mixed-language engineering teams. Comparable analyses are still difficult to run, outputs are hard to interpret consistently, and tool-integration failures can be mistaken for target-code findings within the same investigation cycle.

To consolidate these strands, Table 1 summarizes the major approach families, their strengths, and their recurring practical limits.

Table 1: State-of-the-art approaches and their practical fit for object-escape investigations

Approach family	What it contributes	Typical limitation in day-to-day use
Classical escape-analysis frameworks	Formal object-lifetime reasoning and reusable summaries [1, 2]	Usually embedded in language-specific compiler contexts
Partial and path-sensitive methods	Better precision for branch-dependent lifetime behavior [10, 12]	Interpretation still depends on explicit assumptions [7]

Approach family	What it contributes	Typical limitation in day-to-day use
Modern applied optimization studies	Evidence that improved escape precision yields measurable practical gains [3, 11]	Focus is often optimization workflow, not cross-language operational triage
Cross-language static-analysis practice	Demonstrates ecosystem differences in tooling expectations [6]	Output conventions are hard to compare directly across stacks
Runtime semantic validation tools	Strong evidence for real execution behavior [4]	Often specialized to one language/runtime and not unified with static evidence

Against this background, Graphene is positioned to address this integration gap. It does not attempt to replace every specialized analyzer. Instead, it treats object escapes as a primary target and provides one execution and reporting surface across Python, Java, JavaScript/Node.js, Go, and Rust.

The current implementation strengthens this position through a unified `uv run graphene` workflow, standardized capability metadata, and stronger protocol normalization for bridge outputs and startup failures. The practical goal is not only to find escape paths, but also to make findings easier to compare, reproduce, and apply in real investigations.

1.2.1 Feasibility and Distinct Contributions

The project is feasible because it combines a bounded problem scope with existing infrastructure. The scope is intentionally limited to object escapes and lifecycle-boundary violations rather than all defect classes. Implementation risk is reduced by using language-native bridges behind one orchestrator instead of forcing a single universal intermediate representation. Operational feasibility is also demonstrated by the current artifact. Graphene executes through one command surface across Python, Java, JavaScript/Node.js, Go, and Rust, and produces structured per-session reports. Feasibility is further supported by reproducible evaluation scaffolding. Each language has an annotated 100-case suite with explicit ESCAPE or SAFE intent, and each run records success, crash, escape, and timing data in a common schema. This allows progress to be measured incrementally and compared across languages without redesigning the workflow for each ecosystem.

Relative to comparable object-escape analyzers, Graphene provides five distinct contributions:

1. A cross-language object-escape investigation workflow with one command model and one report structure.
2. A unified capability-metadata layer that makes language coverage and limits explicit before a run starts.
3. Static and runtime evidence collection under one orchestrated session so findings can be cross-checked in one cycle.
4. Normalized bridge and startup diagnostics that separate tooling failures from target-program escape findings.
5. Reproducible session artifacts and benchmark suites that support repeated runs, before/after comparison, and longitudinal tracking.

1.3 Goals

These goals are interdependent. Detection quality has limited value if the workflow remains fragmented, and workflow consistency is insufficient if findings are hard to interpret under operational pressure.

1.3.1 Real World Detection

The primary objective is to detect object-escape behavior that appears in production systems, including retained callback captures, queue jobs that hold stale request state, deferred tasks without clear closure, and resource lifetimes that outlive their intended scope.

Evaluation emphasizes two outcomes, namely sensitivity to known escape patterns in representative cases and interpretability of findings in structured session artifacts.

1.3.2 Unified Workflow

The second objective is workflow consistency across languages, which supports direct cross-stack comparison during one investigation cycle. The command model and report structure are kept stable so users can compare findings without relearning assumptions for each runtime ecosystem.

Internally, language-specific bridges remain native to their environments. Externally, orchestration and reporting remain consistent.

1.3.3 Low Barrier of Entry

To make that workflow operational in practice, the third objective reduces onboarding cost. Many analysis workflows assume specialist familiarity with internal toolchain details. Graphene reduces that barrier through a constrained interface, sensible defaults, and artifact formats that are readable without custom scripts.

Accessibility here does not mean reduced rigor. It means a faster path from question to evidence for both students and practitioners.

1.3.4 Modular Design

The fourth objective preserves research headroom through incremental extension. Additional language bridges, stronger static heuristics, and richer reporting should be possible without full-system redesign. This architectural headroom is necessary for long-term research value.

1.3.5 System Workflow Overview

The current pipeline presents a simple user-facing workflow.

```
[User command: uv run graphene analyze ...]
  -> [Python CLI entrypoint]
  -> [Rust orchestrator]
  -> [Mode routing: static | dynamic | both]
  -> [Language analyzer execution]
  -> [Result aggregation and escape classification]
  -> [Session report generation]
```

```
-> [artifacts/logs/<language>/session_<timestamp>_<id>/]
     |- README.md
     |- results.csv
     `- vulnerabilities.md (when findings exist)
```

This workflow supports investigation loops used during live incidents. Teams typically hypothesize a lifetime boundary, test likely escape paths, confirm behavior under execution, patch ownership closure, and repeat analysis for verification.

1.3.6 Scope and Capabilities

Within its current scope, Graphene supports Python, Java, JavaScript/Node.js, Go, and Rust targets, and it can execute static analysis, runtime analysis, or both in one run. Each session produces both machine-readable and human-readable artifacts. The implementation emphasizes unified UV-based command routing, standardized capability metadata, normalized bridge diagnostics, and a predictable per-session output structure. Clear boundaries are maintained for credible evaluation. Graphene is not a universal defect detector and currently focuses on object escapes and related lifecycle violations. This boundary improves interpretability and avoids overclaiming, while still leaving room for future expansion to broader defect classes.

Evaluation reproducibility is a primary design goal for this project. Complete reproducibility protocols, including exact command sequences and artifact verification procedures, are provided in Section 4 to enable independent re-runs and verification of all reported results.

1.4 Ethical Implications

Like most reliability and security tooling, Graphene has dual-use potential. Capabilities that help maintainers identify latent defects can also support adversarial analysis when source access is available.

1.4.1 Risk and Misuse

The clearest risk is pre-patch exposure. If harmful actors analyze a codebase before maintainers deploy fixes, practical exploitability can increase. Graphene partly mitigates this risk by operating as a local workflow rather than a hosted remote scanning service, but this mitigation is limited. Responsible use still depends on governance, including clear intended-use policy, coordinated disclosure, and disciplined patch cycles. A second risk is false confidence. A clean report cannot be treated as proof of overall software safety. Because Graphene is scoped, absence of findings is not evidence of universal correctness, and results should be interpreted as scoped evidence within a layered assurance strategy that still includes review, targeted tests, dependency analysis, and production observability.

1.4.2 Governance and Data Handling

Graphene stores artifacts locally in session directories. This design reduces default external exposure, but local artifacts can still contain sensitive runtime context such as identifiers, stack traces, and input values. Recommended controls include repository ignore rules for

generated logs, restricted artifact sharing, and sanitization before publication. Accessibility is also a governance concern because when advanced tooling is usable only by experts, teams with fewer resources are disadvantaged. A stable command surface and predictable output schema reduce that barrier, but accessibility must be paired with transparency about what is detected, how detection works, and where known limitations remain. Graphene provides the most value when used as a bounded lifecycle analysis tool with explicit scope language and clear communication of limits.

2 Related work

Object-escape analysis spans compiler theory, static-analysis engineering, and runtime validation. The most relevant prior work for this study can be grouped into four connected themes, namely foundational lifetime modeling, precision and correctness boundaries, operational validation patterns, and cross-language integration constraints. This structure keeps the chapter focused on ideas that most directly inform Graphene's design choices.

2.1 Foundations, Precision, and Correctness

Early Java work established object escape as an interprocedural lifetime and reachability question rather than a purely local syntax question [1, 2]. Connection-graph and summary methods made it practical to reason about whether objects remain stack-local or become reachable from broader program contexts.

That conceptual shift remains important because once lifetime is modeled as reachability, analysis can answer questions that local pattern inspection cannot, including cross-call ownership transfer and persistence beyond return boundaries. Published results from this lineage also showed practical impact, with notable stack-allocation opportunities and runtime improvements in selected benchmark profiles [1, 2].

Later work increased precision through partial and path-sensitive escape reasoning, showing that branch context and call-path structure can change escape outcomes materially [10, 9, 12]. Field and points-to refinements further reduced conservative retention assumptions in newer ecosystems [3].

These gains, however, come with correctness boundaries. Soundness-focused studies show that conclusions from partial escape information depend on explicit assumptions and use context [7]. For practical workflows, this implies that escape findings should be interpreted as scoped evidence with clear semantics and limits, not as universal proofs.

Another recurring point in this body of work is that precision improvements are rarely free. More context sensitivity can increase compile-time or analysis-time cost, and in production settings this tradeoff determines whether a technique is actually adopted. Several studies therefore frame precision as an engineering optimization problem that seeks to improve true signal quality enough to justify additional cost while avoiding complexity that makes analysis too fragile for repeated use.

This tradeoff matters for Graphene's architecture because Graphene is not positioned as a compiler optimization pass for one language runtime. Instead, it must provide acceptable precision and interpretability across multiple language ecosystems at once. The practical requirement is therefore two-dimensional and includes algorithmic quality and operational consistency. The related work supports this framing by showing that either dimension alone can fail in practice. High precision with opaque outputs is hard to operationalize, and simple outputs with weak precision are hard to trust.

Prior research also clarifies that benchmark outcomes should be interpreted by distribution, not by single aggregate score. Tail cases, branch-heavy workloads, and call-graph depth can produce very different escape behavior from median profiles. This is one reason modern analyses often report a mix of aggregate and case-dependent outcomes. The implication for this report is to treat per-language and per-mode variation as expected behavior rather than anomaly.

Table 2: Core research foundations and their implications for this report

Theme	Representative evidence	Practical implication
Interprocedural lifetime modeling	Java connection-graph lineage [1, 2]	Escape should be represented as ownership reachability across boundaries
Precision mechanisms	Partial/path-sensitive and context-aware work [10, 9, 12, 3]	Flow/context sensitivity materially affects signal quality
Correctness boundaries	Soundness limits for partial information [7]	Findings must include assumptions and confidence context

2.2 From Research to Operational Validation

A major trend in newer studies is a shift from "can we optimize?" toward "can we detect, validate, and trust outcomes in repeated engineering loops?" Applied Go work exemplifies this detect-transform-validate-measure cycle and reports practical gains in open-source and industrial settings [11].

Runtime validation research reinforces this operational orientation. Large-scale executable checking in Rust (e.g., Miri deployment at ecosystem scale) shows that runtime evidence can be integrated into normal development and CI practices [4]. Instrumentation-oriented workflows (e.g., ThreadSanitizer history) show that dynamic signals become most useful when they align with real-world code patterns and developer workflows [8].

For object-escape investigation specifically, the practical implication is that static and runtime evidence are complementary. Static methods identify suspicious lifetime paths. Runtime methods confirm which of those paths are exercised under actual inputs and schedules. This dual-evidence pattern is central to Graphene's methodology.

The process lessons from this literature are especially important for deployment in mixed-language repositories. Teams often need to repeat the same analysis cycle after refactoring, dependency updates, or infrastructure changes. A one-off analysis can be technically correct but still operationally weak if results are not reproducible, comparable, or diagnosable in later runs. The validation literature repeatedly emphasizes this point through workflow design, where run consistency, traceable assumptions, and explicit failure semantics are as important as top-line detection metrics.

Another transferable lesson is that dynamic evidence is most useful when it is integrated with clear scope control. Dynamic systems can always execute more inputs or explore more states, but practical workflows require bounded execution budgets. Prior runtime tooling practice therefore emphasizes targeted probes, deterministic reporting, and confidence-aware interpretation. Graphene's session-based dynamic execution model follows this pattern by pairing each run with explicit inputs, mode metadata, and reusable artifacts.

Finally, the operational studies suggest that trust is cumulative. Engineers tend to trust analysis output when they can verify that conclusions recur across independent runs and when false alarms are explained in consistent terms. This motivates protocol normalization and explicit summary metrics in Graphene's workflow, because consistency in interpretation is necessary for repeated use.

Table 3: Validation-oriented patterns and relevance

Validation pattern	Evidence	Relevance to Graphene
Repeated engineering loop (detect, validate, measure)	Applied Go engineering workflow [11]	Supports session-based re-runs and before/after comparisons
Executable semantic checking at scale	Rust runtime validation practice [4]	Motivates dynamic evidence as first-class output
Instrumentation with practical integration	Dynamic tooling history [8]	Supports bridge-level runtime observability design

2.3 Cross-Language Practice and Integration Gaps

Cross-language analysis research highlights that tool value is shaped not only by algorithmic precision but also by ecosystem expectations, output conventions, and operational fit [6]. The same concept can be interpreted differently across stacks if reports use inconsistent schemas or incompatible severity/confidence semantics.

This challenge is amplified in high-assurance contexts, where lifetime reasoning is tied to predictability and bounded resource behavior rather than only throughput gains [5]. In these settings, output consistency, traceability, and assumption transparency are part of correctness, not auxiliary features.

Taken together, the literature suggests that the primary open problem is often integration quality rather than the absence of strong single-language methods. Teams need coherent command surfaces, comparable artifacts, and diagnostics that separate analyzer/toolchain failures from target-program findings.

This integration challenge appears in three practical forms. First, semantic alignment means that different tools can use the same words (e.g., “escape”, “retained”, “leak”) with different thresholds and assumptions. Second, artifact alignment concerns outputs that may vary from compiler diagnostics to JSON traces to ad hoc logs, making comparison expensive. Third, workflow alignment requires one reproducible execution pattern that can be used in local debugging, benchmark runs, and CI.

Related work in cross-language settings indicates that teams often spend significant effort translating findings between tool conventions before they can act on the underlying issue. This translation cost is not usually visible in accuracy metrics, but it directly affects adoption and remediation speed. For that reason, integration quality is best treated as a first-class research and engineering objective, not as documentation polish after detection logic is done.

High-assurance contexts sharpen this argument further. When resource or timing bounds are part of correctness obligations, ambiguity in finding semantics can itself be a reliability risk. In that environment, normalized evidence structures and explicit assumptions are part of the analysis contract.

These foundations point directly to Graphene’s design space. The prior literature establishes that escape reasoning can be precise and useful, but it also shows that results are hardest to use when outputs, assumptions, and execution contexts do not align across tools. Graphene responds to that gap by integrating these established techniques into one cross-language workflow with consistent reporting and explicit failure semantics.

Table 4: Integration challenges in multi-language escape workflows

Challenge	Typical effect in practice	Integration requirement
Inconsistent output schemas	Hard cross-language comparison	Common result protocol and artifact model
Mixed confidence semantics	Reviewer uncertainty and drift	Explicit confidence and assumption labels
Toolchain failure ambiguity	False attribution to target code	Normalized diagnostics and startup checks
Ecosystem-specific expectations	Uneven adoption across teams	Stable command model with language-native detail

2.4 Positioning of Graphene

Graphene is positioned as an integration-first response to this gap. It does not attempt to replace specialized compiler frameworks or runtime validators. Instead, it composes established ideas into a bounded object-escape workflow that supports cross-language execution and comparable reporting.

This positioning is feasible because prior work already provides the core building blocks, including interprocedural lifetime models, precision mechanisms, correctness caveats, and validation-loop discipline [1, 2, 10, 3, 11]. Graphene contributes orchestration, normalization, and repeatability around those foundations.

Relative to common analyzer patterns, the differentiating value is operational and includes one session model across Python, Java, JavaScript/Node.js, Go, and Rust, static and dynamic evidence in one pipeline, and artifact or diagnostic conventions that reduce translation overhead during triage.

This positioning should be read as compositional rather than replacement-based. Graphene can coexist with specialized analyzers and runtime tools by acting as a unified investigation layer in which invocation and reporting are standardized while language-native analyzers still do the detailed work inside each bridge. That design choice follows directly from related-work evidence that deep, runtime-specific precision remains valuable and should not be erased by a cross-language abstraction.

A second positioning point is methodological transparency. The related-work landscape includes strong results, but also clear caveats about assumptions, evaluation context, and transferability. Graphene's contribution is therefore not only a technical pipeline, but also a workflow where assumptions can be exposed through mode labels, reproducible artifacts, and consistent summary semantics.

In practical terms, this means Graphene's value is highest when used for comparative triage and repeated validation across language boundaries. It is less about claiming a universal best algorithm and more about reducing the friction between strong language-local methods and cross-language operational needs.

Table 5: Positioning summary for this study

Dimension	Common single-tool pattern	Graphene positioning
Scope	Language-local workflow	One orchestrated cross-language workflow
Evidence model	Static-only or runtime-only	Static, dynamic, and fused modes
Failure semantics	Generic analyzer failure output	Normalized startup and bridge diagnostics
Reproducibility	Ad hoc outputs	Structured per-session artifacts
Practical goal	Local optimization/detection	Comparable multi-language investigation

In summary, related work supports a clear design direction that combines proven escape-analysis ideas with an integration model that keeps assumptions explicit, results comparable, and repeated validation practical.

The central synthesis is that modern object-escape tooling requires both analytical rigor and workflow rigor. Analytical rigor comes from established lifetime models, precision controls, and correctness-aware interpretation. Workflow rigor comes from reproducible execution, normalized artifacts, and clear diagnostic boundaries. The four-part structure in this chapter is intended to show how those requirements connect directly to Graphene's design and evaluation method.

3 Method of Approach

This chapter presents the three-stage implementation architecture used to build and evaluate Graphene-HA, a multi-language object-escape analysis framework for Python, Java, JavaScript/Node.js, Go, and Rust. The approach combines a Python command-line wrapper, a Rust orchestration core, language-specific analyzer bridges, and a report-generation pipeline that writes structured session artifacts.

3.1 Development Environment and Toolchain

The implementation uses a split runtime model that balances usability, performance, and maintainability. The Python layer serves as the primary interface, with graphene_ha/cli.py functioning as the user entry point that handles command parsing and initial validation. This thin Python wrapper delegates orchestration responsibilities to a compiled Rust core, where src/main.rs and src/orchestrator.rs coordinate static and dynamic analysis subsystems.

Language-specific analysis is handled through dedicated analyzer bridges located in the analyzers/ directory, each implementing language-specific inspection and execution logic. This modular approach allows new language support to be added without modifying the core orchestration engine. All session outputs are materialized under artifacts/logs//session___/ and include structured artifacts: README.md for human-readable summaries, results.csv for machine-readable findings, and optional vulnerabilities.md for security-relevant escape detections.

This architecture is designed to keep user interaction simple through a unified `graphene` command while preserving performance-critical coordination and protocol reliability in the Rust core.

3.2 Static Architecture Views

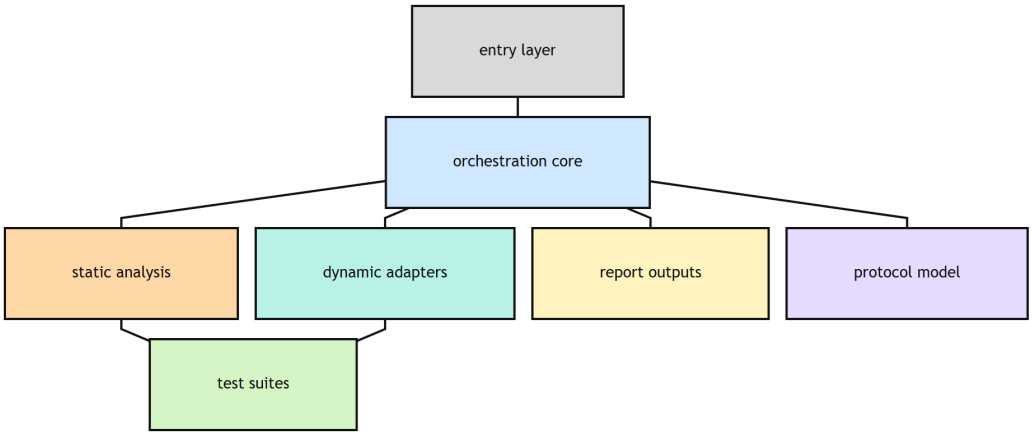

Figure 1. Generalized static communication overview for Graphene-HA.

Figure 1 provides an overview map. To avoid oversimplification, Figures 2 through 8 provide focused static maps for each major subsystem.

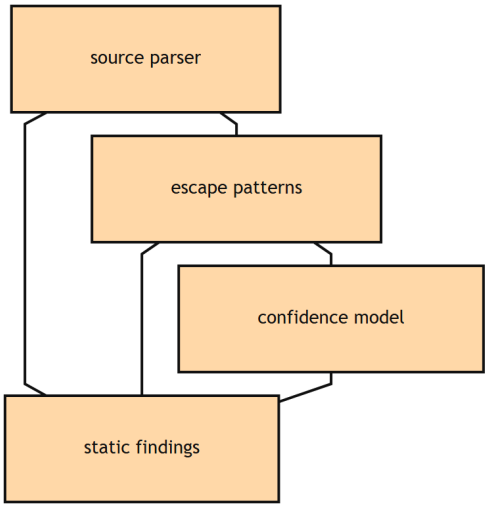

Figure 2. Static analyzer subsystem.

The static analyzer subsystem represents the core machinery for extracting escape information from source code without execution. The process begins with a language-specific source parser that ingests program files and constructs internal representations of code structure. From this representation, the escape patterns module identifies syntactic and semantic features that indicate potential escape behavior, including actions such as assignments to global references, closures that capture mutable state, or heap allocations with escape implications. The confidence model then assesses the likelihood of each detected pattern, distinguishing between certain escapes and probabilistic ones based on analysis precision and language semantics. Finally, all findings are collected and normalized into a unified static findings structure that can be merged with dynamic observations.

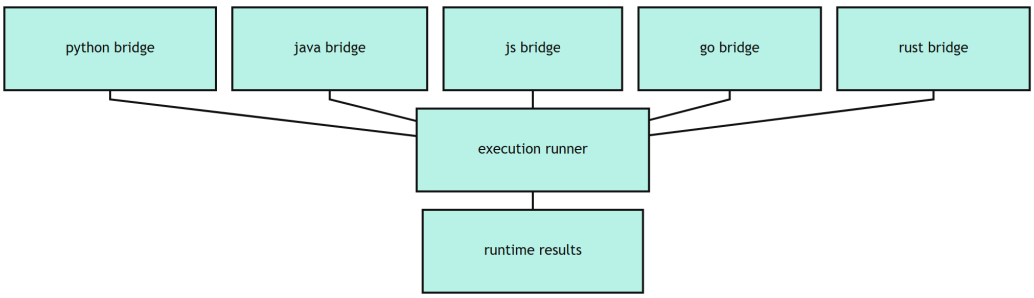

Figure 3. Dynamic adapter subsystem.

The dynamic adapter subsystem manages language-specific execution and runtime observation. Each supported language - Python, Java, JavaScript, Go, and Rust - has a dedicated bridge module that handles the particularities of executing code and capturing object behavior at runtime. These bridge adapters connect to a unified execution runner that coordinates the invocation of target code with supplied or generated inputs. The runner collects runtime observations and produces a standardized results structure that records

behavioral evidence relevant to the escape analysis, such as whether objects propagated beyond expected boundaries during execution.

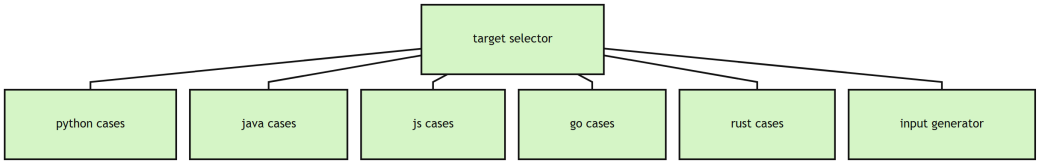

Figure 4. Test suites subsystem.

The test suites subsystem provides the input generation and selection infrastructure used to exercise code paths within target programs. A target selector module allows users or the framework to specify which source files or functions should be analyzed, enabling fine-grained control over scope. Language-specific test case repositories contain manually curated and automatically generated programs that exercise escape behavior across a range of patterns. The input generator module can produce synthetic test cases programmatically, enabling coverage of escape scenarios that might not be readily found in real code.

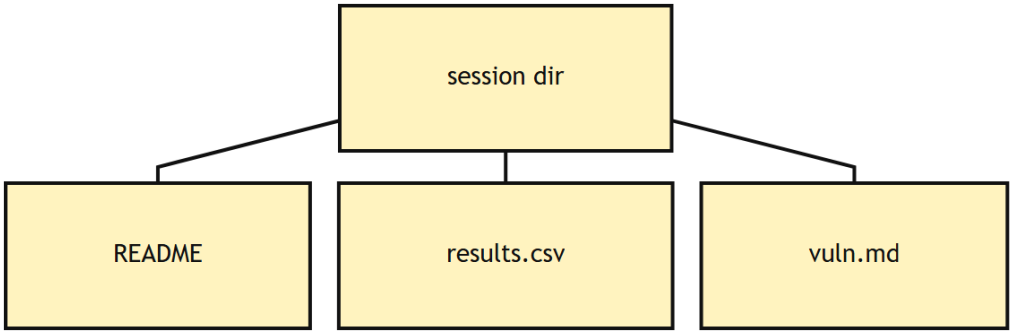

Figure 5. Report outputs subsystem.

Every analysis session produces a timestamped, language-scoped session directory containing reproducible artifacts that enable offline inspection and comparison. Each session folder contains a README documenting input programs, analysis parameters, and high-level findings. It also contains a results.csv file structured for computational analysis and dashboard generation. Optional vulnerabilities.md files highlight escape findings with security implications.

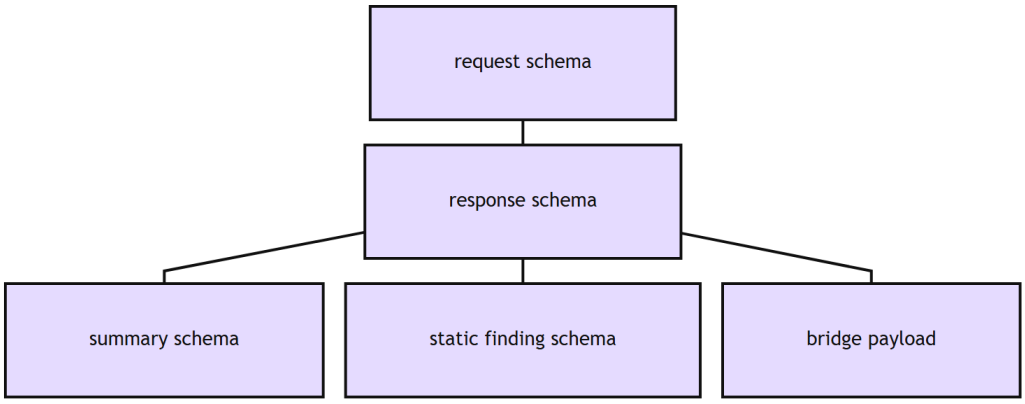

Figure 6. Protocol model subsystem.

The protocol model defines the communication contracts between analysis subsystems. A request schema specifies input parameters: target code, inputs to execute, language filters, and analysis mode selection. The response schema standardizes output, containing a summary schema with aggregate statistics, a static finding schema for code-level observations, and a bridge payload schema for language-specific runtime data.

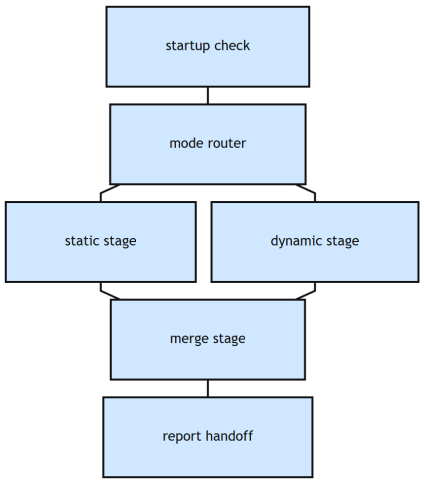

Figure 7. Orchestration core subsystem.

The orchestration core implements the decision tree and sequencing logic that drives an analysis session. At startup, a startup check verifies that required binary dependencies and analyzer modules are present and functional, preventing partial failures mid-execution. A mode router examines the user's selection (static, dynamic, or both) and branches execution accordingly. When both analyses are enabled, the static stage and dynamic stage execute in parallel where possible, reducing total wall-clock time. The merge stage combines results from both paths into a consistent model, and the report handoff passes the unified results to the output generation pipeline.

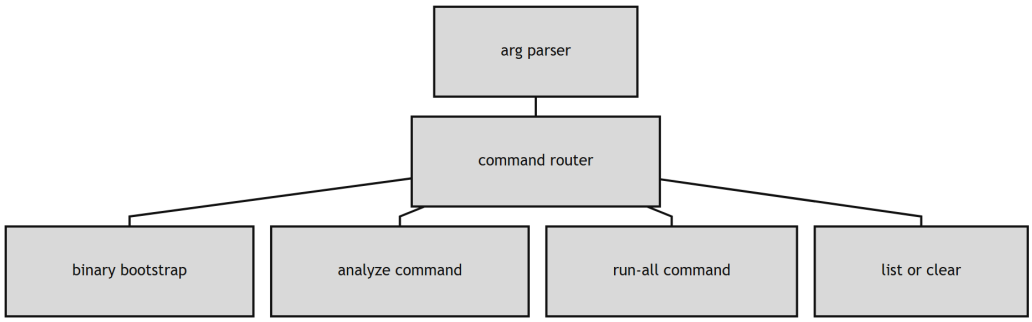

Figure 8. Entry layer subsystem.

The entry layer handles all user-facing command parsing and routing. The argument parser processes command-line flags and positional arguments, validating them against a specification. The command router identifies the requested action - analyze, run-all, list sessions, or clear logs - and branches to the appropriate handler. Binary bootstrap ensures the compiled Rust components are built or cached appropriately. The analyze command handler processes single-program analysis requests, run-all orchestrates batch analysis of multiple programs, and list/clear commands provide session management utilities.

Figures 1 through 8 collectively establish the architectural foundation on which Graphene-HA operates. The static subsystem maps isolate key concerns - parsing, pattern detection, confidence modeling, and test generation - that can be verified, extended, and evolved in isolation. This decomposition reflects three fundamental design principles that guided the implementation.

The Python layer (graphene_ha/cli.py) handles argument validation and delegation, while the Rust core (src/main.rs, src/orchestrator.rs) handles orchestration and protocol-critical behavior. Session artifacts are materialized by src/report.rs, enabling reproducible downstream analysis.

3.3 Runtime Workflow Views

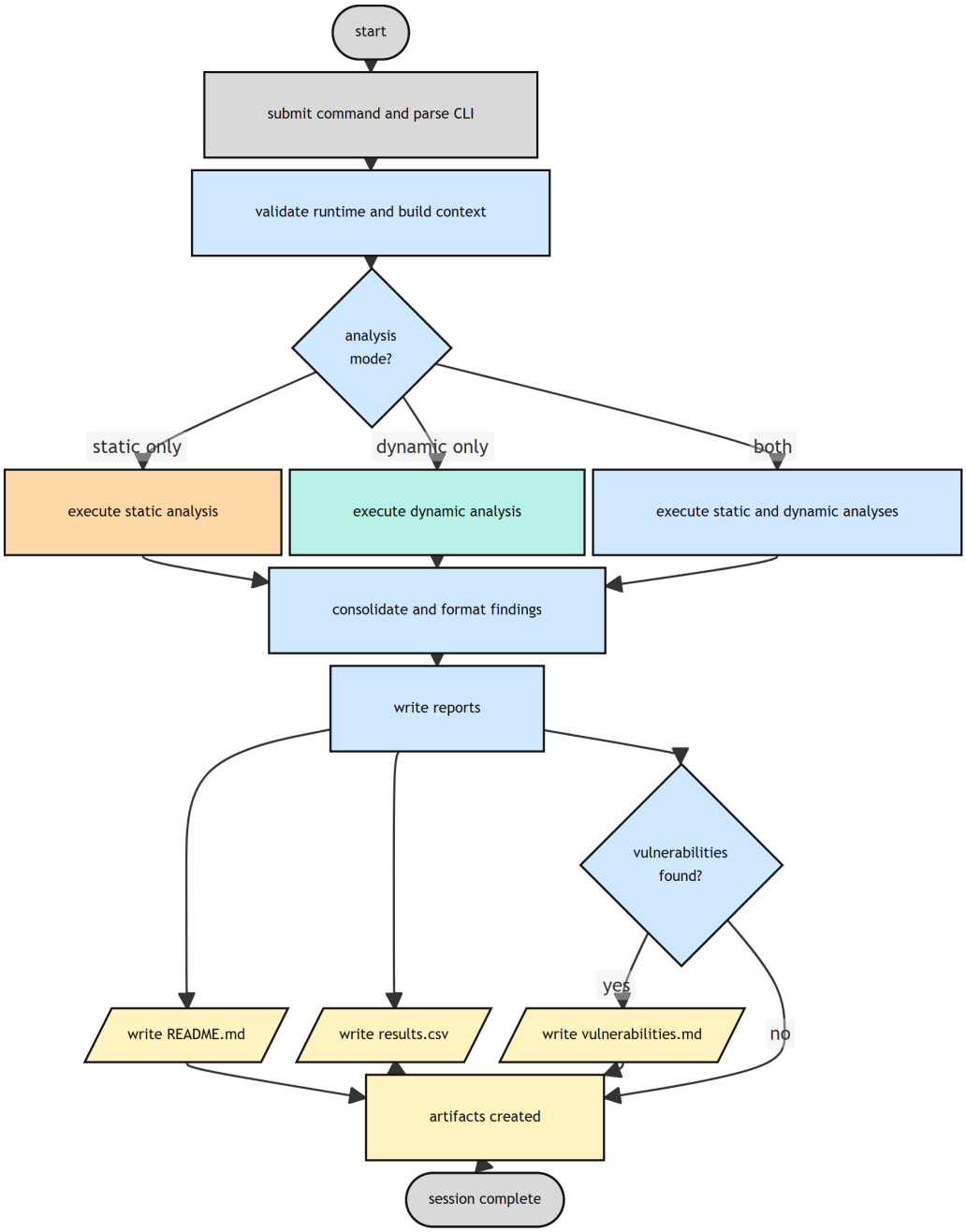

Figure 9. Dynamic user workflow for Graphene-HA analysis sessions.

Figure 9 presents the runtime lifecycle of a Graphene-HA session from command submission to artifact creation. After startup validation, execution branches by analysis mode (static, dynamic, or fused/both), then converges for consolidation and report generation.

The startup phase enforces reliability before analysis begins. The Python entrypoint verifies that required Rust binaries are present and current, and the Rust orchestrator performs dependency self-checks for language bridges and analyzer tools. This preflight step reduces avoidable mid-run failures and improves run-to-run consistency.

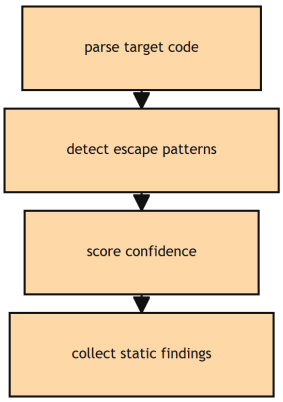

Figure 10. Static stage subchart in the dynamic workflow.

Figure 10 details the static stage. In this phase, analyzers parse source structure, detect escape-relevant patterns, and score confidence for each finding. The output is a normalized static findings set that can be merged with dynamic evidence.

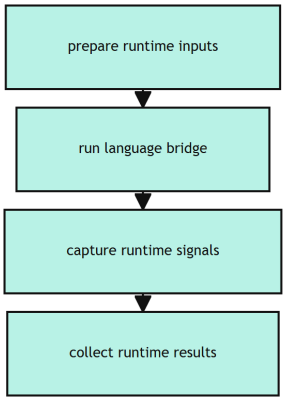

Figure 11. Dynamic stage subchart in the dynamic workflow.

Figure 11 details the dynamic stage. Bridge adapters execute targets with provided or generated inputs, capture runtime escape signals, and package those observations as normalized runtime results for downstream consolidation.

Representative session output (input, runtime behavior, and artifacts) is shown below:

```
$ graphene analyze --lang python --file benchmarks/python/closure_escape.py
↪  --mode both
[entry] parsing arguments...
[core] startup check passed
```

```
[static] pattern detection complete
[dynamic] runtime signal capture complete
[core] merged findings written to session artifacts

# README.md (excerpt)
session_id: session_20260324_214501_a17f
language: python
mode: both
merged_findings: 5

# results.csv (excerpt)
program,language,mode,signal_type,location,confidence,escaped
closure_escape.py,python,merged,confirmed_escape,line 27-34,0.95,true

# vulnerabilities.md (excerpt, when present)
Pattern: object retained beyond request scope
Severity: medium
```

Methodologically, the static subsystem diagrams (Figures 1 through 8) and the dynamic workflow diagrams (Figures 9 through 11) serve complementary purposes. The static maps define architectural boundaries and communication contracts. The dynamic maps define execution order, branching behavior, and merge points. Together, they provide sufficient procedural detail for reproducibility, interpretation of results, and independent validation.

3.4 Dynamic Analyzer Implementation Details: Heap Tracking

The dynamic layer measures heap allocation growth as direct evidence of object escape. Each language bridge executes the target function under a timeout, capturing heap state before and after invocation, then computes the allocation delta. Rather than inferring escape from indirect signals like thread persistence, the bridges directly measure whether unfreed memory accumulates, which is the operational consequence of object escape.

3.4.1 Per-Language Execution and Heap-Tracking Method

In all languages, the orchestrator sends a normalized request over stdin, each bridge resolves the target symbol, executes under timeout protection, and emits protocol-normalized output over stdout.

3.4.2 Dynamic Target Loading and Invocation Model

Across bridges, target resolution follows two implementation families. Python, Java, and Node.js load symbols directly at runtime using each language's native import or reflection model. Go and Rust instead generate temporary runner programs that call the requested symbol, compile those runners, and execute them as subprocesses. This distinction is central to reproducibility: dynamic-language bridges perform in-process symbol loading, while Go and Rust perform out-of-process compiled invocation.

3.4.2.1 Python Target format is module:function or file.py:function. The bridge entry method is analyze(request) in analyzers/python/analyzer_bridge.py. Target resolution is handled by parse_target(…) and load_function_from_target(…) using importlib.import_module(…) or importlib.util.spec_from_file_location(…) for file targets. Execution is dispatched through PythonFunctionTestHarness.run_test(…) in graphene_ha/test_harness.py, configured with prefer_main_thread=True for stable execution semantics.

Concretely, Python loading proceeds in a fixed pipeline.

1. **Target parse and validation**: parse_target(…) enforces module_or_file:function format and rejects missing module/function segments early.
2. **Module resolution strategy**:
   - If the left side ends in .py, the bridge treats it as a file target and uses importlib.util.spec_from_file_location(…).
   - Otherwise, it treats the left side as an importable module path and uses importlib.import_module(…).
3. **Module materialization** (file targets): the bridge creates a module object with module_from_spec(…), registers it in sys.modules, and executes it with spec.loader.exec_module(…) so decorators and module-level references resolve correctly.
4. **Symbol extraction**: the bridge verifies the function exists on the loaded module and resolves it via getattr(…).
5. **Execution binding**: the resulting callable is wrapped by PythonFunctionTestHarness, which standardizes timeout behavior and exception capture before per-input execution.

This in-process import model avoids generated wrappers and preserves native Python call semantics for normal modules and source-file targets.

Heap tracking uses tracemalloc.start(…) and paired snapshots before and after each test execution. The bridge computes positive allocation deltas, maps growth to frame locations, and serializes those observations into protocol fields (escaping_references, escape_paths, and heap metrics in other).

3.4.2.2 Java Target format is ClassName:methodName or classpath:ClassName:methodName (classpath entries split by platform path separator). The bridge parses and loads classes in loadTargetMethod(String target), then resolves invocation targets by reflection. Runtime invocation occurs in executeTest(Method method, String input, double timeoutSeconds) via method.invoke(null, input) on a worker thread, with timeout enforced by Thread.join(timeout_ms).

Java target loading is explicitly two-path and reflection-based.

1. **Right-to-left parse**: loadTargetMethod(…) finds the last and second-last colon to safely parse …:Class:method even when classpath text contains Windows drive letters and path separators.
2. **Classpath mode (classpath:Class:method)**:
   - Split classpath by platform separator.
   - Convert each entry to URL.
   - Construct URLClassLoader.

- Load class through that classloader and locate method by name.

3. **Direct-class mode (Class:method)**:
   - Resolve class through Class.forName(…).
   - Locate method by name from declared methods.
4. **Invocation setup**: set method accessible and execute with method.invoke(null, input) inside a worker thread controlled by join timeout.

The Java bridge therefore treats loading and invocation as explicit reflection steps with consistent timeout enforcement, regardless of whether the class originates from default classpath or a provided classpath payload.

Heap tracking uses java.lang.management.MemoryMXBean snapshots before and after invocation to compute used-heap deltas. Positive deltas are reported as heap escape evidence and merged with other runtime observations.

3.4.2.3 Node.js Target format is module:function or path/to/file.js:function. Entry is analyze(request) in analyzers/nodejs/analyzer_bridge.js. Target resolution is performed by parseTargetReference(…) and loadTargetFunction(…) through Node's require(…) loading path.

Node.js performs in-process CommonJS loading with candidate-path resolution.

1. **Target parse**: parseTargetReference(…) enforces module_or_path:function format.
2. **Target classification**:
   - Path-style targets (`/`, `\\`, or `.js/.mjs/.cjs`) are treated as filesystem module targets.
   - Other targets are treated as module-name imports.
3. **Resolution strategy** (path-style): resolveModuleCandidates(…) builds candidate paths (absolute, current working directory, workspace-root-relative), then loadTargetFunction(…) attempts require(…) in order.
4. **Module strategy** (module-name): use direct require(modulePath) and attempt require.resolve(…) for source-path attribution.
5. **Callable validation**: verify the exported member exists and is a function before execution.
6. **Invocation model**: run through Promise.race(…) between targetFunc(input) and timeout promise, which normalizes synchronous and asynchronous functions under one timeout contract.

Execution is performed in executeTest(targetFunc, targetLabel, input, timeoutSeconds) with Promise.race(…) between function execution and a timeout promise, supporting both synchronous and async targets. Heap tracking uses process.memoryUsage() snapshots, while async-retention context is tracked by AsyncResourceTracker (start, captureBaseline, getEscapedResources, stop) to capture leaked async resources alongside heap growth.

3.4.2.4 Go Target format is `path/to/file.go:ExportedFunction`. The bridge parses targets in `parseTarget(...)`, then builds a temporary executable runner in `buildTargetRunner(sourcePath, functionName)` by rewriting source package declarations and generating an entrypoint that calls the selected function.

Go uses compile-time runner generation rather than runtime symbol import.

1. **Target parse**: `parseTarget(...)` enforces `file.go:ExportedFunction`, validates file existence, and enforces exported identifier rules.
2. **Runner construction** (`buildTargetRunner(...)`):
    - Read target source.
    - Rewrite package declaration to `package main`.
    - Emit generated entrypoint file (`graphene_entrypoint.go`) that calls the selected function.
    - Build temporary binary with `go build` from rewritten source + generated entrypoint.
3. **Execution binding**: return a closure that invokes the compiled binary via `invokeCompiledTarget(...)`.
4. **Input/timeout protocol**: pass input via `GRAPHENE_INPUT`; enforce timeout via `context.WithTimeout(...)`.

This design makes Go invocation deterministic and toolchain-native while avoiding fragile attempts at runtime symbol loading.

Compiled invocation runs through `invokeCompiledTarget(binaryPath, timeout, input)` under `context.WithTimeout(...)`, with test input provided through the `GRAPHENE_INPUT` environment variable. Dynamic runtime behavior is wrapped in `executeTest(...)`.

Heap tracking is based on runtime.MemStats snapshots with explicit runtime.GC() calls to stabilize measurements. Heap allocation and system-heap deltas are converted into normalized escape evidence; goroutine growth remains an additional orthogonal runtime signal.

3.4.2.5 Rust Target format is escape_tests_rust::module::function. The bridge parses this target and builds a temporary runner crate in build_target_runner(target), writes a generated main.rs that calls the selected function, compiles it with cargo build –release, and executes it via create_executor(…).

Rust likewise uses generated-runner execution to align with static linking constraints.

1. **Target parse**: parse_rust_target(…) enforces crate::module::function and validates non-empty crate/module/function components.
2. **Workspace and crate validation**: build_target_runner(…) verifies supported crate expectations (for this benchmark set, escape_tests_rust) and locates the Rust test crate.
3. **Temporary runner crate generation**:
    - Create temporary Cargo project.
    - Write Cargo.toml with dependency on target crate path.
    - src/main.rs invokes escape_tests_rust::::(input).
4. **Runner compilation**: build with cargo build –release and verify the binary exists.
5. **Execution binding**: create_executor(…) wraps subprocess execution of that binary and returns normalized success/error output.
6. **Input transport**: pass each test input through GRAPHENE_INPUT environment variable.

The Rust bridge therefore makes invocation explicit through generated code and subprocess execution, which is robust for static compilation workflows.

Per-test execution is handled by execute_test(…) with channel-based timeout supervision. Input is passed through GRAPHENE_INPUT. Heap tracking is implemented through a custom GlobalAlloc wrapper that counts allocation and deallocation bytes with atomics, enabling direct net-allocation measurement without relying on garbage-collector timing.

All bridges normalize heap-growth data into the unified protocol format: "escaping_references" containing detected objects, "escape_paths" indicating heap-based escape type, and "other" fields quantifying heap growth in bytes. This unified output enables cross-language reporting and fused evidence integration.

3.5 Static-Dynamic Fusion Algorithm

Algorithm 1 summarizes the core runtime procedure used by Graphene-HA with heap tracking.

```
Algorithm 1: Fused object-escape analysis with heap tracking
Input: target program p, language l, analysis mode m in {static, dynamic,
↪  both}
Output: session artifacts A = {README.md, results.csv, optional
↪  vulnerabilities.md}

1. Parse CLI arguments and validate configuration.
2. Execute startup checks for required analyzers/bridges.
3. Build normalized request context r for (p, l, m).
4. If m includes static:
    a. Parse source and detect escape-relevant patterns.
    b. Score static confidence per finding.
5. If m includes dynamic:
    a. Capture baseline heap/allocation state via language-specific
↪  introspection.
    b. Execute target function under timeout.
    c. Capture post-invocation heap/allocation state.
    d. Compute heap growth delta (bytes allocated - bytes freed).
    e. Report non-zero heap deltas as object-escape evidence.
6. Consolidate static and dynamic evidence into unified findings:
    - Static findings are merged with dynamic heap signals.
    - The fused result reports a target as escaping if either static or
↪  dynamic detects escape.
    - Confidence scores are fused using (1 - (1 - c_s)(1 - c_d)).
7. Materialize artifacts in timestamped session directory.
8. Return summary status and output paths.
```

Key characteristics of this approach:

- **Direct measurement**: Heap tracking measures actual allocation outcomes rather than inferring retention through indirect signals.
- **Language-native introspection**: Each bridge uses the most reliable runtime API available (tracemalloc, MemoryMXBean, process.memoryUsage, runtime.MemStats, or custom allocator tracking).

- **Unified protocol**: All bridges normalize heap-growth evidence into the same ExecutionResult structure with escaping_references (detected objects), escape_paths (type = "heap"), and other fields (heap_growth_bytes).
- **Evidence fusion**: The presence of heap growth in dynamic mode increases confidence that static escape patterns represent true runtime retention.

3.6 Implementation Stack and Reproducibility Context

The implementation combines a lightweight Python entry layer with a strongly typed Rust orchestration core and language-native analyzer bridges.

- **CLI and entrypoint**: Python (graphene_ha/cli.py) for argument parsing, validation, and dispatch.
- **Core orchestration**: Rust (src/main.rs, src/orchestrator.rs, src/analyzer.rs) for startup checks, mode routing, bridge execution, evidence merge, and artifact handoff.
- **Static analysis path**: Rust core static analyzer plus language-specific static analyzers in analyzers/python/static_analyzer.py, analyzers/nodejs/static_analyzer.js, and analyzers/go/static_analyzer.go.
- **Dynamic execution path**: language bridges in analyzers/ using runtime-native memory introspection.
- **Report generation**: Rust reporting layer in src/report.rs, producing reproducible session artifacts (README.md, results.csv, optional vulnerabilities.md).

Key dependency profile:

- Python package (pyproject.toml): graphene-ha CLI with standard-library runtime support for bridge interaction.
- Rust package (Cargo.toml): tokio, serde, serde_json, clap, anyhow, chrono, uuid, tracing, tracing-subscriber, async-trait.
- Java bridge: JDK management APIs (MemoryMXBean) and Gson 2.10.1.
- Node.js bridge: standard runtime APIs (process.memoryUsage, async_hooks) without external package-heavy dependencies.
- Go bridge: standard runtime memory APIs with go.mod dependency management.
- Rust bridge: custom allocator instrumentation (GlobalAlloc) and generated runner builds.

Reproducibility is supported in both local and CI-backed execution contexts:

- Local execution: session outputs in artifacts/logs//session___/.
- CI execution: automated publication and artifact workflows in GitHub Actions.

Primary repositories used for implementation and report reproducibility:

- Tool implementation: `https://github.com/ganderson03/Graphene-HA.git`
- Research report: `https://github.com/Allegheny-Computer-Science-580-S2026/junior-seminar-project-journal-and-research-report-chapters-ganderson03.git`

3.7 Data Sets and Benchmark Basis

The analysis corpus is composed of language-scoped benchmark programs and targeted escape-pattern cases. Rather than relying on a single monolithic dataset, the study uses complementary inputs that stress different escape mechanisms.

- benchmarks/ programs in Graphene-HA: multi-language baseline programs for comparability across analyzers.
- Pattern-focused cases in analyzer test suites: curated static/dynamic edge cases for validating known escape motifs.
- Generated runtime inputs: execution-time path expansion for broader behavioral coverage.

3.8 Theoretical Foundation and Scoring Equations

To make integration behavior explicit, the fused stage can be viewed as confidence fusion over static and dynamic signals. Let c_s be static confidence and c_d be dynamic confidence in $[0, 1]$. A conservative combined score is:

$$c_m = 1 - (1 - c_s)(1 - c_d)$$

where c_m is the merged confidence used for unified reporting. For evaluation-oriented summaries, standard metrics remain applicable:

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}, \quad F_1 = \frac{2PR}{P + R}$$

These equations are not presented as universal claims of model optimality. They provide a transparent formal basis for interpreting merged evidence and reporting quality.

3.9 Summary: Integration and Implementation

The complete Graphene-HA framework integrates static analysis, dynamic heap tracking, and fused evidence into a single reproducible workflow. Static analysis provides broad escape-path identification with language-native precision. Dynamic analysis provides execution-grounded confirmation with direct heap-measurement signals. Fusion combines both with explicit confidence modeling to maximize recall while managing static false-positive pressure. This three-stage architecture, supported by protocol-normalized subsystems and session-based artifact management, enables cross-language escape investigation with consistent semantics and repeatable results.

4 Experimental Results

This chapter reports the main experiment for Graphene-HA on 1,421 labeled targets across five languages. It evaluates Graphene's static, dynamic, and fused execution modes on the same labeled benchmark suite.

4.1 Experimental Design

4.1.1 Research Question and Hypothesis

The research question is **Does Graphene's dynamic heap-growth detection method provide a reliable signal for object-escape detection, and does it improve correctness when combined with static analysis?** The hypothesis is that heap-growth detection will achieve high precision and recall because allocation growth is a direct operational consequence of object escape. Unlike indirect retention indicators, heap growth directly measures the unfreed memory that constitutes escape.

4.1.2 Benchmarks and Targets

The labeled suite includes language-specific case files in `tests/<language>/cases/`. Each case is labeled with a `SAFE:` marker when no escape is expected. The absence of this marker indicates an expected escape. The labeled benchmark evaluation uses saved aggregates from `artifacts/logs/` for static, dynamic, and fused runs.

After filtering to cases with resolvable expected labels and available results, the labeled evaluation set contains **1421 targets** across five languages.

- **Go** has 282 targets.
- **JavaScript** has 283 targets.
- **Python** has 286 targets.
- **Java** has 285 targets.
- **Rust** has 285 targets.

4.1.3 Metrics

Each target is assigned a binary label.

- **Expected Escape (positive)** is assigned when the case file does not contain `SAFE:`.
- **Expected Safe (negative)** is assigned when the case file contains `SAFE:`.

For **static**, a target is detected as escaping if the static report lists **Total Escapes > 0**. For **dynamic**, a target is detected as escaping if heap tracking measures **heap growth > 0 bytes** after invocation. For **fused**, a target is detected as escaping if either the static report or dynamic heap tracking detects an escape.

From these binary outcomes, the standard classification metrics are computed.

- **Accuracy** is $(TP + TN)/(TP + TN + FP + FN)$.
- **Precision** is $TP/(TP + FP)$.
- **Recall** is $TP/(TP + FN)$.
- **F1** is $2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$.

These metrics are reported overall and per language. Heap-based detection has the advantage of directly measuring the operational consequence of escape (unfreed memory), rather than inferring retention behavior.

4.1.4 Reproducibility Protocol

This chapter supports two reproducibility paths. The first path is exact replay from archived artifacts used for the reported tables and figures. The second path is full rerun from source using the same Graphene mode configuration and output directories.

For either path, the evaluator should record the exact repository revision before running any command.

```
git rev-parse HEAD
```

4.1.4.1 Exact Replay from Archived Artifacts

Exact replay reproduces the chapter values directly from saved logs and generated assets. The required inputs are stored under `Graphene-HA/artifacts/logs/`.

From the `Graphene-HA` repository root, verify archived mode logs and regenerate mode runs if needed.

```
uv run graphene run-all \
    --generate 10 \
    --analysis-mode static \
    --log-dir artifacts/logs/graphene_static
uv run graphene run-all \
    --generate 10 \
    --analysis-mode dynamic \
    --log-dir artifacts/logs/graphene_dynamic
uv run graphene run-all \
    --generate 10 \
    --analysis-mode both \
    --log-dir artifacts/logs/graphene
```

These commands produce mode-specific session logs required to recompute the reported tables.

4.1.4.2 Full Rerun from Source

Full rerun regenerates the three Graphene mode outputs from source with fixed parameters. From the `Graphene-HA` repository root, run the same three commands shown above in a clean log directory.

```
uv run graphene clear --log-dir artifacts/logs
uv run graphene run-all \
    --generate 10 \
    --analysis-mode static \
    --log-dir artifacts/logs/graphene_static
uv run graphene run-all \
    --generate 10 \
```

```
    --analysis-mode dynamic \
    --log-dir artifacts/logs/graphene_dynamic
uv run graphene run-all \
    --generate 10 \
    --analysis-mode both \
    --log-dir artifacts/logs/graphene
```

This run regenerates session artifacts for static, dynamic, and fused mode analysis only.

4.1.4.3 Reproducibility Checks After either path completes, verify that the following outputs exist.

- `artifacts/logs/graphene`
- `artifacts/logs/graphene_static`
- `artifacts/logs/graphene_dynamic`

Then verify that chapter-level aggregates match the reported values.

- Labeled benchmark size is 1421 targets.
- Overall static accuracy is 61.2%.
- Overall dynamic accuracy is 100.0%.
- Overall fused accuracy is 83.7%.

Minor differences can occur in full reruns if runtime scheduling or environment-specific behavior changes generated paths. When exact numeric agreement is required, use the archived-artifact replay path.

4.1.5 Threats to Validity

The benchmark targets are curated and labeled for controlled comparison, so they do not fully represent production distributions. Single-invocation evaluation cannot capture every multi-request or long-horizon lifecycle issue. Heap-growth delta is a strong operational proxy for retained state, but it is still a proxy rather than a direct proof of every escape case. These limitations shape interpretation of the reported results and support cautious generalization beyond the labeled suite.

4.2 Labeled Benchmark Suite

4.2.1 Overall Correctness Accuracy

Table 1 summarizes the labeled benchmark results for the 1421 targets under Graphene's current mode model.

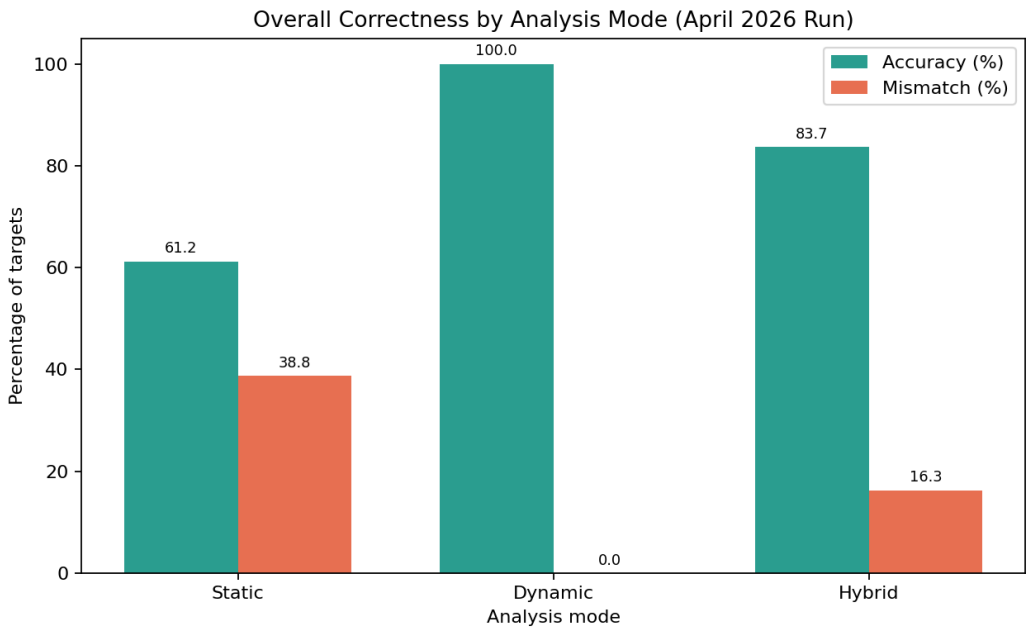

Figure 12: Overall correctness and mismatch by analysis mode.

Table 6: Table 1. Overall correctness metrics by analysis mode.

Approach	Accuracy	Mismatch	Precision	Recall	F1
Static	61.2%	38.8%	68.8%	61.5%	64.9%
Dynamic	100.0%	0.0%	100.0%	100.0%	100.0%
Fused (static + dynamic)	83.7%	16.3%	78.2%	100.0%	87.8%

Relative to static analysis, Graphene's fused mode improves overall accuracy by **22.5 percentage points** and raises recall from **61.5%** to **100.0%**, eliminating false negatives in this dataset. Dynamic mode is perfect on this benchmark slice, while fused mode still reflects static false-positive pressure in languages where static rules over-approximate escapes.

4.2.2 Per-Language Correctness Results

Table 2 breaks down correctness performance by language. Dynamic mode is uniformly perfect in this dataset, while fused behavior depends on how much static false-positive pressure is preserved in each language profile.

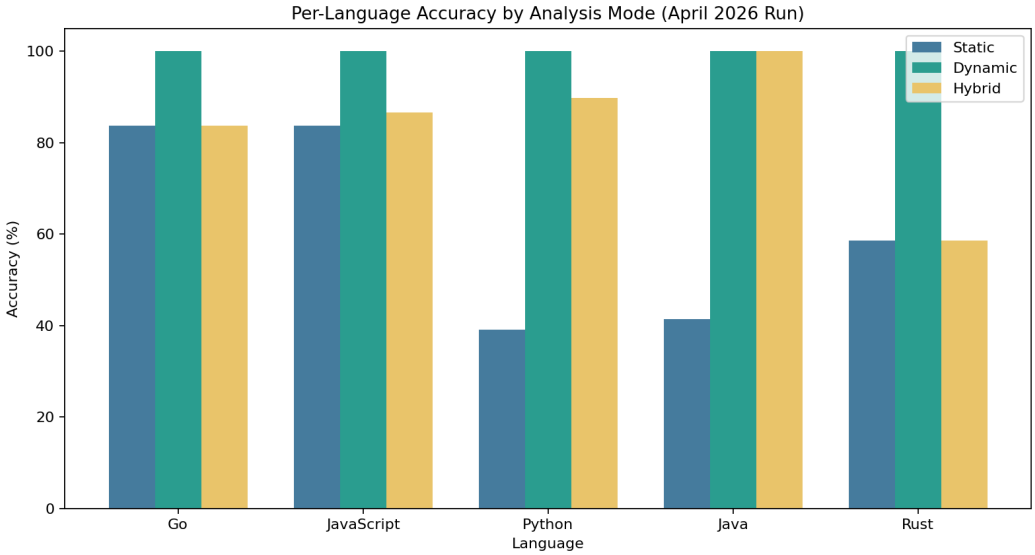

Figure 13: Per-language correctness accuracy by analysis mode.

Table 7: Table 2. Correctness metrics by language and configuration.

Language	Approach	Accuracy	Mismatch	Precision	Recall	F1
Go	Static	83.7%	16.3%	78.1%	100.0%	87.7%
Go	Dynamic	100.0%	0.0%	100.0%	100.0%	100.0%
Go	Fused	83.7%	16.3%	78.1%	100.0%	87.7%
JavaScript	Static	83.7%	16.3%	80.5%	95.2%	87.2%
JavaScript	Dynamic	100.0%	0.0%	100.0%	100.0%	100.0%
JavaScript	Fused	86.6%	13.4%	81.3%	100.0%	89.7%
Python	Static	39.2%	60.8%	43.1%	13.2%	20.2%
Python	Dynamic	100.0%	0.0%	100.0%	100.0%	100.0%
Python	Fused	89.9%	10.1%	85.2%	100.0%	92.0%
Java	Static	42.8%	57.2%	58.0%	21.6%	31.5%
Java	Dynamic	100.0%	0.0%	100.0%	100.0%	100.0%
Java	Fused	100.0%	0.0%	100.0%	100.0%	100.0%
Rust	Static	58.6%	41.4%	58.6%	100.0%	73.9%
Rust	Dynamic	100.0%	0.0%	100.0%	100.0%	100.0%
Rust	Fused	58.6%	41.4%	58.6%	100.0%	73.9%

The per-language results suggest three trends. Static performance remains uneven, especially in Python and Java. Dynamic mode is perfect on this benchmark set. Fused mode raises recall to 100% but does not eliminate static false positives in Go and Rust, so its accuracy remains below dynamic mode in those language slices.

For Java specifically, the latest split-case run shows static improvement to **42.8% accuracy**, **58.0% precision**, and **21.6% recall**, but static remains a weak standalone detector in this benchmark profile.

4.3 Error Analysis and Heap-Signal Reliability

Dynamic heap tracking performs well because it measures direct runtime consequences instead of inferring retention from static structure alone. Each bridge uses runtime-native memory APIs (or allocator instrumentation) to compare before/after allocation state, and positive deltas are reported in a normalized cross-language schema.

In this benchmark set, that directness reduces ambiguity compared with pure static pattern matching, especially in languages where static rules are conservative. The remaining fused-mode errors are largely inherited static false positives in language slices such as Go and Rust.

4.4 Precision-Recall Tradeoff Interpretation

The mode comparison highlights a consistent tradeoff. Static mode is lower-cost but can under-detect or over-flag depending on language profile. Dynamic mode provides direct runtime confirmation but remains benchmark-conditioned. Fused mode maximizes recall, but it can retain static false-positive pressure in conservative static profiles.

Mode choice is therefore policy-dependent. Dynamic or fused mode is appropriate for high-sensitivity triage, while tuned static mode is appropriate for narrow, low-overhead scans where false-positive control is prioritized.

4.5 Interpretation for the Research Question

The results support the claim that direct heap-growth measurement is a reliable dynamic signal for object-escape detection. On the labeled benchmark suite, dynamic heap tracking achieves 100.0% accuracy (F1=1.0), and Graphene's fused mode raises overall accuracy from 61.2% to 83.7% with perfect recall.

Language-level behavior is consistent with that claim. Python and Java show the largest fused gains, while Go and Rust retain static false-positive inheritance in fused mode. This indicates that heap tracking is most impactful where static analysis struggles with aliasing and deferred-lifetime behavior.

4.6 Threats to Validity

Several limitations may affect the interpretation of these results.

1. **Label and benchmark fidelity.** Expected labels rely on `SAFE:` markers in curated cases. Any labeling error or suite bias directly affects measured precision and recall and limits external generalization.

2. **Heap-signal semantics.** Positive heap growth is a strong operational signal, but not every retained allocation is problematic (e.g., intended caches/pools), and not every escape necessarily appears as persistent growth.

3. **Coverage limits in dynamic evaluation.** The 100% dynamic result is benchmark-conditioned. Real deployments can introduce path explosion, environment-specific branching, and input under-sampling that change recall/precision.

4. **Runtime measurement variability and fusion behavior.** GC timing in managed runtimes introduces residual measurement noise, and fused mode can still inherit

conservative static false positives in language profiles where static over-approximation is high.

Overall, the main validity risk is external generalization. The experiment strongly supports the conclusions for these curated suites, but broader claims depend on input diversity, labeling fidelity, heap-semantics assumptions, and behavior on less curated codebases. Dynamic results provide the strongest evidence, while fused results still depend on static-analysis quality in each language context.

5 Conclusions and Future Work

This study addressed a focused research question, whether object-escape behavior can be detected consistently across multiple language ecosystems through a unified workflow that combines static reasoning and runtime evidence. Graphene-HA approached this question through a single command surface, language-native bridges, and a fused evidence model. Empirical findings suggest that dynamic heap-growth measurement demonstrated high reliability on the labeled benchmark set of 1,421 targets, and that fused analysis improved accuracy over static-only analysis, increasing overall correctness from 61.2% to 83.7%. These outcomes establish technical feasibility and clarify both where current confidence is warranted and where additional evidence remains necessary.

5.1 Significance and Impact

The significance of this work lies in transforming cross-language escape triage from an ad-hoc investigation process into a repeatable engineering workflow. In production systems, escape-related defects are difficult to prioritize because evidence is fragmented across tools with incompatible output schemas and inconsistent diagnostics. Graphene-HA contributes an integration layer that standardizes artifacts and failure semantics, reducing translation overhead during triage and improving the reproducibility of follow-up investigations.

Empirical results also reveal practical interpretation boundaries. Dynamic evidence appears to have been the strongest signal within this benchmark setting, while fused analysis remained constrained by static false-positive behavior in certain language profiles. This finding suggests that fusion's immediate value lies in recall protection and cross-signal validation, not automatic precision improvement across all runtimes. Viewed through this lens, Graphene-HA functions as a decision-support system for lifecycle diagnostics rather than as a universal defect detector.

From a research perspective, this project demonstrates that workflow-level rigor qualifies as a first-class scientific contribution. Much prior work emphasizes algorithmic precision within a single language or runtime. This work instead shows that protocol normalization, artifact consistency, and startup-phase diagnostics can materially affect whether technically sound analysis methods remain usable across repeated operational cycles. Impact derives from both detection capability and interpretability quality, each supporting the other.

From practical and educational perspectives, this work lowers barriers to systematic object-lifetime analysis in mixed-language repositories. A stable command interface and consistent report format enable users to compare findings across Python, Java, JavaScript/Node.js, Go, and Rust without relearning tool semantics for each ecosystem. That portability is not purely ergonomic. It enables better longitudinal monitoring because repeated sessions can be directly compared with minimal ambiguity.

5.2 Results Interpretation

The strongest positive finding is that direct heap-growth observation functioned as a reliable runtime signal under the benchmark conditions employed in this study. However, this result does not imply that dynamic analysis should replace static analysis. Static analysis continues to provide low-cost candidate generation and broad initial coverage. Dynamic analysis contributes execution-grounded confirmation. Fusion contributes robustness against missed cases, but only when static and dynamic signals are interpreted under explicit policy.

The results motivate a policy-oriented view of mode selection that depends on operational context. Static mode is appropriate for broad screening where runtime cost must remain minimal and false positives can be addressed downstream. Dynamic mode is appropriate for confirmation and high-confidence triage under representative workloads. Fused mode is appropriate when the cost of missed escapes is high and conservative escalation is acceptable.

This framing offers an insight beyond simple mode ranking. The operative question is not "which mode is superior," but rather "which evidence policy best aligns with operational risk in the current context."

5.3 Limitations and Threats to Validity

Five distinct validity concerns constrain the generalizability of these findings.

Construct validity concerns rest on the dynamic signal, which is based on post-invocation heap-growth deltas. Although this represents a strong operational proxy for retained state, not every positive delta indicates harmful retention. Similarly, not every escape manifests as a clean persistent delta across all runtime conditions.

Internal validity concerns arise from bridge-level measurement differences, including garbage collection timing, allocator behavior, and runtime scheduling factors that can influence observed deltas. Although controls were implemented to mitigate these effects, residual runtime noise remains possible.

External validity is limited by evaluation on curated and labeled suites, which enable controlled comparison but do not fully represent production distributions. This limitation is especially acute for long-lived services with multi-stage asynchronous workflows.

Statistical conclusion validity is constrained because the current report emphasizes descriptive metrics and effect sizes without formal confidence intervals, significance testing, or uncertainty quantification across repeated randomized trials.

Ecological validity is limited because experiments focus on bounded invocation windows, whereas many real-world incidents involve multi-request interactions, queue backlogs, and temporal retention spanning minutes or hours.

These threats do not invalidate the findings. Rather, they circumscribe the scope of defensible generalizations. The current evidence supports the system's utility in controlled laboratory settings and motivates a more rigorous empirical program for production-representative contexts.

5.4 Future Work with Technical Depth

Future work is organized around four technically focused tracks designed to strengthen both detection accuracy and evidence quality.

5.4.1 Retention Lineage and Causal Attribution

Current dynamic analysis identifies that retention occurred but does not fully explain causality. The next phase will introduce lineage-aware attribution to address this gap.

Planned technical changes include the following steps:

1. Add allocation-site identifiers and reference-edge snapshots at key boundaries (pre-invocation, post-invocation, post-garbage-collection).
2. Construct a compact retention graph per finding that maps surviving objects to owning structures (for example, callback registries, queue payloads, cache entries).

3. Extend session artifacts with causal summaries that distinguish “retained by design” from “retained unexpectedly” using explicit rule labels.

This approach is expected to reduce triage latency and improve remediation precision.

5.4.2 Static Precision Improvements for Weak Profiles

Static analysis performance varied substantially across language profiles, particularly where aliasing and deferred execution dominate code structure.

Planned technical changes include the following steps:

1. Introduce interprocedural escape summaries for common escape carriers (queues, closures, executor submissions, channels).
2. Add path-sensitive guards for branch-local ownership transfers to reduce over-approximation.
3. Integrate language-specific heuristics for framework idioms (for example, task schedulers and callback registries) while preserving normalized output schemas.

This approach is expected to reduce false-positive pressure in fused analysis and improve static-only triage quality.

5.4.3 Adaptive Dynamic Input and Path Exploration

Current dynamic execution relies on bounded, largely fixed execution strategies. Future work will make dynamic probing adaptive while maintaining cost controls.

Planned technical changes include the following steps:

1. Add coverage-guided input mutation for target functions exhibiting high static uncertainty.
2. Add budget-aware path exploration that prioritizes branches associated with suspected retention patterns.
3. Add stability checks across repeated runs to distinguish deterministic retention from input-dependent noise.

This approach is expected to strengthen recall on complex control-flow targets without incurring unbounded execution overhead.

5.4.4 Multi-Context and Long-Horizon Retention Analysis

Single-invocation experiments do not fully represent lifecycle defects that emerge in queue-driven and distributed systems.

Planned technical changes include the following steps:

1. Add session chaining to model retention across request sequences.
2. Add queue-aware probes that track object survival across enqueue/dequeue boundaries.
3. Add optional distributed-trace correlation for services where ownership transfers cross process boundaries.

This approach is expected to improve relevance for production incident classes that emerge over extended runtime horizons.

5.4.5 Language Ecosystem Expansion

Currently, Graphene-HA supports five languages (Python, Java, JavaScript/Node.js, Go, and Rust). Future work will extend support to additional runtime environments where object-escape analysis delivers practical value.

Planned technical changes include the following steps:

1. Add bridge implementations for C# and the .NET runtime, enabling analysis in the growing ecosystem of managed web services and cloud applications.
2. Add support for TypeScript and WebAssembly, capturing the emerging class of systems that bridge compile-time and runtime boundaries.
3. Add preliminary support for Kotlin and Scala, leveraging the Java Virtual Machine infrastructure already built.
4. Establish a bridge-design template and validation harness that reduces engineering cost for adding new language support while maintaining artifact schema consistency.

This approach is expected to increase the operational relevance of Graphene-HA by covering a broader segment of modern software stacks.

5.4.6 Error Type Diversification

Current Graphene-HA focuses exclusively on object escapes, one category of lifetime-management defect. Future work will extend detection to complementary error classes that affect correctness, stability, and resource efficiency.

Planned technical changes include the following steps:

1. Add detection for premature deallocation, where objects are released before their last use site, causing use-after-free or null-pointer errors.
2. Add detection for circular references and reference cycles that prevent garbage collection, leading to memory leaks that differ structurally from escapes.
3. Add detection for resource leaks beyond memory (file handles, network connections, locks), extending the framework to address broader lifecycle integrity.
4. Add detection for temporal atomicity violations, where sequences of allocations and deallocations across multiple operations expose race conditions.
5. Establish a unified error taxonomy that maps each error class to its corresponding analysis requirement (static, dynamic, or hybrid) to guide framework design decisions.

This approach is expected to make Graphene-HA a comprehensive lifecycle-diagnostics platform rather than a single-issue detector.

5.5 Empirical Evaluation Plan for Future Work

Each future-work track will be evaluated through explicit hypotheses, standardized datasets, and predefined acceptance criteria.

5.5.1 Evaluation Design

1. Datasets
    - Existing labeled benchmark suites to maintain continuity with current results.

- New stress suites for asynchronous pipelines, queue-heavy workflows, and framework-mediated ownership transfer.
- At least two production-derived anonymized corpora to strengthen external validity.

2. Baselines
   - Current Graphene-HA release (static, dynamic, and fused modes).
   - Selected specialized analyzers already integrated in the comparison workflow.
3. Metrics
   - Detection metrics (accuracy, precision, recall, F1).
   - Calibration metrics (confidence reliability and error concentration by pattern class).
   - Operational metrics (median triage time, false-alarm burden, runtime overhead, and artifact interpretability measured through blinded reviewer studies).

5.5.2 Hypothesis-Driven Milestones

1. Retention-lineage attribution should reduce median investigation time by at least 30% in controlled debugging tasks.
2. Static precision upgrades should reduce fused false positives by at least 20% in Python and Java benchmark subsets without decreasing fused recall below 99%.
3. Adaptive dynamic exploration should improve dynamic recall on long-tail asynchronous cases by at least 5 percentage points at no more than 1.5× median runtime cost.
4. Multi-context analysis should identify retention scenarios that single-invocation analysis misses, demonstrating measurable incremental recall on staged workflows.
5. Language ecosystem expansion should achieve at least 70% detection accuracy on C# and .NET targets within 12 months of bridge implementation, validated on curated benchmarks equivalent to the existing five-language suite.
6. Error type diversification should establish detection baselines for premature deallocation, circular references, and resource leaks within the existing five-language ecosystem before expanding to new languages.

5.5.3 Experimental Method and Reporting

1. Use preregistered evaluation protocols and fixed random seeds for reproducibility where applicable.
2. Report confidence intervals for primary metrics and perform statistical significance testing for key pairwise comparisons.
3. Publish ablation studies for each major component to isolate contribution magnitude.
4. Release artifact bundles (inputs, outputs, configurations, and version metadata) to enable full reproducibility.

This evaluation plan is designed to convert future work from conceptual aspiration into testable claims with explicit stopping criteria.

5.6 Operational Significance

For engineering teams that maintain polyglot codebases, this work means escape related diagnostics can be prioritized within one operational loop instead of being reconstructed across separate tools for each language. That reduction in translation overhead shortens the path from detection to remediation and makes repeated validation more practical during active maintenance. The broader contribution is therefore not only better signal quality, but also a more usable workflow for day to day investigation.

5.7 Ethical Implications Revisited in Light of Results

The empirical findings sharpen several critical ethical concerns relevant to deployment.

5.7.1 Over-Trust and Confidence Calibration

Because dynamic signals appear robust in controlled conditions, a risk of over-reliance emerges when deployed in noisier production environments. Over-reliance can foster false assurance and delay complementary validation activities. Therefore, reports must present confidence boundaries and scope limits explicitly, including mode assumptions and known blind spots.

5.7.2 Labor Burden and Governance

False positives in static or fused analysis can impose nontrivial labor costs, particularly for resource-constrained teams with limited review capacity. Ethical deployment requires cost-aware governance mechanisms, including escalation thresholds, reviewer rotation, and explicit criteria for deferring low-confidence findings.

5.7.3 Data Sensitivity and Artifact Handling

Richer runtime artifacts improve diagnosability but increase data-protection risk. As lineage tracking is added in future work, artifact handling policy must enforce data minimization, role-based access controls, retention windows, and redaction defaults for potentially sensitive identifiers.

5.7.4 Equity and Accessibility

Fairness of tooling impact matters significantly. If analysis outputs are only interpretable by specialized experts, resource-constrained teams face disadvantage. Future interface and reporting design must therefore include accessibility validation with reviewers of mixed expertise levels.

5.7.5 Synthesis

Ethical use fundamentally depends on three factors: calibrated claims about what the system can and cannot detect, appropriately bounded automation that preserves human oversight, and governance structures that account for both safety benefits and operational burdens.

5.8 Closing Perspective

This work establishes that cross-language object-escape investigation can be made both technically rigorous and operationally sustainable through a fused-evidence workflow. The central contribution is not a universal detection guarantee, but rather a practical framework that makes lifecycle-risk evidence easier to generate, compare, and apply across heterogeneous codebases.

The empirical results provide a strong foundation for operational use, while the documented validity threats and structured future evaluation plan define clear paths toward stronger generalization. Should the next phase of development succeed, Graphene-HA will evolve from a promising benchmark-validated prototype into a production-grade, evidence-calibrated system for real-world lifecycle diagnostics. That trajectory captures the essence of this project by unifying algorithmic analysis, reproducible workflow design, and responsible engineering practice within a single coherent research and engineering program.

References

[1] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Association for Computing Machinery, New York, NY, USA, 1–19. https://doi.org/10.1145/320384.320386

[2] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. 2003. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Program. Lang. Syst.* 25, 6 (2003), 876–910. https://doi.org/10.1145/945885.945892

[3] Boyao Ding, Qingwei Li, Yu Zhang, Fugen Tang, and Jinbao Chen. 2024. MEA2: A Lightweight Field-Sensitive Escape Analysis with Points-to Calculation for Golang. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 1362–1389. https://doi.org/10.1145/3689759

[4] Ralf Jung, Benjamin Kimock, Christian Poveda, Eduardo Sánchez Muñoz, Oli Scherer, and Qian Wang. 2026. Miri: Practical Undefined Behavior Detection for Rust. *Proc. ACM Program. Lang.* 10, POPL (2026), 1383–1411. https://doi.org/10.1145/3776690

[5] Clemens Lang and Isabella Stilkerich. 2020. Design and Implementation of an Escape Analysis in the Context of Safety-Critical Embedded Systems. *ACM Trans. Embed. Comput. Syst.* 19, 1 (2020), 1–20. https://doi.org/10.1145/3372133

[6] Maurizio Martignano. 2022. Static Analysis for Ada, C/C++ and Python: Different Languages, Different Needs. *Ada Lett.* 41, 2 (2022), 77–80. https://doi.org/10.1145/3530801.3530807

[7] Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund. 2017. Correctness of Partial Escape Analysis for Multithreading Optimization. In *Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3103111.3104039

[8] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*. Association for Computing Machinery, New York, NY, USA, 62–71. https://doi.org/10.1145/1791194.1791203

[9] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. Association for Computing Machinery, New York, NY, USA, 165–174. https://doi.org/10.1145/2581122.2544157

[10] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2018. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. Association for Computing Machinery, New York, NY, USA, 165–174. https://doi.org/10.1145/2544137.2544157

[11] Cong Wang, Mingrui Zhang, Yu Jiang, Huafeng Zhang, Zhenchang Xing, and Ming Gu. 2020. Escape from escape analysis of Golang. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. Association for Computing Machinery, New York, NY, USA, 142–151. https://doi.org/10.1145/3377813.3381368

[12] Matthew Edwin Weingarten, Theodoros Theodoridis, and Aleksandar Prokopec. 2022. Inlining-Benefit Prediction with Interprocedural Partial Escape Analysis. In *Proceedings of the 14th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3563838.3567677