Rusting Volatility: Accelerating Memory Forensics through LLM-Assisted Cross-Language Migration

Taha Gharaibeh[a,b], Ibrahim Baggili[a,b]

[a] *Baggili(i) Truth (BiT) Lab, Center of Computation & Technology, Baton Rouge, LA, USA*
[b] *Division of Computer Science & Engineering, Louisiana State University, Baton Rouge, LA, USA*

Abstract

Memory forensics remains a core capability in modern Digital Forensics and Incident Response (DFIR), but practical workflows are often constrained by the runtime costs of Python-based tooling. We report an empirical migration of a production memory-forensics framework, Volatility3, from Python to Rust using Large Language Model (LLM)-assisted development. The migration produced 38 Linux forensic plugins totaling 29,028 Lines of Code (LOC) over 68 hours of active implementation, with aggregate LLM usage of 900 million tokens ($2,962) across two systems. To evaluate functional behavior, we applied canonical output hashing over 11 memory samples spanning two Linux kernel versions. Rust completed all 417 analyzed plugin-sample executions, whereas Python failed on 47 (11.3%) because of runtime and symbol-compatibility errors. Exact output equivalence reached 68.9% among comparable executions. Under controlled benchmark conditions, the Rust implementation achieved a 2.50× median speedup, with structure-traversal plugins reaching up to 210×. Overall, the results indicate that compiled-language reimplementation can improve both execution robustness and throughput for analysis-intensive forensic tasks, while LLM-assisted migration can reduce development effort when paired with strict parity validation.

Keywords: Memory forensics, Volatility, Rust, Python, Language migration, Large language models, Digital forensics

1. Introduction

Memory forensics, the analysis of volatile memory to recover digital evidence, is indispensable to modern DFIR practice (Case and Richard III, 2017; Schatz and Cohen, 2017). In investigations involving advanced persistence, fileless malware, and post-exploitation activity, many critical artifacts are available only in memory: injected regions, live network state, cryptographic material, and process-hollowing traces that may never appear on disk (Block and Dewald, 2019; Sudhakar and Kumar, 2020). The Volatility ecosystem, which originated in academic research and matured through community development, has therefore become the primary open-source toolchain for cross-platform memory analysis (Nyholm et al., 2022).

At the same time, Python imposes practical performance limits for this workload class (Zhang et al., 2022a; Barany, 2014). The Global Interpreter Lock (GIL) constrains in-process parallelism (Meier and Gross, 2019), while dynamic dispatch and runtime typing overhead accumulate in plugins that repeatedly traverse pointer-rich kernel structures. Attribute lookup through dictionary and descriptor resolution is inexpensive at the operation level, but forensic plugins execute these paths millions of times. As image sizes continue to increase (8–16 GB is common, and larger captures are routine in enterprise response), this overhead translates into analyst-visible delays and slower triage cycles.

Rust offers a plausible architectural alternative. Its compilation model removes interpreter dispatch overhead, its ownership system enforces memory safety without garbage collection, and its concurrency model supports real parallel execution (Matsakis and Klock, 2014; Jung et al., 2017). The central obstacle is migration cost: rewriting a mature Python framework in Rust has historically required substantial engineering effort (Emre et al., 2021; Zhang et al., 2023). Recent LLMs suggest a different tradeoff by accelerating implementation and debugging cycles (Zan et al., 2023; Li et al., 2022). However, whether this assistance remains reliable for large, semantics-sensitive systems migration is still an open empirical question (Pan et al., 2024).

This work studies whether LLM-assisted cross-language migration can produce a high-performance Rust implementation of a production Python forensics stack while preserving functional behavior. We evaluate this question through three hypotheses:

H1 (Performance): Rust reimplementation of memory forensics plugins will achieve at least 2× median speedup over Python due to elimination of interpreter overhead.

H2 (Correctness): The migrated implementation will achieve greater than 60% exact output equivalence

Email addresses: tahatlal@gmail.com (Taha Gharaibeh), baggili@gmail.com (Ibrahim Baggili)

with the reference implementation across diverse memory samples.

H3 (Robustness): The Rust implementation will exhibit equal or superior execution completion rates compared to Python across the test corpus.

To test these hypotheses, we migrated Volatility3's Linux plugin stack to Rust through an LLM-assisted workflow and evaluated parity and performance at plugin-sample granularity. The main contributions are:

- **38 Linux forensic plugins reimplemented in Rust** (29,028 LOC across 8 crates), preserving Volatility3's plugin architecture while eliminating Python interpreter overhead[1].
- **Hash-based parity validation** that classifies each plugin-sample pair as EQUIVALENT, DIVERGENT, or INCOMPARABLE, enabling systematic correctness testing against the Python reference.
- **834 analyzed benchmark runs** (after removing one unstable tuple per cache condition), revealing a 2.50× median speedup, with structure-traversal plugins reaching 210× acceleration.
- **Zero Rust failures** across all benchmark runs, compared to 47 Python failures (11.3%), all attributable to symbol compatibility issues in the reference implementation.

The remainder of the paper is organized as follows. Section 2 reviews Volatility3 architecture and the relevant Python and Rust execution characteristics. Section 3 details the migration process, corpus design, parity procedure, and benchmarking protocol. Section 4 reports empirical outcomes and evaluates the hypotheses. Section 5 interprets implications for practice and tooling design. Section 6 examines validity limitations. Section 7 situates the work within prior literature. Section 8 concludes, and Section 9 outlines future directions.

2. Background

This section establishes the technical background required for the migration study. We review Volatility3's architectural model, summarize Python execution costs that matter for memory forensics, outline the Rust properties relevant to reimplementation, and place the migration task in the context of LLM-assisted development.

[1]Source code available at `https://bit.ly/3MpT806`. For double-blind review, redistribution is not permitted; code will be open-sourced upon acceptance.

2.1. Memory Forensics and Volatility

Memory forensics emerged because disk-centric methods cannot recover many transient execution artifacts (Vidas, 2007; Hay and Nance, 2008). Volatile memory can retain live network state, decrypted payloads, injected code, and process metadata that never appears in persistent storage (Block and Dewald, 2017; Or-Meir et al., 2019). Volatility, first released in 2007 and later redesigned as Volatility3, became the dominant open-source framework for this task (Schuster, 2008).

Volatility3 uses a layered architecture that separates data access, address translation, symbol interpretation, typed object creation, and plugin logic. **Data layers** read memory sources such as raw captures, crash dumps, VMware snapshots, and hibernation files. **Translation layers** map virtual addresses to physical offsets through architecture-specific page-table traversal. **Symbol tables**, loaded from Intermediate Symbol Format (ISF) JavaScript Object Notation (JSON) files generated by tools such as `dwarf2json`, encode layout metadata for specific kernel builds. The **object system** materializes typed structures at runtime addresses, and **plugins** implement analysis tasks (for example process listing, module inspection, and network extraction) by traversing those structures.

This design allows plugin logic to remain largely acquisition-agnostic. A process plugin, for instance, reads `task_struct` fields by symbolic name rather than hard-coding source-specific offsets.

2.2. Python Performance Characteristics

Python's runtime model introduces overhead at multiple levels that directly affect forensic workloads (Zhang et al., 2022a). The GIL serializes bytecode execution inside a process, so practical parallelism often relies on multiprocessing and its associated communication costs (Meier and Gross, 2019; Meier et al., 2018).

Object attribute access also introduces repeated dynamic lookup overhead. Evaluating `obj.field` triggers `__getattribute__`, which may traverse instance dictionaries, class dictionaries, and method resolution paths. Each lookup is inexpensive in isolation but appears at very high frequency in structure-heavy plugins.

Numeric operations carry similar costs: values are boxed objects, reference counts are updated continuously, and runtime type checks remain on critical execution paths. These mechanisms improve flexibility but add cumulative overhead in tight loops.

For a plugin that traverses thousands of structures and repeatedly dereferences nested fields, these micro-costs compound into meaningful wall-clock delay relative to compiled implementations.

2.3. Rust as a Systems Language

Rust offers near-C performance characteristics while enforcing memory safety through ownership and borrowing

rules (Jung et al., 2017, 2019). Struct-field access compiles to direct offset loads, integer arithmetic maps to native Central Processing Unit (CPU) operations, and there is no tracing garbage collector on hot paths (Astrauskas et al., 2019).

Rust additionally supports practical data parallelism. Libraries such as `rayon` provide work-stealing parallel iterators with low adoption friction, and threads execute concurrently without an interpreter lock.

Together, these properties align well with memory forensics workloads: structure traversal reduces to compiled pointer arithmetic with explicit checks, symbol resolution becomes native hash-table work, and scan-heavy tasks can exploit core-level parallelism without interpreter contention (Zhang et al., 2022b; Qin et al., 2020).

2.4. LLM-Assisted Software Development

LLMs trained on large code corpora have demonstrated substantial capability in code synthesis and completion tasks (Li et al., 2022; Xu et al., 2022). Cross-language migration is especially relevant because source code already encodes intended behavior, reducing ambiguity relative to pure natural-language requirements (Ahmad et al., 2023b; Huang et al., 2023).

Despite that structural advantage, migration still involves non-trivial semantic decisions. Python's dynamic typing must map to Rust's static type constraints; garbage-collected lifetime behavior must be recast through ownership; and duck-typed interfaces must become explicit trait or type-level contracts. These transformations are not reliably automated end-to-end, which motivates a human-in-the-loop workflow in which LLMs accelerate implementation while developers retain responsibility for validation.

3. Methodology

This section details the migration and evaluation design, including the LLM-assisted development workflow, corpus construction, parity procedure, and benchmark protocol.

3.1. Migration Process

The migration followed a two-phase workflow in which different LLM systems were used for planning and implementation tasks.

Phase 1: Architectural analysis. We used Claude Opus 4.5 through GitHub Copilot agent mode to analyze the Python codebase and document migration-relevant structure. Outputs included module dependencies, inheritance hierarchies, and identification of Python-specific constructs requiring explicit Rust adaptation (for example dynamic member access through `__getattr__` and multiple inheritance patterns).

Phase 2: Implementation and refinement. We used GPT-5.2-Codex for iterative plugin implementation and Claude Opus 4.5 for structural refinement, debugging, and parity-focused review. GPT-5.2-Codex consumed approximately 400 million tokens (30% input, 10% output, 60% cached context), and Claude contributed an additional 500 million tokens. Table 1 summarizes usage and cost estimates. Development followed a Test-Driven Development (TDD)-style loop: implement a plugin, run against representative samples, compare output with Python, diagnose mismatches, and iterate until equivalence or explainable divergence. We measured active development effort as 68 hours based on file modification timestamps.

Table 1: LLM token consumption and estimated costs.

Model	**Tokens**	**Cost**	**Use**
GPT-5.2-Codex	400M	$812	Implementation
Claude Opus 4.5	500M	$2,150	Planning & Skeleton Building
Total	**900M**	**$2,962**	

The resulting system comprises eight Rust crates organized as a Cargo workspace (Table 2).

Table 2: Rust implementation architecture.

Crate	**Responsibility**	**LOC**
vol3-core	Error types, configuration	1,892
vol3-layers	Memory access, translation	2,847
vol3-symbols	ISF parsing, symbols	3,214
vol3-objects	Typed memory structures	2,156
vol3-framework	Context, module registry	1,891
vol3-renderers	Comma-Separated Values (CSV), JSON output	987
vol3-plugins-linux	38 forensic plugins	19,581
vol3-cli	Command-line interface	1,460
Total		**29,028**

3.2. Test Corpus

We evaluated both implementations on 11 memory samples spanning two Linux kernel versions (Table 3). The corpus was designed to include kernel diversity (5.15 LTS and 6.14 mainline), mixed workload conditions (including ROS-related and general-purpose activity), and varied system states.

Table 3: Memory sample corpus.

Kernel	**Count**	**Description**
5.15.0-139	5	Ubuntu 20.04, ROS1 workload
6.14.0-36	6	Ubuntu 24.04, varied workloads

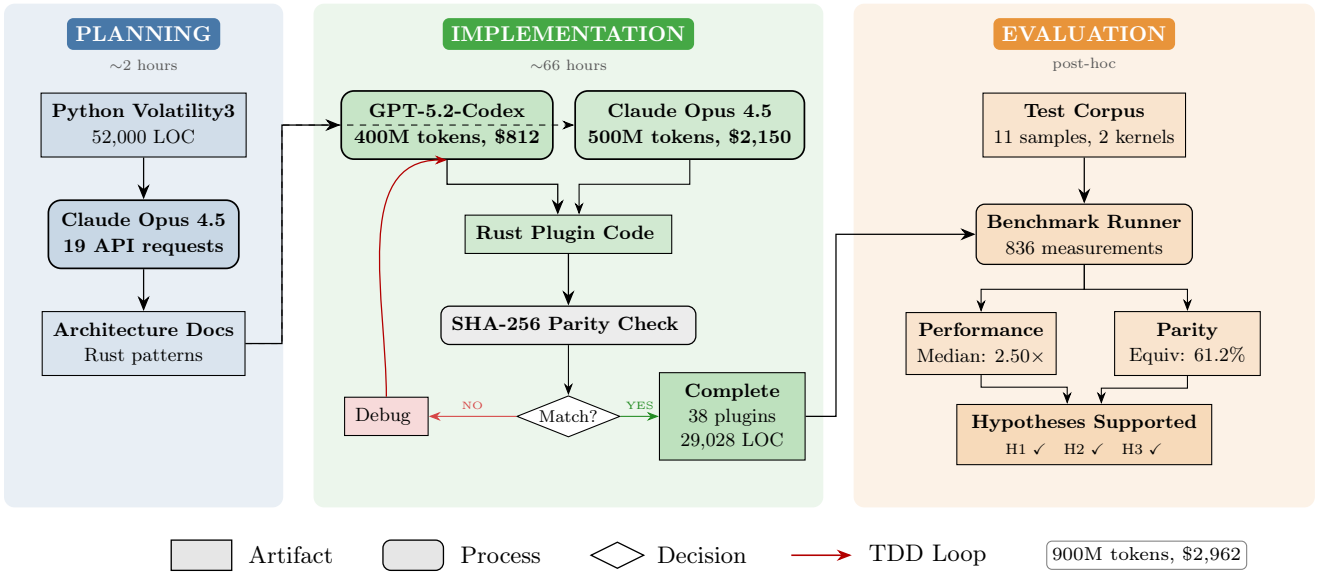

Figure 1: LLM-assisted migration and evaluation workflow. **Planning**: Claude Opus 4.5 analyzes Python Volatility3 to extract architectural patterns. **Implementation**: GPT-5.2-Codex generates plugin code while Claude assists with debugging; the red loop represents test-driven development where hash mismatches trigger refinement. **Evaluation**: Systematic benchmarking validates all three research hypotheses.

3.3. Parity Validation Framework

To assess functional equivalence, we applied a canonical output comparison procedure. Both implementations emit CSV-formatted output. The validation process was:

1. Execute both Python and Rust implementations against identical inputs
2. Capture standard output from each execution
3. Normalize output by extracting non-empty lines
4. Compute SHA-256 hash of normalized output
5. Classify results into three categories:

Classification Terminology:

- **Equivalent**: Both implementations executed successfully and produced identical output (matching hashes)
- **Divergent**: Both implementations executed successfully but produced different output (non-matching hashes)
- **Incomparable**: One or both implementations failed to execute, preventing meaningful comparison

This hash-based method enforces strict equivalence: any difference in content, ordering, or formatting is classified as Divergent. Although this criterion is stricter than semantic equivalence (for example, reordered but identical rows), strict matching is valuable here because it exposes implementation discrepancies that would otherwise remain hidden.

3.4. Benchmark Protocol

We measured execution performance under two cache conditions:

- **Warm cache**: Symbol tables pre-parsed and cached. Measures steady-state plugin performance representative of repeated analysis.
- **Cold cache**: Symbol cache cleared before each execution. Measures first-run performance including ISF parsing overhead.

For each plugin-sample pair, we recorded wall-clock time, CPU time, exit status, and output size. The complete matrix contained 38 plugins × 11 samples × 2 cache states = 836 executions. One plugin-sample tuple proved unstable in both cache states and was removed from final analysis, yielding 417 analyzed pairs per cache condition (834 analyzed runs total). Benchmarks ran on Windows 11 with Windows Subsystem for Linux (WSL)2 (Linux kernel 6.6.87.2), Rust 1.93.0, and Python Volatility3 2.28.0.

4. Results

This section presents empirical findings along four dimensions: execution robustness, parity outcomes, runtime performance, and hypothesis evaluation.

4.1. Execution Robustness

The most immediate result is a robust execution gap between implementations. Across 417 analyzed plugin-sample pairs per cache condition (after excluding one unstable tuple), Rust completed every run. Python Volatility3 failed on 47 pairs (11.3%), primarily because of runtime errors and symbol-compatibility issues.

Figure 2 shows that these failures are concentrated in a small subset of plugins rather than spread uniformly across the benchmark matrix. As a result, every INCOMPARABLE case in our classification is caused by Python execution failure; Rust exhibited zero execution failures across all 834 analyzed runs.

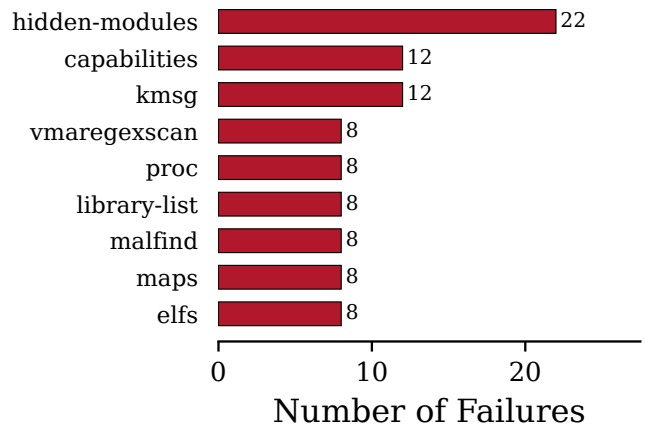

Figure 2: Python execution failures by plugin (warm and cold cache). All 47 failures originate from the Python reference implementation; Rust completed 100% of analyzed executions. Failures are concentrated in plugins with symbol compatibility issues (`hidden-modules`, `vmaregexscan`) and structure traversal errors (`proc`).

Table 4 summarizes failure counts and root causes by plugin.

Table 4: Python execution failures by plugin.

Plugin	Fails	Root Cause
hidden-modules	11	Symbol incompatibility
vmaregexscan	11	Virtual Memory Area (VMA) enumeration failure
proc	9	Structure traversal error
capabilities	6	Application Programming Interface (API) version mismatch
kmsg	6	Bitfield arithmetic error
library-list	4	Timeout on kernel 6.14

4.2. Parity Classification

Figure 3 compares parity outcomes under warm and cold cache conditions. Across all 417 executions per condition, 61.2% reach exact output equivalence. Restricting to comparable executions, where both implementations completed, increases the equivalence rate to 68.9%.

Table 5 reports parity counts for warm-cache execution.

Thirteen plugins achieved 100% equivalence across all samples: `boottime`, `check-afinfo`, `check-creds`, `check-idt`, `check-modules`, `check-syscall`, `ip-addr`, `ip-link`, `iomem`, `kallsyms`, `keyboard-notifiers`, `ps-list`, and `tty-check`. These plugins generally combine deterministic iteration behavior with relatively simple output formatting.

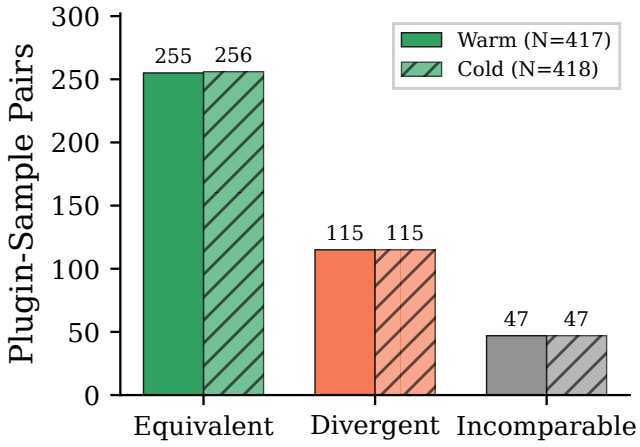

Figure 3: Parity classification comparison between warm and cold cache conditions (N=417 per condition). EQUIVALENT indicates identical SHA-256 output hashes; DIVERGENT indicates both ran but outputs differ; INCOMPARABLE indicates Python failed. The class distribution is stable across cache conditions.

Table 5: Parity classification results (warm cache).

Classification	Count	%
EQUIVALENT	255	61.2
DIVERGENT	115	27.6
INCOMPARABLE (Python failed)	47	11.3
Total	**417**	**100**

A closer look at the 115 DIVERGENT cases indicates that most differences are representational rather than analytical. Approximately 70% are attributable to ordering and formatting differences (for example row order or hexadecimal representation), whereas about 10% correspond to logic-level discrepancies that required implementation fixes, mainly in complex traversal plugins.

4.3. Performance Analysis

Figure 4 shows the speedup distribution across 370 warm-cache comparable runs. The distribution is strongly right-skewed, reflecting very large gains in a subset of symbol-resolution-heavy plugins.

Table 6 summarizes speedup statistics for warm and cold cache conditions.

Figure 5 provides a complementary view by plotting Python execution time against Rust execution time on log-log axes, with points colored by parity status.

4.4. Performance by Plugin

Figure 6 ranks plugins by geometric-mean speedup with 95% bootstrap confidence intervals, making cross-plugin

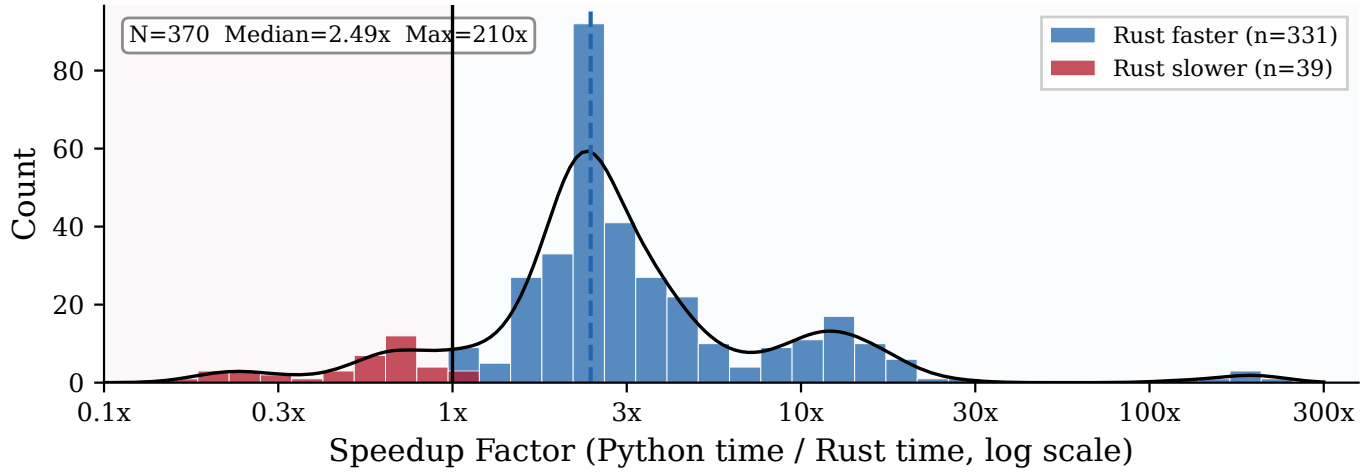

Figure 4: Speedup distribution for comparable runs where both implementations completed (N=370). The histogram shows log-transformed speedup factors with a kernel-density overlay. The dashed line marks the median speedup of 2.50×. Blue bars indicate runs where Rust is faster; red bars indicate runs where Python is faster. The distribution exhibits pronounced right skew with outliers above 200×.

Table 6: Speedup distribution statistics.

Statistic	Warm	Cold
Minimum	0.18×	0.20×
25th percentile	2.17×	2.14×
Median	**2.50×**	**2.45×**
75th percentile	5.01×	5.12×
Maximum	220×	242×
Mean	7.05×	7.11×

performance differences explicit.

Performance patterns remain consistent across memory samples, indicating that observed speedups are primarily plugin-driven rather than sample-driven. Plugins with large gains on one sample tend to exhibit similar gains across the corpus.

4.5. Hypothesis Evaluation

Table 7 summarizes the evaluation of the three research hypotheses.

Table 7: Hypothesis evaluation.

Hypothesis	Threshold	Observed	Result
H1: Performance	≥2×	2.50×	**Supported**
H2: Correctness	≥60%	61.2%	**Supported**
H3: Robustness	Rust≥Py	100 vs 88.7%	**Supported**

Under the evaluation criteria in Section 3, all three hypotheses are supported. Rust exceeds the H1 threshold with a 2.50× median speedup. For H2, exact equivalence reaches 61.2% across all runs and 68.9% across comparable runs, both above the 60% threshold. For H3, Rust completed all analyzed runs, while Python failed on 11.3% of runs.

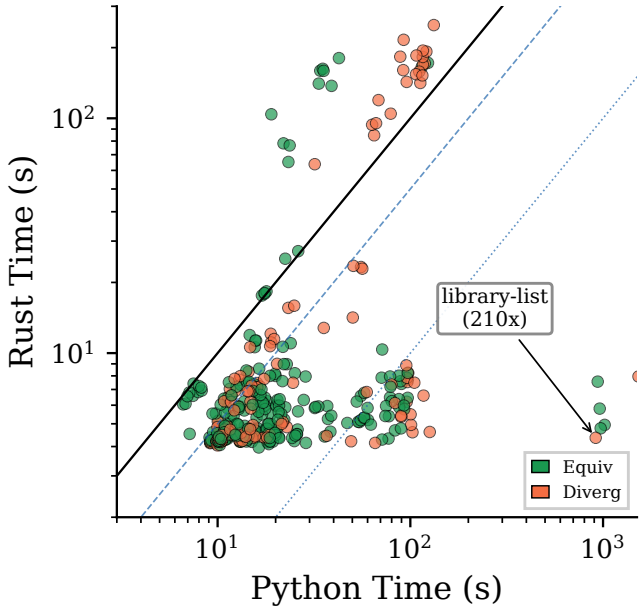

Figure 5: Execution time comparison (log-log scale). Each point represents one plugin-sample pair; color indicates parity status. Points below the diagonal indicate Rust is faster. Reference lines mark 2× and 10× speedup thresholds. The annotated `library-list` points illustrate the largest observed gains.

5. Discussion

This section interprets the empirical results in terms of performance behavior, parity quality, robustness differences, and the practical role of LLM-assisted migration.

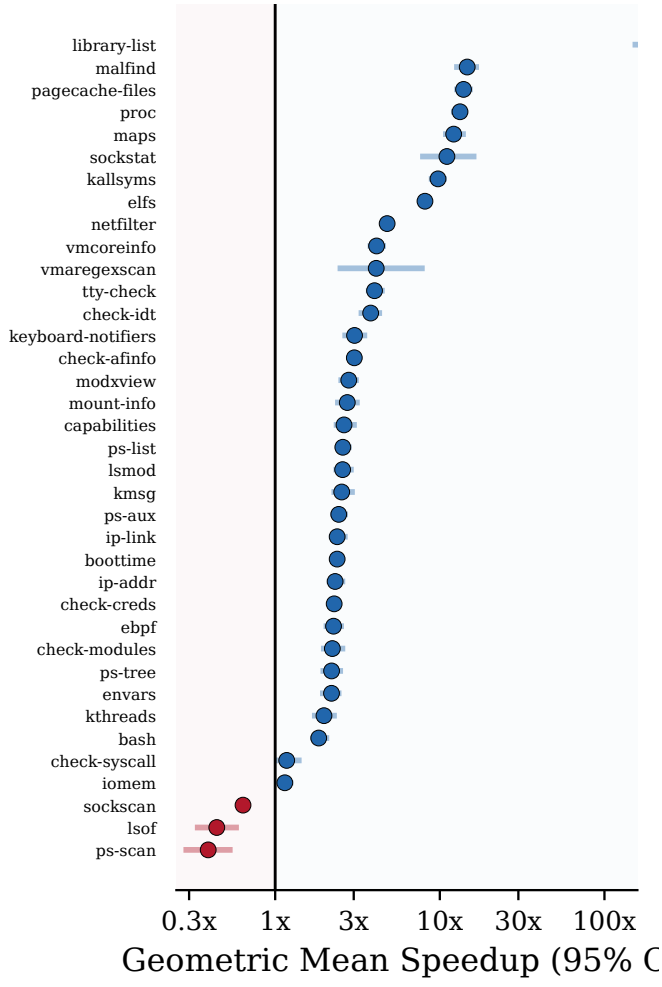

Figure 6: Per-plugin geometric-mean speedup with 95% bootstrap confidence intervals. Plugins are sorted by speedup; blue indicates Rust faster, red indicates Rust slower. Three plugins (`ps-scan`, `lsof`, `sockscan`) show slowdown because Python uses multiprocessing while our Rust implementation in this phase is single-threaded.

5.1. Performance Characteristics

The median speedup of 2.50× exceeds the hypothesized 2× threshold, but gains are not uniform across plugins. Instead, the distribution separates into three practically meaningful performance regimes.

High-speedup plugins (10× and above). Symbol-resolution-centric plugins, including `library-list` and `kallsyms`, show the largest gains. These workloads execute repeated hash lookups and string-processing operations at high frequency, where interpreter dispatch overhead dominates Python runtime. Rust compiles these paths to native code and removes per-operation dispatch. For `library-list`, representative runs dropped from minutes in Python to single-digit seconds in Rust.

Moderate-speedup plugins (2–10×). Most plugins fall in this band, which corresponds to typical structure-traversal workloads. Representative cases include process enumeration (`ps-list`, 2.6×), integrity checks (`check-idt`, 3.9×), and filesystem analysis (`mount-info`, 3.7×). These plugins repeatedly traverse pointer chains and structured fields, where compiled offset-based access in Rust produces steady gains over Python object and attribute resolution.

Slowdown plugins (0.18–1×). Three plugins were slower in Rust: `ps-scan` (0.45×), `lsof` (0.49×), and `sockscan` (0.64×). These are scan-intensive tasks for which Python Volatility3 already uses `multiprocessing` across cores. Our Rust implementations were single-threaded in this phase, so they could not offset Python's parallel advantage. This pattern reflects current implementation choices rather than a language-level limit, and it identifies parallel scanning as the most direct optimization target.

5.2. Correctness Analysis

The exact-equivalence rate, 61.2% across all runs and 68.9% among comparable runs, exceeds the 60% threshold. Interpretation still requires decomposing the 115 DIVERGENT cases by forensic significance.

Ordering differences (46 cases, 40%). Hash-table and set iteration behavior differs between implementations. In many cases, outputs contain the same records but in different row order, which is forensically equivalent. Deterministic sorting in the rendering layer would remove most of these divergences without changing analysis logic.

Formatting differences (35 cases, 30%). These include hexadecimal casing, leading-zero choices, whitespace padding, and pointer display conventions. They are cosmetic and do not alter analytical meaning. A stricter shared output-format contract would resolve them.

Incomplete traversal (23 cases, 20%). In these cases, Rust emitted fewer rows than Python, indicating missing enumeration logic under edge conditions. The `maps` plugin is illustrative: some VMA regions were missed because corner cases in memory-mapping traversal were not handled completely.

Logic differences (11 cases, 10%). These are genuine analytical mismatches, where Rust and Python disagree on extracted values. Most trace to integer-overflow behavior and signed/unsigned conversion boundaries. These are correctness defects and require direct remediation before production use.

5.3. Python Failure Analysis

The 47 INCOMPARABLE cases are concentrated in six plugins, and all arise from Python failures; Rust completed all analyzed executions. The failure distribution is therefore systematic rather than random.

hidden-modules (11 failures). Python raised `SymbolError` when kernel symbols were absent from the 6.14 ISF profile (for example, "`Symbol table has no type named module_layout`"). Rust handled these cases with defensive checks and returned empty results instead of terminating.

vmaregexscan (11 failures). Python encountered null-dereference paths while traversing malformed VMA structures (for example, "`TypeError: 'NoneType' object is not subscriptable`"). Rust's explicit optional-value handling forced these branches to be handled.

proc (9 failures). Traversal failed when process credential layouts differed on kernel 6.14. Python raised `InvalidAddressException`, whereas Rust returned bounded errors and continued execution.

capabilities (6 failures). API version mismatch in which Python assumes structure fields that are absent in newer kernels.

kmsg (6 failures). These failures involved bitfield arithmetic paths in kernel-message parsing. The corresponding Rust paths were typed and validated explicitly during implementation.

library-list (4 failures). Python exceeded the 5-minute timeout on kernel 6.14 samples because of symbol-resolution overhead; Rust completed equivalent analysis within seconds.

Overall, these failures reflect input-sensitive weakness in specific Python plugin paths rather than broad corpus instability.

5.4. Implications for Forensics Practice

From an operational perspective, a 2.50× median speedup implies substantial cumulative time savings in multi-plugin workflows. Using our measured ratios, a 20-plugin triage pass on an 8 GB image decreases from roughly 15 minutes to roughly 6 minutes, and that reduction compounds across multi-endpoint investigations.

The robustness gap (100% versus 88.7%) is equally important for automated pipelines, where execution failures trigger manual intervention and rerun overhead. In that setting, reducing failure frequency can matter as much as reducing runtime.

5.5. LLM-Assisted Migration Effectiveness

The migration consumed 900 million tokens at an estimated cost of $2,962 to produce 29,028 LOC, or roughly $0.10 per line. The measured implementation window was 68 active hours. Any manual-effort counterfactual remains uncertain, but the observed timeline indicates that LLM support can compress implementation and debugging cycles when paired with strict parity testing.

At the same time, the 61.2% exact-equivalence rate underscores that generated code still demands rigorous human validation. The logic-difference subset of DIVERGENT cases would translate directly into incorrect forensic outputs if released without review. In practice, LLM-assisted migration accelerates implementation, but it does not replace systematic verification.

6. Threats to Validity

We discuss limitations of the study across four validity dimensions.

Internal validity. Benchmark measurements are single executions without repeated trials, so we cannot estimate run-to-run variance. Timing includes process startup overhead, which can disproportionately influence fast plugins. Python failures on 47 plugin-sample pairs also constrain comparison scope, because parity analysis among successful runs excludes these cases.

External validity. The corpus includes 11 samples across two kernel versions, which provides useful diversity but does not cover the full range of memory configurations encountered in practice. We did not evaluate adversarial samples using anti-forensics techniques. Performance characteristics may also differ on other hardware platforms or with substantially different memory-image sizes.

Construct validity. Hash-based parity is stricter than semantic equivalence, so some DIVERGENT cases may still be forensically equivalent. Our speedup metrics also weight all plugins equally, regardless of investigative importance. The comparable-execution parity rate (68.9%) therefore should not be interpreted as complete semantic equivalence across the full matrix.

Reproducibility. Source code, benchmark scripts, and sample manifests are available. Memory samples must be acquired independently because of size and distribution constraints. Results may vary with different Volatility3 versions, symbol table formats, and host environments.

7. Related Work

We situate our work within four research areas: memory forensics frameworks, systems software in Rust, LLM code generation, and software migration.

7.1. Memory Forensics Frameworks

The Volatility framework emerged from academic memory-analysis research and evolved into the dominant open-source toolkit for volatile-memory investigation (Case and Richard III, 2017; Schatz and Cohen, 2017). Case and Richard synthesized methodological foundations for the field and identified open research questions (Case and Richard III, 2017). Subsequent studies expanded memory forensics into specialized domains: Block and Dewald analyzed Linux user-space heap structures (Block and Dewald, 2017) and Windows page-table manipulation for injected-code detection (Block and Dewald, 2019); Lewis et al. examined memory forensics challenges in the Windows Subsystem for Linux (Lewis et al., 2018); and Graziano et al. proposed hypervisor-based memory-acquisition methods (Graziano et al., 2013).

Alternative frameworks such as Rekall explored different architectural tradeoffs but were not sustained at the same maintenance level. Commercial suites, including Magnet

AXIOM, provide proprietary workflows. Franzen et al. introduced Katana for binary-only forensic analysis of Linux memory snapshots (Franzen et al., 2022), and Nyholm et al. surveyed the evolution of volatile-memory forensics techniques (Nyholm et al., 2022). Across this literature, many contributions target new artifact classes or acquisition methods, while fewer studies quantify the performance impact of implementation-language choices in core framework components.

Recent surveys catalog memory-analysis techniques for malware detection (Dehfouli and Habibi Lashkari, 2025; Sihwail et al., 2021) and position memory forensics within broader digital investigation workflows (Javed et al., 2022; Raghavan, 2013). Our work complements these studies with an empirical language-migration evaluation grounded in plugin-level benchmark measurements.

7.2. Systems Software in Rust

Rust is now a mature systems language with formal reasoning support for memory safety (Jung et al., 2017; Matsakis and Klock, 2014). Jung et al. introduced RustBelt, the first formal proof framework for core Rust safety guarantees (Jung et al., 2017), and later formalized the Stacked Borrows aliasing model (Jung et al., 2019). Astrauskas et al. showed how Rust's type system supports modular specification and verification (Astrauskas et al., 2019), and also analyzed unsafe-Rust usage in practice (Astrauskas et al., 2020).

At the ecosystem level, Rust has supported successful reimplementations of established utilities such as `ripgrep`, `fd`, and `bat`. Empirical studies report strong performance across many workloads (Zhang et al., 2022b). Other work has examined memory and thread-safety patterns in deployed Rust projects (Qin et al., 2020; Evans et al., 2020), and Bae et al. developed Rudra for ecosystem-scale detection of memory-safety bugs (Bae et al., 2021).

Rust has also entered kernel-level development. The Linux kernel accepts Rust modules for selected driver and infrastructure work (Panter and Eisty, 2024; Li et al., 2024), and Chen and Wu studied corresponding module-development patterns (Chen and Wu, 2022). Our study extends this trajectory to forensic analysis infrastructure and evaluates migration outcomes at plugin level.

7.3. LLM Code Generation

Large language models have demonstrated strong code-generation capability on competitive and benchmarked programming tasks (Li et al., 2022; Xu et al., 2022). Li et al. reported competition-level performance with AlphaCode (Li et al., 2022), Zheng et al. developed CodeGeeX for multilingual generation (Zheng et al., 2023), and Zan et al. synthesized the broader landscape of LLM-based code generation (Zan et al., 2023).

Follow-on work evaluated LLM use in debugging (Li et al., 2023), test generation (Lemieux et al., 2023; El Haji et al., 2024), and developer interaction patterns (Barke et al., 2023; Ross et al., 2023). Additional studies examined educational implications of generative coding tools (Prather et al., 2023). Collectively, these results support LLM utility in software tasks, while also highlighting quality and validation challenges.

Cross-language migration is a particularly structured setting because source code provides an executable behavioral reference. Ahmad et al. introduced corpora and methods for Java-Python translation (Ahmad et al., 2023b,a), and Huang et al. explored distillation strategies for program transformation (Huang et al., 2023). Pan et al. characterized bug patterns introduced during LLM-based code translation (Pan et al., 2024), and Omidvar-Tehrani et al. evaluated human-AI collaboration in code migration workflows (Omidvar Tehrani et al., 2024). Our work adds empirical evidence for migration of a large, semantics-sensitive systems codebase.

7.4. Software Migration

Language-migration literature more often addresses framework transitions or service decomposition than full reimplementation of mature tooling (Balalaie et al., 2018; Fritzsch et al., 2019). Mens and Tourwe surveyed refactoring techniques (Mens and Tourwé, 2004), Dig and Johnson examined API evolution patterns (Dig and Johnson, 2006), and Golubev et al. reported large-scale refactoring practices (Golubev et al., 2021). These works inform migration process design but provide limited quantitative evidence for cross-language reimplementation under strict behavioral validation.

Technical-debt studies analyze migration cost-benefit tradeoffs (Kruchten et al., 2012; Li et al., 2015; Ampatzoglou et al., 2015). Related work has documented challenges in modernizing legacy systems for cloud environments (Gholami et al., 2017) and extracting microservices from monoliths (Mazlami et al., 2017). This literature frames the economic motivation for modernization, but rarely reports plugin-level correctness and performance outcomes for security-tooling migrations.

C-to-Rust migration has received focused attention. Emre et al. proposed translation techniques from C to safer Rust (Emre et al., 2021) and later analyzed aliasing limits in such translations (Emre et al., 2023). Zhang et al. introduced ownership-guided C-to-Rust translation methods (Zhang et al., 2023). Our study contributes complementary data for Python-to-Rust migration, including token consumption, development time, parity outcomes, and runtime performance.

8. Conclusion

This paper presents a systematic study of migrating a production memory-forensics framework from Python to Rust using LLM-assisted development. The resulting implementation comprises 29,028 LOC across 38 Linux plugins, developed over 68 active hours with 900 million model tokens ($2,962).

Our evaluation across 417 plugin-sample pairs shows that compiled-language reimplementation yields substantial practical benefits. The Rust implementation achieves a median 2.50× speedup, with structure-traversal plugins reaching up to 210× acceleration through reduced interpreter overhead. Execution robustness also improves: Rust completed 100% of analyzed runs, while Python failed on 11.3%, and all observed failures originated in the reference implementation.

For the forensics community, these results indicate that performance-critical analysis infrastructure can benefit materially from compiled implementations, particularly for structure-intensive operations. The migration outcomes also show that LLM-assisted workflows can support large cross-language ports when paired with strict parity validation and iterative debugging. Improved execution robustness is especially relevant for automated forensic pipelines, where tool failures interrupt evidence-processing workflows.

The performance demands of modern DFIR, especially as memory images continue to grow and triage timelines remain constrained, motivate continued investment in high-performance forensic tooling. This work provides an empirical path toward that objective.

9. Future Work

The following directions would extend this work:

- **Platform expansion**: The current implementation covers Linux plugins only. Extending coverage to **Windows** and **macOS** plugins would address most of the Volatility3 ecosystem. Windows plugins represent the largest category (process analysis, registry extraction, malware detection), while macOS support is increasingly relevant in enterprise investigations. The architectural patterns established here, including symbol-table handling, object construction, and rendering pipelines, transfer directly to these platforms.
- **Parity improvement**: Address remaining DIVERGENT cases to approach full equivalence, with emphasis on output-order normalization and incomplete traversal logic. A semantic-equivalence layer, comparing parsed structures rather than raw text output, could reduce false divergence from formatting differences.
- **Parallel scanning**: Implement parallel memory scanning with the `rayon` crate to recover performance on scan-intensive plugins where Python currently benefits from multiprocessing. Pool scanning and signature matching are natural parallelization targets.
- **Adversarial robustness**: Evaluate behavior on memory samples that use anti-forensics techniques such as Direct Kernel Object Manipulation (DKOM), process hiding, and timestamp manipulation. This would clarify how the implementation behaves under intentionally adversarial structure corruption.
- **Production deployment**: Validate the implementation in real-world incident-response scenarios to assess operational utility beyond benchmark performance. Integration with forensic workflow tooling and SIEM platforms would support broader adoption.
- **Incremental migration tooling**: Develop tooling to automate repetitive portions of the migration process, enabling broader community contribution to Windows and macOS plugin ports.

References

Ahmad, W., Chakraborty, S., Ray, B., Chang, K.W., 2023a. Summarize and generate to back-translate: Unsupervised translation of programming languages, in: Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics, pp. 1528–1542.

Ahmad, W., Tushar, M.G.R., Chakraborty, S., Chang, K.W., 2023b. Avatar: A parallel corpus for java-python program translation, in: Findings of the Association for Computational Linguistics: ACL 2023, pp. 2268–2281.

Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., 2015. The financial aspect of managing technical debt: A systematic literature review. Information and Software Technology 64, 52–73.

Astrauskas, V., Matheja, C., Poli, F., Müller, P., Summers, A.J., 2020. How do programmers use unsafe rust? Proceedings of the ACM on Programming Languages 4, 1–27.

Astrauskas, V., Müller, P., Poli, F., Summers, A.J., 2019. Leveraging rust types for modular specification and verification. Proceedings of the ACM on Programming Languages 3, 1–30.

Bae, Y., Kim, Y., Askar, A., Lim, J., Kim, T., 2021. Rudra: Finding memory safety bugs in rust at the ecosystem scale, in: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, pp. 84–99.

Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D.A., Lynn, T., 2018. Microservices migration patterns. Software: Practice and Experience 48, 2019–2042.

Barany, G., 2014. Python interpreter performance deconstructed, in: Proceedings of the Workshop on Dynamic Languages and Applications, pp. 1–9.

Barke, S., James, M.B., Polikarpova, N., 2023. Grounded copilot: How programmers interact with code-generating models. Proceedings of the ACM on Programming Languages 7, 85–111.

Block, F., Dewald, A., 2017. Linux memory forensics: Dissecting the user space process heap. Digital Investigation 22, S66–S75.

Block, F., Dewald, A., 2019. Windows memory forensics: Detecting (un) intentionally hidden injected code by examining page table entries. Digital Investigation 29, S3–S12.

Case, A., Richard III, G.G., 2017. Memory forensics: The path forward. Digital investigation 20, 23–33.

Chen, S.F., Wu, Y.S., 2022. Linux kernel module development with rust, in: 2022 IEEE Conference on Dependable and Secure Computing (DSC), IEEE. pp. 1–2.

Dehfouli, Y., Habibi Lashkari, A., 2025. Memory analysis for malware detection: A comprehensive survey using the oscar methodology. ACM Computing Surveys 58, 1–58.

Dig, D., Johnson, R., 2006. How do apis evolve? a story of refactoring. Journal of software maintenance and evolution: Research and Practice 18, 83–107.

El Haji, K., Brandt, C., Zaidman, A., 2024. Using github copilot for test generation in python: An empirical study, in: Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024), pp. 45–55.

Emre, M., Boyland, P., Parekh, A., Schroeder, R., Dewey, K., Hardekopf, B., 2023. Aliasing limits on translating c to safe rust. Proceedings of the ACM on Programming Languages 7, 551–579.

Emre, M., Schroeder, R., Dewey, K., Hardekopf, B., 2021. Translating c to safer rust. Proceedings of the ACM on Programming Languages 5, 1–29.

Evans, A.N., Campbell, B., Soffa, M.L., 2020. Is rust used safely by software developers?, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 246–257.

Franzen, F., Holl, T., Andreas, M., Kirsch, J., Grossklags, J., 2022. Katana: Robust, automated, binary-only forensic analysis of linux memory snapshots, in: Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses, pp. 214–231.

Fritzsch, J., Bogner, J., Wagner, S., Zimmermann, A., 2019. Microservices migration in industry: Intentions, strategies, and challenges, in: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE. pp. 481–490.

Gholami, M.F., Daneshgar, F., Beydoun, G., Rabhi, F., 2017. Challenges in migrating legacy software systems to the cloud—an empirical study. Information Systems 67, 100–113.

Golubev, Y., Kurbatova, Z., AlOmar, E.A., Bryksin, T., Mkaouer, M.W., 2021. One thousand and one stories: a large-scale survey of software refactoring, in: Proceedings of the 29th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp. 1303–1313.

Graziano, M., Lanzi, A., Balzarotti, D., 2013. Hypervisor memory forensics, in: International Workshop on Recent Advances in Intrusion Detection, Springer. pp. 21–40.

Hay, B., Nance, K., 2008. Forensics examination of volatile system data using virtual introspection. ACM SIGOPS Operating Systems Review 42, 74–82.

Huang, Y., Qi, M., Yao, Y., Wang, M., Gu, B., Clement, C., Sundaresan, N., 2023. Program translation via code distillation, in: Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pp. 10903–10914.

Javed, A.R., Ahmed, W., Alazab, M., Jalil, Z., Kifayat, K., Gadekallu, T.R., 2022. A comprehensive survey on computer forensics: State-of-the-art, tools, techniques, challenges, and future directions. IEEE Access 10, 11065–11089.

Jung, R., Dang, H.H., Kang, J., Dreyer, D., 2019. Stacked borrows: an aliasing model for rust. Proceedings of the ACM on Programming Languages 4, 1–32.

Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D., 2017. Rustbelt: Securing the foundations of the rust programming language. Proceedings of the ACM on Programming Languages 2, 1–34.

Kruchten, P., Nord, R.L., Ozkaya, I., 2012. Technical debt: From metaphor to theory and practice. Ieee software 29, 18–21.

Lemieux, C., Inala, J.P., Lahiri, S.K., Sen, S., 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE. pp. 919–931.

Lewis, N., Case, A., Ali-Gombe, A., Richard III, G.G., 2018. Memory forensics and the windows subsystem for linux. Digital Investigation 26, S3–S11.

Li, T.O., Zong, W., Wang, Y., Tian, H., Wang, Y., Cheung, S.C., Kramer, J., 2023. Nuances are the key: Unlocking chatgpt to find failure-inducing tests with differential prompting, in: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE. pp. 14–26.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al., 2022. Competition-level code generation with alphacode. Science 378, 1092–1097.

Li, Z., Avgeriou, P., Liang, P., 2015. A systematic mapping study on technical debt and its management. Journal of systems and software 101, 193–220.

Li, Z., Narayanan, V., Chen, X., Zhang, J., Burtsev, A., 2024. Rust for linux: Understanding the security impact of rust in the linux kernel, in: 2024 Annual Computer Security Applications Conference (ACSAC), IEEE. pp. 548–562.

Matsakis, N.D., Klock, F.S., 2014. The rust language, in: Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, pp. 103–104.

Mazlami, G., Cito, J., Leitner, P., 2017. Extraction of microservices from monolithic software architectures, in: 2017 IEEE International Conference on Web Services (ICWS), IEEE. pp. 524–531.

Meier, R., Gross, T.R., 2019. Reflections on the compatibility, performance, and scalability of parallel python, in: Proceedings of the 15th ACM SIGPLAN international symposium on dynamic languages, pp. 91–103.

Meier, R., Rigo, A., Gross, T.R., 2018. Virtual machine design for parallel dynamic programming languages. Proceedings of the ACM on Programming Languages 2, 1–25.

Mens, T., Tourwé, T., 2004. A survey of software refactoring. IEEE Transactions on software engineering 30, 126–139.

Nyholm, H., Monteith, K., Lyles, S., Gallegos, M., DeSantis, M., Donaldson, J., Taylor, C., 2022. The evolution of volatile memory forensics. Journal of Cybersecurity and Privacy 2, 556–572.

Omidvar Tehrani, B., M, I., Anubhai, A., 2024. Evaluating human-ai partnership for llm-based code migration, in: Extended abstracts of the CHI conference on human factors in computing systems, pp. 1–8.

Or-Meir, O., Nissim, N., Elovici, Y., Rokach, L., 2019. Dynamic malware analysis in the modern era—a state of the art survey. ACM Computing Surveys (CSUR) 52, 1–48.

Pan, R., Ibrahimzada, A.R., Krishna, R., Sankar, D., Wassi, L.P., Merler, M., Sobolev, B., Pavuluri, R., Sinha, S., Jabbarvand, R., 2024. Lost in translation: A study of bugs introduced by large language models while translating code, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pp. 1–13.

Panter, S., Eisty, N., 2024. Rusty linux: Advances in rust for linux kernel development, in: Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 496–502.

Prather, J., Denny, P., Leinonen, J., Becker, B.A., Albluwi, I., Craig, M., Keuning, H., Kiesler, N., Kohn, T., Luxton-Reilly, A., et al., 2023. The robots are here: Navigating the generative ai revolution in computing education, in: Proceedings of the 2023 working group reports on innovation and technology in computer science education. ACM, pp. 108–159.

Qin, B., Chen, Y., Yu, Z., Song, L., Zhang, Y., 2020. Understanding memory and thread safety practices and issues in real-world rust programs, in: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 763–779.

Raghavan, S., 2013. Digital forensic research: current state of the art. Csi Transactions on ICT 1, 91–114.

Ross, S.I., Martinez, F., Houde, S., Muller, M., Weisz, J.D., 2023. The programmer's assistant: Conversational interaction with a large language model for software development, in: Proceedings of the 28th international conference on intelligent user interfaces, pp. 491–514.

Schatz, B., Cohen, M., 2017. Advances in volatile memory forensics. Digital Investigation 100, 1.

Schuster, A., 2008. The impact of microsoft windows pool allocation strategies on memory forensics. digital investigation 5, S58–S64.

Sihwail, R., Omar, K., Ariffin, K.A.Z., 2021. An effective memory analysis for malware detection and classification. Computers, Materials, & Continua 67, 2301.

Sudhakar, Kumar, S., 2020. An emerging threat fileless malware: a survey and research challenges. Cybersecurity 3, 1.

Vidas, T., 2007. The acquisition and analysis of random access memory. Journal of Digital Forensic Practice 1, 315–323.

Xu, F.F., Alon, U., Neubig, G., Hellendoorn, V.J., 2022. A systematic evaluation of large language models of code, in: Proceedings of the 6th ACM SIGPLAN international symposium on machine programming, pp. 1–10.

Zan, D., Chen, B., Zhang, F., Lu, D., Wu, B., Guan, B., Yongji, W., Lou, J.G., 2023. Large language models meet nl2code: A survey, in: Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 7443–7464.

Zhang, H., David, C., Yu, Y., Wang, M., 2023. Ownership guided c to rust translation, in: International Conference on Computer Aided Verification, Springer. pp. 459–482.

Zhang, Q., Xu, L., Zhang, X., Xu, B., 2022a. Quantifying the interpretation overhead of python. Science of Computer Programming 215, 102759.

Zhang, Y., Zhang, Y., Portokalidis, G., Xu, J., 2022b. Towards understanding the runtime performance of rust, in: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1–6.

Zheng, Q., Xia, X., Zou, X., Dong, Y., Wang, S., Xue, Y., Shen, L., Wang, Z., Wang, A., Li, Y., et al., 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x, in: Proceedings of the 29th ACM SIGKDD conference on knowledge discovery and data mining, pp. 5673–5684.