Team Atlanta

ATLANTIS: *AI-driven Threat Localization, Analysis, and Triage Intelligence System*

+

TAESOO KIM, HYUNGSEOK HAN, SOYEON PARK, DAE R. JEONG, DOHYEOK KIM, DONGKWAN KIM, EUNSOO KIM, JIHO KIM, JOSHUA WANG, KANGSU KIM, SANGWOO JI, WOOSUN SONG, HANQING ZHAO, ANDREW CHIN, GYEJIN LEE, KEVIN STEVENS, MANSOUR ALHARTHI, YIZHUO ZHAI, CEN ZHANG, JOONUN JANG, YEONGJIN JANG, AMMAR ASKAR, DONGJU KIM, FABIAN FLEISCHER, JEONGIN CHO, JUNSIK KIM, KYUNGJOON KO, INSU YUN, SANGDON PARK, DOWOO BAIK, HAEIN LEE, HYEON HEO, MINJAE GWON, MINJAE LEE, MINWOO BAEK, SEUNGGI MIN, WONYOUNG KIM, YONGHWI JIN, YOUNGGI PARK, YUNJAE CHOI, JINHO JUNG, GWANHYUN LEE, JUNYOUNG JANG, KYUHEON KIM, YEONGHYEON CHA, AND YOUNGJOON KIM

https://team-atlanta.github.io/

September 19, 2025

Version 1.0

Revision History

Version	Date	Note
1.0	2025-09-17	Initial public release of the ATLANTIS final report.

Contents

1 Introduction

The Artificial Intelligence Cyber Challenge (AIxCC) [14], launched by DARPA in collaboration with ARPA-H, represents an unprecedented effort to revolutionize cybersecurity through artificial intelligence. This two-year competition (2023–2025), backed by $29.5 million in prizes and partnerships with leading AI companies including Anthropic, Google, and OpenAI, challenged teams to build autonomous Cyber Reasoning Systems (CRSs) that can discover and patch vulnerabilities at the speed and scale demanded by modern software ecosystems. The competition addresses a critical gap: while software complexity grows exponentially, human-driven vulnerability discovery remains fundamentally limited, leaving countless zero-day vulnerabilities undiscovered in the critical infrastructure that powers our digital society.

Team Atlanta claimed victory at the AIxCC Final Competition during DEF CON 33 in August 2025, culminating a journey that began as one of 42 entrants and continued as one of seven semifinalist teams. Our collaboration unites researchers from Georgia Institute of Technology, Samsung Research, KAIST, and POSTECH, bringing together complementary expertise in systems security, program analysis, and artificial intelligence. We developed ATLANTIS, a system that orchestrates state-of-the-art vulnerability discovery techniques—symbolic execution, directed fuzzing, and static analysis—while pioneering the deep integration of large language models to overcome the limitations of autonomous vulnerability discovery and patching.

This technical report documents the design philosophy, architectural decisions, and implementation strategies behind ATLANTIS. We present our solutions to the fundamental challenges of autonomous vulnerability discovery: scaling across diverse codebases from C to Java, achieving high precision while maintaining broad coverage, and generating patches that are semantically correct. Beyond describing our winning approach, we share the lessons learned from pushing the boundaries of what automated security tools can achieve when traditional program analysis meets modern AI. Our complete system is available as open source, enabling the community to build upon our work and advance the state of automated cybersecurity.

2 AIxCC Final Competition Overview

Goals. DARPA's Cyber Grand Challenge (CGC, 2014–2016) pioneered fully autonomous *cyber reasoning systems* that detected, exploited, and patched software flaws in a controlled testbed of simplified *binaries*, showcasing machine-on-machine defense using techniques like fuzzing and symbolic execution. In contrast, the AI Cyber Challenge (AIxCC, 2023–2025) builds on that foundation by harnessing modern AI, especially LLMs, to secure real-world *open-source software* critical to infrastructure such as healthcare and energy systems. While CGC served as a proof-of-concept on a contrived computer architecture, AIxCC emphasizes practical impact: finalist systems are required to open-source their CRSes and are supported for commercialization after the competition. The shift marks a move from lab demonstrations of autonomy to deployable, AI-powered tools designed to strengthen national cybersecurity at scale.

Competition Structure. The competition format differed substantially between the initial AIxCC Semifinal Competition (ASC) and the AIxCC Final Competition (AFC) that determined the ultimate winners. This report only describes the AFC format and our submission to it. The final format comprised three unscored exhibition rounds followed by a scored final round, with each round introducing challenge projects (CPs) derived from OSS-Fuzz [22] projects. Teams were required to process CPs within strict time and budget constraints, strategically allocating LLM API usage and Azure compute resources to maximize their vulnerability discovery and patching capabilities. Table 1 summarizes key parameters across all final competition rounds.

During competition rounds, no internet access was available except for AIxCC organizer-approved commercial LLM access and necessary resources to build the CRS. This isolated environment ensured fair competition while providing teams with essential AI capabilities. Note that teams are allowed to prepare their own fine-tuned custom LLMs and run these LLMs on Azure cloud during the competition.

Round	Date	Scored	LLM	Azure	Max Conc.[↑]	Repos	CPs (D+F+U)[*]	Delta	Full
Exhibition 1	04/01/2025	No	$10K	$20K	2	2	2 (2+0+0)	48h	N/A
Exhibition 2	05/06/2025	No	$10K	$20K	4	8	15 (9+6+0)	8h	24h
Exhibition 3	06/05/2025	No	$30K	$50K	8	14	30 (18+9+3)	6h	12h
Final	06/26/2025	Yes	$50K	$85K	8	30	55 (33+17+5)	6h	12h

[↑] Max Conc. refers to the maximum number of concurrent challenge projects teams could receive simultaneously.
[*] The number of CPs are sum of the numbers of Delta, Full, and Unharnessed CPs.

Table 1: AIxCC Final Competition Round Summary

2.1 Challenge Projects

OSS-Fuzz Foundation. Challenge projects were derived from OSS-Fuzz [22], Google's continuous fuzzing service that has discovered over 10,000 vulnerabilities in critical open-source software since 2016. OSS-Fuzz provided the foundation for AIxCC challenges through three key components: (1) fuzzing harnesses that exercise specific APIs and functionalities, (2) build configurations with various sanitizers (*e.g.,* AddressSanitizer, MemorySanitizer, and UndefinedBehaviorSanitizer for C/C++, and Jazzer for Java), and (3) real-world software packages ranging from libraries and parsers to complex applications.

Challenge Types. The competition featured two challenge types, known as *full* and *delta* modes. In both modes, teams began with bug discovery by providing a bug-triggering input, known as Proof-of-Vulnerability (PoV), using the provided fuzzing harnesses.

- **Full-scan challenges (full mode):** Teams were asked to discover bugs anywhere in the entire codebase reachable from the given harnesses. Unintended 0-day vulnerabilities were also scorable when demonstrated with PoVs.
- **Delta-scan challenges (delta mode):** Teams were asked to identify bugs that are *newly* introduced by provided source code changes in the form of a diff (representing a new commit or patch). PoVs discovered in the baseline source code were *not* scorable.

Starting from Exhibition Round 3, the competition also included unharnessed challenges that lacked pre-written fuzzing harnesses, requiring teams to generate their own test harnesses.

Challenge Project Structure. Each CP contained intentionally injected vulnerabilities ("challenge project vulnerabilities", or CPVs) within the original codebase, providing authentic environments for vulnerability discovery and remediation. A typical CP consisted of: (1) the source code repository with hidden vulnerabilities, (2) one or more fuzzing harnesses from OSS-Fuzz (except for unharnessed challenges), (3) build scripts and configurations, and (4) in delta mode, a diff showing modified files. Teams received no prior knowledge of vulnerability types or locations, thereby simulating realistic security assessment scenarios.

Competition Tasks. For each CP, teams' CRSes needed to (focusing on C/C++ and Java vulnerabilities):

- **Find PoVs:** Generate inputs that trigger crashes to demonstrate vulnerabilities in the challenge source code, not defects in the harnesses. Scorable vulnerabilities included standard vulnerability categories detectable by OSS-Fuzz and unintended 0-day vulnerabilities that were harness-reachable. Scorable outcomes did not need to be explicitly caused by sanitizers, encompassing crashes, exceptions, and timeouts (if configured).
- **Generate Patches:** Create source code modifications in unified diff format that fix vulnerabilities while preserving intended functionality. Patches could not modify harnesses or functional tests (*e.g.,* `test.sh` scripts or Maven test cases). Each patch was validated independently against the original challenge and underwent post-competition manual assessment for scoring.

- **Assess SARIF Reports:** Evaluate Static Analysis Results Interchange Format (SARIF) reports broadcast during round execution. SARIF reports describe potential vulnerabilities in active challenges, but are not guaranteed to be accurate and contain no PoVs. Teams submitted assessments (correct/incorrect) with justification descriptions. Correct SARIF reports describe harness-reachable, sanitizer-triggered crashes.
- **Bundle Submissions:** Group related submissions to indicate relationships between discoveries. Bundles could contain PoVs, patches, SARIF reports, and broadcast SARIF assessments. For example, a CRS could bundle a patch with the PoV that demonstrates the vulnerability it fixes, or combine a PoV with its SARIF assessment. Bundles required at least two components and could be modified after submission.

2.2 Scoring System

The competition employed a comprehensive scoring system designed to incentivize both the discovery of novel vulnerabilities and the generation of correct, deployable patches. Rather than attacking and analyzing other teams' binaries as in CGC, each CRS in AIxCC operated on the organizer-provided CPs independently. The scoring format evolved through several iterations; this report describes the final "Version 2" methodology [15].

The final team score is calculated as:

$$Team\ Score = \sum Challenge\ Scores$$

The challenge score is a weighted sum of CRS performance across vulnerability discovery, program repair, SARIF assessment, bundling, and accuracy for all vulnerabilities in the challenge. Each challenge score is calculated as:

$$Challenge\ Score = AM \times (VDS + PRS + SAS + BDL)$$

where AM is the Accuracy Multiplier, VDS is the Vulnerability Discovery Score, PRS is the Program Repair Score, SAS is the SARIF Assessment Score, and BDL is the Bundle Score. Here, the AM emphasizes submission quality over quantity. Any false submissions (incorrect patches, wrong SARIF assessments) significantly degrade the overall score, incentivizing teams to prioritize accuracy.

Points are awarded based on uniqueness of discovered vulnerabilities, correctness of generated patches, accuracy of SARIF assessments, and bonus points for bundled submissions (PoV + patch + SARIF assessment for the same vulnerability). Penalties apply for incorrect patches or multiple patch submissions addressing the same root cause. For comprehensive scoring details, refer to the official scoring guide [15].

2.3 Budget and Resource Constraints

Budget Allocation for Development. For development, teams were allocated $100,000 in Azure compute resources for infrastructure development and testing. Additionally, teams received substantial support from leading AI companies: Anthropic, Google, and OpenAI each provided $50,000 in LLM credits per team, totaling $150,000 for extensive experimentation and system development. Notably, we were allowed to exceed the given LLM credits, as long as we paid the difference ourselves.

Budget Allocation for the Competition. As shown in Table 1, each competition round provided fixed budgets for (1) LLM API usage ($10,000 ~ $50,000 per round) and (2) Azure compute resources ($20,000 ~ $85,000 per round). Teams had to strategically allocate these resources across multiple CPs that arrived in batches, with overlapping deadlines requiring careful resource management and scheduling. For instance, Exhibition Round 3 presented a particularly challenging scenario: with $30K LLM budget and $50K Azure budget, teams received up to *8 concurrent CPs* from 14 repositories with 30 total challenges, with tight deadlines of 6 hours for delta challenges and 12 hours for full challenges.

Module	OpenAI	Anthropic	Gemini	Grok	Total ($)	%
ATLANTIS-Multilang	8,839	8,839	884	0	18,563	37.1%
ATLANTIS-C	3,713	2,784	2,784	0	9,281	18.6%
ATLANTIS-Java	3,713	2,784	2,784	0	9,281	18.6%
ATLANTIS-Patch	3,713	7,425	1,238	0	12,375	24.8%
ATLANTIS-SARIF	75	100	250	75	500	1.0%
Total	20,052	21,933	7,940	75	50,000	100%

Table 2: Team Atlanta's LLM Budget Distribution Across Modules and Providers

Model	RPM	Input TPM	Output TPM	Total TPM
Claude Opus 4	Uncapped	875,000	175,000	1,050,000
Claude Sonnet 4	Uncapped	1,500,000	275,000	1,775,000
Claude Sonnet 3.7	6,250	125,000	50,000	175,000
Claude Sonnet 3.5	6,250	500,000	100,000	600,000
Claude Haiku 3.5/3	6,250	625,000	125,000	750,000
Claude Opus 3	6,250	250,000	18,750	268,750

RPM: Requests per minute, TPM: Tokens per minute

Table 3: Enhanced Anthropic Rate Limits for AIxCC Final Competition

Team Atlanta's Resource Strategy. For the final round, we strategically allocated the $85K Azure budget and $50K LLM budget to maximize concurrent CP processing while maintaining system reliability. Our Azure environment utilized two key resource quotas: 57,500 vCPUs of Standard `Ddsv6` Family for dynamically allocated CRS components, and 300 vCPUs of Standard `NCASv3_T4` Family with GPUs, with 64 cores dedicated to our custom fine-tuned LLM.

We allocated 20% of the Azure budget for infrastructure overhead, accounting for estimated operational costs over the 10-day competition period, including cluster management, system services, networking, specialized hardware, monitoring, and storage. The remaining 80% was reserved for dynamic CP processing, providing approximately 1,500 cores per full-mode CP (12-hour deadline) and 3,000 cores per delta-mode CP (6-hour deadline). Note that we dynamically allocated CPUs per CP based on the remaining budget and the number of harnesses each CP has (see §3.2).

Our LLM allocation strategy is shown in Table 2. Across our five modules—vulnerability discovery (ATLANTIS-Multilang, ATLANTIS-C, ATLANTIS-Java), program repair (ATLANTIS-Patch), and static analysis assessment (ATLANTIS-SARIF)— the budget distribution reflected each module's computational demands and strategic importance. ATLANTIS-Multilang received the largest allocation (37.1%) due to its central role in cross-language vulnerability discovery, while ATLANTIS-Patch received 24.8% for generating high-quality fixes. ATLANTIS-SARIF received a minimal allocation (1.0%) as static analysis assessment tasks were predictable in scope. For detailed system architecture, please refer to §3.

LLM Rate Limit Management. Beyond budget allocation, processing concurrent CPs required careful rate limit management. While OpenAI's Tier 5 (10K RPM, 30M TPM) and Google Gemini's Tier 3 (2K RPM, 8M TPM) limits were sufficient, Anthropic's standard Tier 4 limits (200K ITPM, 80K OTPM) were insufficient. AIxCC organizers negotiated enhanced Anthropic limits (over 4× increase) as shown in Table 3.

We implemented internal rate limits for each module to prevent any single module from exhausting the enhanced limits. For Sonnet 4, we set limits of 1,125K TPM for ATLANTIS-Multilang, 300K TPM for ATLANTIS-Patch, and 375K TPM each for ATLANTIS-C and ATLANTIS-Java. For Opus 4, we configured 750K TPM each for ATLANTIS-C and ATLANTIS-Java, with 300K TPM each for ATLANTIS-Multilang and ATLANTIS-Patch as fallback. This 3:1 ratio between ATLANTIS-Multilang and language-specific modules reflected their usage patterns, ensuring no single module could monopolize resources and enabling consistent performance across concurrent CPs.

Rank	Team	Total Score	Accuracy	VDS	PRS	SAS	BDL	Vulns Found
1	Team Atlanta	392.76	91.27%	79.71	171.1	5.99	136.38	43/70 (61%)
2	Trail of Bits	219.35	89.33%	52.49	101.21	1.0	65.29	28/70 (40%)
3	Theori	210.68	44.44%	58.12	110.34	4.97	53.57	34/70 (49%)
4	All You Need IS A Fuzzing Brain	153.7	53.77%	54.81	77.6	6.52	28.28	28/70 (40%)
5	Shellphish	135.89	94.83%	47.94	54.31	8.47	25.29	28/70 (40%)
6	42-b3yond-6ug	105.03	89.23%	70.37	14.22	9.8	10.97	41/70 (59%)
7	Lacrosse	9.59	42.86%	1.68	5.43	0.0	3.62	1/70 (1%)

VDS: Vulnerability Discovery Score, PRS: Program Repair Score, SAS: SARIF Assessment Score, BDL: Bundle Score. The Accuracy column indicates the percentage of submissions marked as correct by the scoring system.

Table 4: AIxCC Final Competition Results

		Budget Spending			LLM Usage		
Rank	Team	Azure	LLM	Total	Queries	Input Tokens	Output Tokens
1	Team Atlanta	$73.9K	$29.4K	$103.3K	696.5K	4.09B	641.6M
2	Trail of Bits	$18.5K	$21.1K	$39.6K	613.9K	12.83B	402.2M
3	Theori	$20.3K	$11.5K	$31.8K	187.6K	2.09B	112.5M
4	All You Need IS A Fuzzing Brain	$63.2K	$12.2K	$75.4K	122.9K	415.6M	85.4M
5	Shellphish	$54.9K	$2.9K	$57.8K	301.0K	4.69B	205.1M
6	42-b3yond-6ug	$38.7K	$1.1K	$39.8K	37.5K	96.7M	74.4M
7	Lacrosse	$7.1K	$0.7K	$7.8K	70.7K	246.4M	9.6M

Teams were allocated budgets of $85K for Azure compute resources and $50K for LLM usage in the final round. The varying resource utilization patterns reveal distinct strategic approaches to competition challenges.

Table 5: AIxCC Final Competition Resource Utilization.

2.4 Final Scores

The final competition round brought together seven teams for the ultimate test of their cyber reasoning systems. Each team deployed their developed CRS against 55 challenge projects from 28 repositories, competing for the highest score within strict time and budget constraints. The following results demonstrate the effectiveness of each team's automated vulnerability discovery and patching capabilities.

Table 4 presents the final competition results for all seven participating teams in the scored final round. Our team, Team Atlanta, achieved first place with a total score of 392.76 points, demonstrating superior performance across vulnerability discovery, program repair, and bundle submissions. The results reveal significant performance variations among teams: while several teams achieved high correctness rates for their submissions, we distinguished ourselves through both volume and quality, discovering 43 out of 70 total vulnerabilities (61% coverage) and excelling in successful patch generation and comprehensive bundle submissions that effectively combined PoVs, patches, and SARIF assessments.

Table 5 reveals the detailed resource utilization and LLM usage patterns across all teams during the final competition round. Notably, teams had allocated budgets of $85K for Azure compute resources and $50K for LLM usage in the final round. The data demonstrates clear correlations between resource investment and performance. Our team utilized the highest total budget ($103.3K) and generated the most LLM queries (696.5K), processing over 4 billion input tokens and generating 641.6 million output tokens. Interestingly, second-place team Trail of Bits demonstrated the highest input token usage (12.83B) with relatively efficient resource allocation. They relied on less powerful but cheaper LLMs (Claude Sonnet 4, GPT-4.1 mini, GPT-4.1) while we used more powerful LLMs (o4-mini, GPT-4o, o3) on average. In addition, we got more verbose outputs (*e.g.,* bug candidates and input generators) from LLMs. This choice may have helped surface richer insights during vulnerability analysis and patch generation, ultimately leading to stronger results.

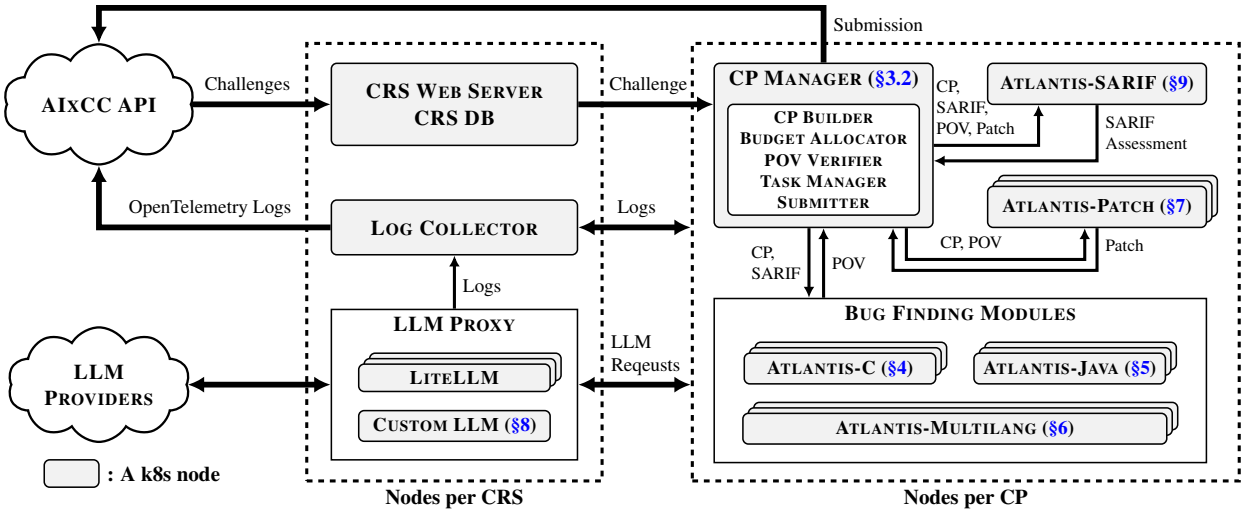


Figure 1: The overview of ATLANTIS.

3 ATLANTIS

The primary goal of the AIxCC competition is to build a cyber reasoning system (CRS) that employs large language models (LLMs) to automatically find proof-of-vulnerabilities (PoVs) in challenge projects (CPs), fix the vulnerabilities, and assess given SARIF reports. To this end, we designed and implemented our CRS, **ATLANTIS**, while achieving the following requirements:

R1: Support multiple CPs concurrently. In both real-world settings and the AIxCC competition, ATLANTIS should be able to process multiple CPs simultaneously. When multiple CPs are provided concurrently, ATLANTIS should scale effectively to automatically find PoVs and fix the corresponding vulnerabilities in all CPs, as well as assess given SARIF reports. Notably, during the AIxCC competition, CPs are delivered in multiple batches, meaning that ATLANTIS must handle overlapping sets of CPs while continuing to process newly received ones after previous deadlines have passed.

R2: Be fail-safe as much as possible. While processing multiple CPs concurrently, ATLANTIS must be designed with a fail-safe architecture to minimize the impact of individual task failures. For example, even if our CRS fails to handle a CP, it must continue to process the remaining CPs reliably without interruption.

R3: Fully utilize LLM and Azure budget. During the AIxCC competition, the organizers allocate fixed budgets for LLM usage and Azure cloud resources (see §2.3). ATLANTIS should maximize the utilization of these resources to achieve optimal performance while supporting multiple CPs across multiple batches, for each round.

R4: Other requirements. For observability, the AIxCC organizers required participants to collect and submit meaningful logs (*e.g.,* LLM requests and responses) in the OpenTelemetry format they specified. Furthermore, to ensure consistent deployment and easier infrastructure management, they mandated that our CRS be deployed on Azure with Terraform.

3.1 Overview

Figure 1 presents the overall architecture of ATLANTIS. ATLANTIS is deployed on a Kubernetes (k8s) cluster on Azure, provisioned and managed via Terraform, thereby fulfilling **R4**. To achieve effective scalability (**R1**) and enhance fault tolerance (**R2**), ATLANTIS adopts a two-tier node architecture within the k8s cluster: (1) CRS-level nodes, which host shared system components, and (2) CP-level nodes, each dedicated to processing a single CP.

ATLANTIS begins by launching CRS-level nodes for three key components: CRS-WEBSERVER, LOG COLLECTOR, and LLM PROXY. The LOG COLLECTOR is responsible for gathering and forwarding meaningful logs to the organizers (**R4**). The LLM PROXY, which is based on LiteLLM, centrally manages LLM usage to stay within the allocated budget (**R3**) and logs all LLM requests and responses for observability.

CRS-WEBSERVER listens for incoming CPs or SARIF reports from the AIxCC organizers. Upon receiving a new CP, CRS-WEBSERVER stores its metadata (e.g., repository and diff) in the database and spawns a dedicated CP-MANAGER instance on a CP-level k8s node.

Each CP-MANAGER instance is responsible for managing the analysis of a single CP. It first builds the given CP and allocates both the Azure compute budget and the LLM usage budget proportionally. Based on the allocated Azure budget and the number of fuzzing harnesses in the CP, CP-MANAGER launches an appropriate number of k8s nodes for bug-finding modules. For C-based CPs, it deploys ATLANTIS-C and ATLANTIS-Multilang; for Java-based CPs, it launches ATLANTIS-Java and ATLANTIS-Multilang. In addition, it starts ATLANTIS-Patch and ATLANTIS-SARIF to handle patch generation and SARIF assessment, respectively. To control LLM usage, CP-MANAGER issues LiteLLM API keys for each module, with budgets calculated based on the total number of CPs and the overall LLM budget. All modules launched under a given CP-MANAGER instance use the corresponding keys to ensure budget compliance and evenly distribute RPM and TPM of LLM models for each module. Notably, only ATLANTIS-Patch was able to utilize our custom LLM, which was fine-tuned for the patch generation.

When a bug-finding module discovers a PoV, it is sent to CP-MANAGER, which verifies whether the PoV triggers a crash and deduplicates results based on stack traces. If the PoV is confirmed to be unique, CP-MANAGER submits it to the AIxCC organizers and forwards it to ATLANTIS-Patch and ATLANTIS-SARIF for patch generation and SARIF assessment, respectively. Once the patch and SARIF report are produced, they are returned to CP-MANAGER. It then bundles them with the PoV and submits the complete bundle package to the AIxCC organizers. Although the rules permit submitting a patch or SARIF assessment without the corresponding PoV, we follow this integrated process to ensure that generated patches and assessments are sufficiently precise.

3.2 CP-MANAGER

In this subsection, we provide more details about several subcomponents of CP-MANAGER.

CP BUILDER. CP-MANAGER begins by compiling the given CP and making the compiled artifacts available to bug-finding modules via a shared file system within k8s. If the target CP is written in C, it is compiled using the default configuration as well as three additional configurations (see §6): one for ATLANTIS-Multilang, one for coverage build in OSS-Fuzz, and one for symbolic execution with SymCC. For Java-based CPs, CP-MANAGER compiles the project with both the default configuration and the configuration required by ATLANTIS-Multilang. The compiled outputs are then shared via the k8s shared file system, allowing the bug-finding modules to retrieve them and initiate the fuzzing process. Additionally, CP-MANAGER identifies all available fuzzing harnesses within the CP and finds the corresponding source code locations. This information is essential for both accurate budget allocation and identifying the entry points for LLM-based analyses. Note that this information is stored in the shared file system in JSON format.

BUDGET ALLOCATOR. CP-MANAGER is responsible for allocating both the LLM and Azure budgets for each CP. The allocation is based on the remaining total budget and the number of CPs yet to be processed. Once the total budget for a CP is calculated, CP-MANAGER distributes it among the corresponding modules.

For the LLM budget, a fixed amount of $500 is reserved for ATLANTIS-SARIF across all CPs, because the number of SARIF reports is predetermined by the competition. The remaining LLM budget is divided among the other modules as follows: 25% to ATLANTIS-Patch, and 37.5% each to ATLANTIS-Multilang and ATLANTIS-C (or ATLANTIS-Java).

For the Azure budget, we first allocate a fixed number of k8s nodes per CP. Three `Standard_D64ds_v6` nodes are allocated for CP-MANAGER, ATLANTIS-SARIF, and shared utilities of ATLANTIS-Multilang. Five `Standard_D32ds_v6` nodes are allocated to ATLANTIS-Patch, with each node serving one patch-generation agent. Next, based on the deadline of the CP, we compute how many cores can be utilized within the allocated Azure budget. This remaining compute capacity is equally divided between ATLANTIS-Multilang and ATLANTIS-C (or ATLANTIS-Java). For ATLANTIS-C, we allocate up to 15 nodes, regardless of the number of fuzzing harnesses, provided the budget allows. For ATLANTIS-Multilang and ATLANTIS-Java, we determine the available budget per fuzzing harness and select appropriate node types accordingly to launch fuzzing tasks. Finally, if any portion of the allocated budget remains unused, it is reallocated to future CPs to maximize overall resource efficiency.

POV VERIFIER. Once a bug-finding module identifies a PoV, it submits it to CP-MANAGER. CP-MANAGER then verifies whether the PoV reliably triggers a crash in an environment identical to that used by the AIxCC organizers. The verifier checks for specific crash return codes: sanitizer crashes (1 or 77), timeouts (70, which may indicate DoS vulnerabilities when the organizer's build configuration sets timeout limits), and out-of-memory conditions (71). For delta-mode CPs, CP-MANAGER additionally verifies that the PoV does not crash on the BASE version, ensuring the vulnerability exists only in the target changes. Rather than relying on the AIxCC-provided PoV deduplication mechanism based on textual similarity, we implement a custom deduplication strategy using stack trace analysis. Our approach incorporates multiple heuristics tailored to specific bug types to more accurately identify unique PoVs. This design choice is crucial because we request patch generation only for unique PoVs, with a rate limit of 10 patch requests per fuzzer-sanitizer combination to manage resource allocation effectively. Our improved PoV deduplication reduces LLM and CPU usage in ATLANTIS-Patch, enabling more efficient budget utilization.

TASK MANAGER. Another key responsibility of CP-MANAGER is orchestrating complex, multi-step workflows across modules. When CP-MANAGER receives a SARIF report from CRS-WEBSERVER, it forwards the report to ATLANTIS-SARIF to initiate SARIF assessment. Once ATLANTIS-SARIF completes its analysis, the result is returned to CP-MANAGER, which forwards it to the bug-finding modules to facilitate directed fuzzing. When a PoV is determined as unique, CP-MANAGER submits it to the AIxCC organizers and sends a patch generation request to ATLANTIS-Patch while polling for the organizers' verification results. If the PoV passes the organizers' verification, CP-MANAGER then initiates a PoV–SARIF mapping request to ATLANTIS-SARIF. This design maximizes throughput while respecting the rate limit constraints. After receiving the generated patch from ATLANTIS-Patch, CP-MANAGER verifies the patch in the AIxCC environment and, upon successful verification, sends the patch to ATLANTIS-SARIF to assist in final SARIF assessment. All inter-module communication for task management is implemented through web APIs exposed by each module, with state synchronization managed through Redis to ensure consistency across concurrent operations.

SUBMITTER. The primary role of CP-MANAGER is to submit PoVs, patches, and SARIF assessments to the AIxCC organizers. However, submitting multiple patches for the same underlying bug incurs a penalty. To avoid this, CP-MANAGER employs a sophisticated bundling strategy that dynamically manages PoV-patch-SARIF combinations. When new patches or SARIF assessments become available, the system evaluates existing bundles and either updates them or recreates them to ensure optimal point scoring. The bundling algorithm groups related PoVs under a single representative and manages patch selection to avoid redundant submissions. Additionally, the competition awards more points when PoVs, patches, and SARIF assessments are submitted as a bundled package. To maximize points, CP-MANAGER uses each PoV as a unique key and continuously attempts to match it with corresponding patches and SARIF assessments. A background process monitors unfinished components and automatically updates bundles when beneficial matches are found. This dynamic bundling approach ensures that PoVs are bundled with their corresponding patches and SARIF assessments whenever possible, maximizing competition points while avoiding penalties for duplicate submissions.

Target	Module	PoVs			Patches		Harnesses		Contribution
		Total	Passed	Dup.	Total	Passed	Affected	w/o dup	
C	ATLANTIS-Multilang	217	76	1	24	23	33	33	64.4%
	ATLANTIS-C	185	18	68	2	2	6	5	15.3%
Java	ATLANTIS-Java	424	15	73	14	9	14	10	12.7%
	ATLANTIS-Multilang	176	8	9	6	6	9	6	6.8%
	Unknown*	1	1	0	1	1	—		0.8%
	Total	**1,003**	**118**	**151**	**47**	**41**	**62**	**54**	**100.0%**

* Due to log truncation from unstable server environments, the originating module for this entry could not be determined.

Table 6: Performance breakdown by module in the final competition.

3.3 Final Competition Results by Module

As shown in Table 6, ATLANTIS achieved competitive performance in the AIxCC final competition, generating 1,003 PoVs with 118 passing verification, producing 47 patches with 87.2% success rate (41 passed), and correctly assessing 8 out of 10 valid SARIF reports. The system found PoVs across 62 harnesses, with 54 harnesses producing unique (non-duplicate) results, indicating that some vulnerable points can be reached through multiple harnesses. The system's 89.4% bundle submission success (42/47) demonstrates effective end-to-end integration across vulnerability discovery, patch generation, and assessment.

These results derive from OpenTelemetry logs provided by organizers, with important caveats: log truncation occurred due to unstable server environments; thus, data reflects only successfully logged operations. Additionally, PoVs that passed the competition environment's verification may have been identified as duplicates or failed manual assessment during post-competition analysis. Therefore, the results presented here differ from the final official scores in Table 4.

The results demonstrate how diversified AI-enhanced approaches can effectively address complex vulnerability discovery challenges. ATLANTIS-Multilang contributed the majority of verified PoVs through its multi-tier LLM integration across six specialized input generation modules, while specialized systems provided valuable complementarity: ATLANTIS-C's multi-fuzzer ensembles with LLM-augmented seed generation proved effective for coverage-guided scenarios, and ATLANTIS-Java's sink-centered analysis successfully identified security-sensitive API vulnerabilities.

The system's effectiveness emerged from complementary specialization where each component addressed distinct vulnerability patterns across diverse program characteristics (detailed analysis in Appendix A). The results suggest that comprehensive vulnerability discovery benefits from multiple coordinated approaches, with AI techniques applied where they provide clear advantages.

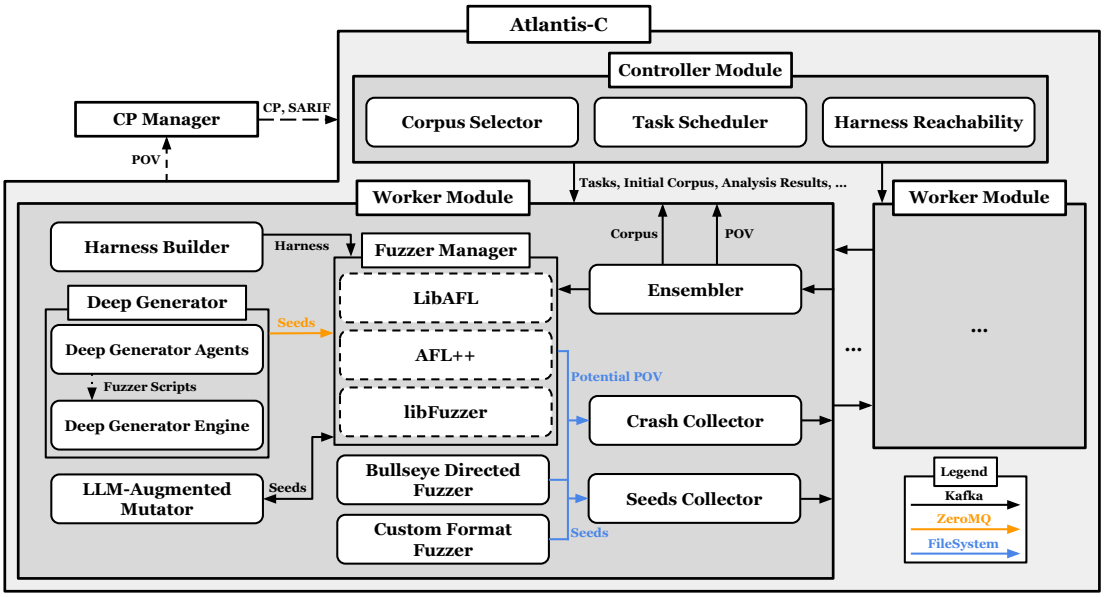


Figure 2: Overall Design of ATLANTIS-C

4 ATLANTIS-C

ATLANTIS-C targets vulnerability discovery in C and C++ challenge projects (CPs) by orchestrating a coordinated ensemble of fuzzers. Our experience shows that individual engines such as libAFL, AFL++, and libFuzzer respond differently to harness structures, input formats, and coding idioms found in CPs, leading to suboptimial performance when used alone [20]. To smooth out these gaps, ATLANTIS-C runs the engines side by side and wraps them with containerized services that can be attached to any engine to provide shared corpus management, feedback, and automation.

4.1 Overview

Design. ATLANTIS-C is an independent, multi-node, LLM-augmented fuzzing system. For robustness and flexibility, we design it with a Kafka-based microservice architecture. Figure 2 shows the high-level architecture of ATLANTIS-C, where each microservice is a separate containerized service. These services can be grouped into two modules: the CONTROLLER module and the WORKER module. When ATLANTIS-C is deployed onto multiple Kubernetes nodes, one of the nodes (the "controller" node) runs both a CONTROLLER module and a WORKER module, and the other nodes each run a WORKER module.

The CONTROLLER module contains microservices that are responsible for the overall task scheduling and interaction with other systems outside of ATLANTIS-C. It uses an epoch-based scheduling algorithm to schedule fuzzing tasks and orchestrate other modules accordingly. The WORKER module contains multiple microservices that are either part of the fuzzing pipeline or support it.

Workflow. For each challenge, the HARNESSBUILDER produces multiple instrumented builds per harness (libFuzzer, LibAFL, AFL++, directed, and coverage) and registers them as fuzzing tasks. Execution is orchestrated by the TASKSCHEDULER and the FUZZERMANAGER by assigning tasks to nodes and rebalancing priorities based on SARIF announcements, Open Source Vulnerabilities (OSV) metadata, reachability, crashes, and fuzzer health signals (see §4.3 and §4.3.2).

Seeds and crashes circulate through a central loop: the SEEDSCOLLECTOR and CRASHCOLLECTOR ingest inputs from LLM generators and fuzzing engines, and the ENSEMBLER deduplicates and merges seeds

using libFuzzer's merge mode, redistributes useful ones to active fuzzers, and triages scorables (see §4.4.4). LLM components, notably DEEPGENERATOR, generate high-quality seeds and on-the-fly fuzzing scripts tailored to harness formats or patch regions (see §4.5.2). Communication uses two planes: Kafka provides reliable control among microservices, while fuzzers embed ZeroMQ consumers that use shared-memory transport for high-rate seed injection (see §4.4.3). ATLANTIS-C integrates LibAFL, AFL++, libFuzzer, our directed fuzzer BULLSEYE (see §4.6), and a few project domain-specific fuzzers, all participating in the same seed/coverage loop.

In the following sections, we describe the features of ATLANTIS-C in detail, along with the design choices and implementation details for each related component.

4.2 Multi-Fuzzer Integration

ATLANTIS-C employs a multi-fuzzer ensemble approach that combines LibAFL, AFL++, and libFuzzer to maximize bug discovery across diverse target programs. Rather than selecting a single "best" fuzzer, we recognize that different engines excel in different scenarios, and their parallel execution with coordinated seed sharing can achieve better coverage than any individual fuzzer alone.

4.2.1 Instrumentation

ATLANTIS-C integrates a range of fuzzing engines, including LibAFL, AFL++, and libFuzzer [18, 19, 29], each with distinct requirements for build-time instrumentation. To accommodate these diverse needs, we developed the HARNESSBUILDER module, which orchestrates the building of fuzzing harnesses across multiple instrumentation modes.

Instrumentation multiplexing is implemented through two primary mechanisms:

1. Invoking the OSS-Fuzz `infra/helper.py` script with customized environment variables (*e.g.,* setting `CC` to a specific compiler or wrapper), and
2. Bypassing `infra/helper.py` entirely by executing Docker commands directly via a dedicated `compile` wrapper.

The second approach introduces the risk of inconsistencies or breakages, as it deviates from the officially supported OSS-Fuzz build infrastructure. Therefore, we apply additional validation and caution when handling artifacts generated via the direct Docker invocation method.

In total, ATLANTIS-C supports seven instrumentation modes:

1. LibFuzzer
2. LibAFL
3. Compiler Optimizations
4. Source-Based Code Coverage
5. AFL++
6. Directed Fuzzer
7. Artifact Extraction

These modes differ in their degree of integration with OSS-Fuzz. Mode 1 is essentially the standard harness build process and requires minimal deviation from it. Modes 2–4 are implemented by selectively overriding environment variables, and modes 5–7 involve deeper modifications to the build pipeline.

For instance, the artifact extraction mode applies post-processing to the generated harness binaries to recover the original source file paths. It also extracts auxiliary build artifacts, such as decompressed source directories (*e.g.,* from curl) or consolidated source files (*e.g.,* the monolithic amalgamation used by SQLite3).

Because OSS-Fuzz's `infra/helper.py` does not natively support the injection of custom build steps, the HARNESSBUILDER invokes Docker directly while attempting to replicate the environment that `helper.py` would normally establish. Although the artifact extraction process closely mirrors a standard libFuzzer build under OSS-Fuzz, we maintain it as a distinct mode to provide redundancy and improve fault isolation.

Supporting all seven instrumentation modes imposes a significant computational cost. Sequentially building each variant would result in unacceptable delays that bottleneck fuzzer startup. To mitigate this, ATLANTIS-C distributes build tasks across all of its available CRS nodes by launching a separate HARNESSBUILDER instance per node. Given the allocated computation budget, this allows all instrumentation modes to be built in parallel, ensuring timely fuzzer initialization without compromising throughput.

4.2.2 Fuzzer Fallback

ATLANTIS-C primarily relies on a custom fuzzer built with LibAFL for high-performance fuzzing. However, we experienced fragility issues with this fuzzer, which could sometimes crash during instrumentation or fuzzing. To mitigate this and prevent a complete failure of our system, we integrated multiple fuzzing engines with different performance and stability trade-offs. AFL++ is more stable than our LibAFL fuzzer, but could still fail due to our modifications to make it run in parallel, and because the CPs were only designed and tested with libFuzzer. LibFuzzer is the least performant of our engines, but is the most stable as it is the default fuzzer for OSS-Fuzz.

The interface for running all fuzzers is unified in ATLANTIS-C's fuzzer manager. We wrap all fuzzer engines with an interface that supports starting the fuzzer, stopping the fuzzer, monitoring for crashing test cases, monitoring for when the fuzzer successfully starts testing, monitoring for execution errors, and monitoring logs to extract coverage, execution, and crash statistics. By collecting such metrics from each type of fuzzer, we can determine when to abort the execution of a fuzzing engine and fall back to another one. For instance, if LibAFL aborts more than 10 times in a single epoch, we consider it to be too fragile, and proceed by killing the fuzzing container and restarting it with AFL++.

The fuzzer not initializing is another error condition that needs to be considered. If we had used traditional space-based partitioning, we could assign each fuzzing engine a long initial timeout, and if the fuzzer did not report healthy within the threshold, we would abort and fall back. In ATLANTIS-C's time-based harness partitioning (§4.3), however, setting a long initial timeout is not possible due to the short epoch period. Initially we tried using hard-coded short timeouts, but we encountered cases where harness initialization would have succeeded but was too slow for our timeouts and was misidentified as a failure. Instead, we decided to simply give the fuzzer the entire epoch to initialize, and then handle fallback, if necessary, during the next epoch.

4.3 Time-based Task Scheduling

An important design choice for ATLANTIS-C is the use of time-based partitioning to schedule fuzzing tasks—that is, instead of executing all fuzzing tasks in parallel, ATLANTIS-C executes them based on time slices, which we refer to as "epochs". Fuzzing tasks correspond to different harness binaries built by the HARNESSBUILDER that are used for general fuzzing. This means that for each given harness, the three instrumentation modes libFuzzer, LibAFL, and AFL++ each define a fuzzing task.

The main reason we use time-based scheduling is for the flexibility to respond to events that occur during the competition time, such as SARIF report announcements, SARIF report analysis results, diff reachability analysis results, and fuzzer fallback-triggering events. Specific details of these events and how they are handled will be discussed in the following sections.

Another reason for choosing time-based scheduling is that ATLANTIS-C's instrumentation modes are completely separate from those of the ATLANTIS-level harness builder. This means that, at deployment time, ATLANTIS-C will not know the exact number of harnesses or fuzzing tasks for the challenge project it is to analyze. Since the number of nodes is determined at deployment time, we need a scheduling strategy able to deal with the case where there are more fuzzing tasks than nodes allocated to ATLANTIS-C.

4.3.1 Implementation

The implementation of time-based scheduling is done through the interaction between the TASKSCHEDULER, FUZZERMANAGER, and other fuzzer-related modules like SEEDSCOLLECTOR, CRASHCOLLECTOR, DEEPGENERATOR, and LLMAUGMENTEDMUTATOR. Every epoch is normally 20 minutes long, but the TASKSCHEDULER can also end an epoch early for a timely response to the events mentioned in §4.3.

The TASKSCHEDULER module manages a task queue, and an integer-value priority weight for each task. At the start of each epoch, it pops a number of fuzzing tasks from the task queue equal to the number of nodes allocated to ATLANTIS-C. These fuzzing tasks are then distributed to each node by publishing two rounds of Kafka messages. The first round of messages stops the currently running fuzzing tasks that are not also in the set of just-popped tasks. These messages instruct the FUZZERMANAGER containers to exit, and Kubernetes's restart policy will then kick in to restart these containers in a clean state. The second round of messages is sent to these clean FUZZERMANAGER containers to start the new fuzzing tasks.

Once the task queue is empty, the controller will repopulate it with weighted random sampling from the list of fuzzing tasks. By using this repopulation strategy, the controller can prioritize certain tasks and ensure that all fuzzing tasks with non-zero priority are eventually scheduled. To minimize the overhead of switching epochs, we implemented a basic optimization allowing an epoch switch to be skipped if all of the following heuristic conditions are met:

- **No Starvation:** there is no non-zero priority fuzzing task that has not been scheduled.
- **No Priority Weight Update:** no priority weight has been updated since the last epoch.
- **Priority Weights Already Respected:** the number of times each fuzzing task has been scheduled is already rankwise equal to its priority weight.

Whenever a FUZZERMANAGER stops and restarts with a new fuzzing task, it will publish a Kafka message that is subscribed to by the TASKSCHEDULER and other fuzzer-related modules. The TASKSCHEDULER uses this information to track the current state of ATLANTIS-C and respond to failures in certain fuzzing tasks. An example of such a failure would be if a harness instrumented by LibAFL is scheduled as a new fuzzing task, but its execution speed is extremely slow or it repeatedly crashes abnormally. This information will be included in the message, and allows the TASKSCHEDULER to trigger a fuzzer fallback event, which will stop that specific fuzzing task, set its priority weight to zero, and replace it with a new one using AFL++ instead of LibAFL. The same flow can also apply to an AFL++ harness, resulting in a fallback to libFuzzer. This fallback mechanism allows us to deal with all types of instrumentation failures and greatly enhances the robustness of ATLANTIS-C.

Other fuzzer-related modules also use this information to modify their behavior. For example, the SEEDSCOLLECTOR uses it to determine which path it should collect seeds from and what harness the seeds belong to. This is important because different fuzzing engines have different conventions for how to store seeds in the file system.

4.3.2 Harness Deprioritization

ATLANTIS-C consists of multiple components that collect, mutate, triage and distribute important seeds and results for all active fuzzing tasks. To utilize our computational resources efficiently, we need a smart approach for identifying which tasks—in particular, which harnesses—should be run with higher priority. More specifically, for delta-mode challenges, we want to prioritize harnesses with execution paths that can actually reach the code affected by the delta, and effectively ignore ("deprioritize") all other harnesses.

While a straightforward solution would to be use precise static analysis to filter "unreachable" harnesses before fuzzing begins, a purely sequential approach (static analysis first, then fuzzing) has both performance and accuracy limitations. First, precise static analysis is computationally expensive, with runtime varying across programs. In our experiments on (the organizer-provided version of) SQLite3, it required several hours to complete analysis using CodeQL. Second, static analysis tools can produce both false positives and false negatives. While false positives would be acceptable, false negatives are a critical issue, as misclassifying a reachable harness as unreachable would completely prevent the opportunity to discover scorable bugs. In fact, during our testing, we observed such false negatives in the curl project.

To ensure correct and efficient deprioritization, we implement a hybrid approach: lightweight analysis is done before fuzzing starts, and heavier analysis is done in parallel during fuzzing. Thanks to our time-based scheduling strategy, deprioritization based on the results of the heavier analysis can be applied dynamically soon after those results become available. Our approach utilizes three different sources of reachability information:

1. Compilation-time analysis
2. BULLSEYE's instrumentation results
3. Analysis results from ATLANTIS-SARIF

The HARNESSBUILDER features a build mode that applies aggressive compilation and linking optimization flags, triggering dead code elimination passes in the compiler. We use this to create a simple oracle to determine reachability. In delta mode, we compile all fuzzing harnesses with these aggressive flags both before and after the provided patch is applied to the codebase. If the two binaries (pre- and post-patch) for a given harness are identical, it indicates that all of the code modified by the patch was eliminated as dead by the compiler, implying that the patched code must be unreachable by that harness. ATLANTIS-C disables all harnesses deemed irrelevant by this method, allowing compute to be allocated to more relevant harnesses.

The other two sources of reachability information take a code location and harness as input, and determine whether the harness can reach the location. To select these locations, we use an LLM agent that analyzes the diff and selects a few locations with corresponding priority values. When passing these locations to reachability analysis, there may be conflicting results for certain harnesses, where some locations are reported as reachable and others as unreachable. If too many such conflicting cases exist, it could lead to a situation where *all* harnesses become disabled and analysis cannot continue. Prioritizing certain locations over others is key to resolving these conflicts and avoiding that corner case.

ATLANTIS-C also leverages BULLSEYE's instrumentation to provide reachability results. After receiving target locations from either a SARIF report or the delta mode location-choosing agent, ATLANTIS-C dispatches a directed fuzzing task, which has two stages: instrumenting and running. If the harness instruments without error on the target location, the directed fuzzer starts running. If, however, a "target not found" error is reported by the directed fuzzer, it means that the static analysis based on SVF determined that the harness cannot reach the specified location. If some harnesses report a particular location as reachable and others report it unreachable, we disable the latter set of harnesses. Otherwise (*i.e.,* either all harnesses or no harnesses report a location to be reachable), no action is taken. As results for more locations become available, we combine them, and stop fuzzing harnesses confirmed to be unreachable in later epochs.

After starting the fuzzing process, our primary source of reachability results comes from ATLANTIS-SARIF's callgraph analysis, detailed in §9. This hybrid approach begins with static analysis, then monitors program execution to detect new call relationships at runtime. When new relationships are discovered, they are added to existing results and the entire callgraph is recalculated. ATLANTIS-SARIF's callgraph artifacts are kept in a shared directory that ATLANTIS-C actively monitors for real-time harness deprioritization. ATLANTIS-C uses `libSARIF`, detailed in §9.1, combined with our chosen locations to extract an aggregate result of disabled harnesses.

This real-time, adaptive deprioritization strategy balances computational efficiency with precision, ensuring that we optimize resource allocation without sacrificing the ability to uncover critical bugs.

4.4 Corpus Management

Since ATLANTIS-C is essentially a distributed fuzzing system, an important engineering task is to manage the corpora, or fuzzing seeds, across multiple fuzzing campaigns. Notably, ATLANTIS-C runs multiple fuzzing instances at the same time across a number of Kubernetes nodes, and may also fuzz the same harnesses again in future epochs. Therefore, we need a scalable way to accumulate the progress of each campaign across both space and time.

4.4.1 Initial Corpus

To enhance fuzzing performance, ATLANTIS-C emphasizes the importance of a high-quality and diverse initial corpus. A well-curated set of seed inputs significantly accelerates coverage discovery and reduces time-to-crash by priming each fuzzer with inputs relevant to its target. Recognizing that generic or randomly generated inputs are often insufficient for complex real-world targets, ATLANTIS-C incorporates a large and semantically rich corpus from a variety of sources.

Collection. We manually collected seed inputs and test cases from over 400 open-source projects, organizing the resulting dataset into 90 semantic categories. These categories group inputs by structure or domain, such as SQL queries, JavaScript programs, and HTML documents. This categorization enables better alignment between the fuzzing strategy and the expected input domain of a given target. No crashing seeds from any AFC round were included in our collection. Seed sources include:

- Existing corpus directories from OSS-Fuzz projects
- Test cases from OSS-Fuzz issue trackers
- Test cases from project-specific issue trackers
- Sample and test data found in project repositories

Runtime Selection. The runtime initial corpus selection process is implemented in the CORPUSSELECTOR component, which is run whenever a new challenge project is received, via two LLM-assisted phases:

1. **Harness Analysis Phase.** An LLM agent inspects the structure of the fuzzing harness—such as entry-point functions, data types, and surrounding context—to understand the expected input format and usage patterns.
2. **Category Matching Phase.** Based on the extracted information, a simple prompt is issued to the LLM to suggest the most relevant seed categories from the predefined set. This phase does not involve complex reasoning, and serves as a lightweight query to guide selection.

For each matched category, ATLANTIS-C extracts the corresponding seed archive and loads it into the fuzzers as the initial input corpus.

While supplying large corpora may incur a short-term startup overhead, this does not affect overall system correctness, provided the inputs are consumed within the active scheduling epoch. After this initial corpus processing, only interesting seeds filtered by the fuzzer will persist across epochs for the same harness (see §4.4.2). The early performance cost is offset by the long-term gain in fuzzer effectiveness and faster convergence during exploration.

4.4.2 Corpus Seed Lifecycle

Essentially, all corpus seeds that ATLANTIS-C produces should be consumed by the fuzzers and contribute to their coverage. The main sources of seeds are:

- The rich initial corpus (see §4.4)
- Seeds generated by LLM modules (see §4.5.2, §4.5.3)
- Corpora from the directed fuzzer (see §4.6) and project domain-specific fuzzers
- Corpora from previously or currently running fuzzing tasks
- Seeds shared from ATLANTIS-Multilang and ATLANTIS-SARIF

We must also consider that the harnesses that consume the seeds could be instrumented with any of LibAFL, AFL++, or libFuzzer. All of these cases were covered using three different methods.

The first method is sending the seeds via Kafka through the ENSEMBLER. The SEEDSCOLLECTOR is responsible for watching the seed directories for the directed fuzzers, project domain-specific fuzzers, and seeds shared from ATLANTIS-Multilang and ATLANTIS-SARIF. When a new seed is added to any of these directories, the SEEDSCOLLECTOR will send it to the ENSEMBLER through Kafka. These seeds will be deduplicated and sent to the fuzzers to be consumed (see §4.4.4). This additional step prevents redundant seeds from wasting the fuzzers' computational resources. Our LibAFL instrumentation includes a Kafka consumer, so this method is applicable when a LibAFL harness is running.

The second method is direct filesystem dumps. The SEEDSCOLLECTOR periodically compresses the corpora of each currently running fuzzing task and stores them in a shared directory. The most recent version of the compressed corpora for a given harness is essentially the deduplicated, accumulated set of seeds from all sources for that harness up to that point in time. Therefore, by loading this compressed corpus directly into the fuzzer's corpus directory during fuzzing task initialization (or during runtime, in the case of libFuzzer), we can effectively utilize all seeds we have collected so far.

The third method is using ZeroMQ to directly feed the seeds into the fuzzers. This method is especially useful for modules that generate seeds at a very high rate, such as DEEPGENERATOR. This method is described in more detail in §4.4.3.

4.4.3 ZeroMQ

Although we mostly rely on Kafka for message-passing between microservices, we use ZeroMQ for low-latency communication between fuzzers and other components such as agents and DEEPGENERATOR's dynamically generated fuzzer scripts.

ZeroMQ Fuzzer Consumer. We need a way to send seeds to fuzzers while they are running. To this end, we design and implement ZeroMQ consumers as a special mutation stage in LibAFL, AFL++, and libFuzzer. Specifically, we launch a background thread in each fuzzer to receive external seeds and queue them in an

internal buffer. When triggering mutation, the fuzzers first check the internal buffer for any pending external seeds, and consume them if present.

4.4.4 Ensembler

The role of the ENSEMBLER is to test seeds produced by other parts of the system, such as fuzzers, LLM-based mutators, and ATLANTIS-Multilang. Depending on the results, seeds can be submitted to CP-MANAGER's PoV Verifier (see §3.2) for scoring, sent to fuzzers and ATLANTIS-Multilang to improve their code coverage, or discarded.

libFuzzer, which is available by default for all OSS-Fuzz projects, features a "merge" mode in which, instead of performing fuzzing, it updates a corpus directory by adding seeds from other provided directories. More specifically, it executes each seed in a set of specified folders, measures their code coverage, and copies any seeds that reach code not reachable by the seeds in the first folder to that folder. The ENSEMBLER uses this libFuzzer feature to continually update its "master corpus", a set of seeds intended to approximately represent all code coverage that has been achieved by the CRS so far. Whenever a new seed is added to the ENSEMBLER's master corpus directory, it sends the seed to the fuzzers to be incorporated into their corpora, as well as to ATLANTIS-Multilang to improve its coverage. It also monitors libFuzzer's console output to watch for crashes. If any crashes are detected, basic deduplication is performed, and the responsible seeds are submitted to CP-MANAGER for scoring.

Parallel merging. To handle the required throughput, the ENSEMBLER continually runs multiple instances of libFuzzer in parallel, from a pool of worker processes that all draw seeds from a shared internal queue. If all of the libFuzzer processes were to update the same master corpus directory directly, they could potentially interfere with each other, possibly leading to loss of seeds or other issues.

To prevent such problems, the ENSEMBLER prepares a "snapshot" of the master corpus directory before each libFuzzer call. The snapshot will not be modified while libFuzzer is running, even if other libFuzzer instances add new seeds to the master corpus directory during that time. For efficiency, snapshot directories consist of symbolic links to seeds in the master corpus directory rather than full file copies. After the libFuzzer process exits, the ENSEMBLER moves any seed files that were added to the snapshot directory to the master corpus directory.

Timeouts. Timeout bugs are in-scope for AIxCC. Specifically, inputs that take longer to execute than a certain timeout duration (25 seconds by default, but configurable by the competition organizers per harness) can be submitted as PoVs for points. Thus, the ENSEMBLER must be able to run seeds for at least that long to determine whether a seed is scorable in this way. However, to maintain high throughput, it must also avoid being slowed down too much by any seed or batch of seeds.

To manage these conflicting goals, the ENSEMBLER configures the libFuzzer merge invocations with a short per-seed timeout of 1 second. Any seeds that reach this timeout are placed in a "slow-seeds queue" for additional processing later. Furthermore, to guard against large batches of slow-running seeds, a per-*batch* timeout is also used, equal to the number of seeds in the batch times 0.5 seconds, with a minimum of 10 seconds. If a batch times out in this way, its remaining seeds are discarded without being tested at all, on the assumption that they are unlikely to be able to contribute any useful coverage or crashes, and are not worth spending more time on.

The ENSEMBLER executes seeds drawn from the slow-seeds queue whenever no other internal jobs are available from its main queue. If one of them reaches the scorable timeout duration, it is submitted to CP-MANAGER.

4.5 LLM Components

4.5.1 Agent Library: LIBAGENTS

To facilitate agent development, we built a research-oriented library, LIBAGENTS. This library contains a finite-state machine to split a query into sub-queries, external plugins and LLM tools, and an answer evaluator to assess answer quality. Users enable the desired plugins and provide a query; the library then iterates over sub-queries and calls external plugins to obtain and evaluate answers. Note that this library is only used by some CRS components.

Finite-State Machine. When using the LIBAGENTS API, users first specify the plugins to enable and the query to be solved. LIBAGENTS selects evaluation metrics to use for the query (*e.g.,* completeness, plurality), then asks the LLM to answer the question. When an answer is received, the evaluator assesses its quality based on the metrics. If it passes, the query is solved; otherwise, LIBAGENTS asks the LLM to reflect and generate new sub-queries. It then iterates over the new sub-queries and adds the responses to the context. Once either the original query is solved or the token limit is reached, the loop is exited and the answer is returned. Note that, to save tokens, evaluation metrics are not used for sub-queries, only for the main query.

Plugins. To help LLMs understand external context, we provide plugins for accessing external sources. We offer `read`, `ls`, `grep`, and `sed` for file operations, `code-browser` for code browsing, and `aider` for AI-assisted programming.

Answer Evaluator. Based on the metrics selected for a query, the evaluator assesses answer quality. In particular, we use `definitive` to ensure an answer is unambiguous, `plurality` to ensure it contains sufficient information, and `completeness` to ensure coverage.

4.5.2 Agent: DEEPGENERATOR

DEEPGENERATOR is an LLM-based agent that can analyze a CP and generate fuzzer scripts on the fly. It contains multiple LLM agents—currently, `harness-analysis`, `delta-scan-analysis`, and `self-evolving`—and an engine for scheduling fuzzer-script and testcase generation.

Main Engine. Upon receiving a new CP, the engine calls the LLM agents to generate fuzzer scripts, which are Python scripts that produce testcases. The engine watches for new scripts, and runs all of them in a loop to produce testcases continuously. All testcases are sent to the fuzzers via ZeroMQ.

Harness Analysis. The `harness-analysis` agent analyzes a fuzzing harness, walks through the referenced source code, and infers the input format. It then generates fuzzer scripts based on the collected information.

Delta-Scan Analysis. For delta-scan challenges, the `delta-scan-analysis` agent analyzes the specific commit or pull request and generates a fuzzer script focused on the relevant changes. Specifically, it first analyzes the diff and generates a list of suspicious locations, then generates a fuzzer script targeting them.

Self-Evolving Agent. The previous agents cannot always generate high-quality fuzzer scripts. To address this, we introduce the `self-evolving` agent, which analyzes fuzzer scripts produced by other agents and generates new ones. It uses predefined metrics (*e.g.,* coverage, error rate) to evaluate script quality, then asks the LLM to propose new fuzzing strategies to improve the scripts.

DEEPGENERATOR is also used in ATLANTIS-Java (see §5.4).

4.5.3 Agent: LLM-Augmented Mutator

Fuzzers often get stuck when exploring deep execution paths or encountering complex program logic. As a result, code coverage stagnates and fuzzing becomes ineffective. This can result from several factors: lack of semantic awareness of input structure, dependencies on specific input data regions to pass conditional checks, or intricate logic that standard mutation strategies fail to navigate effectively.

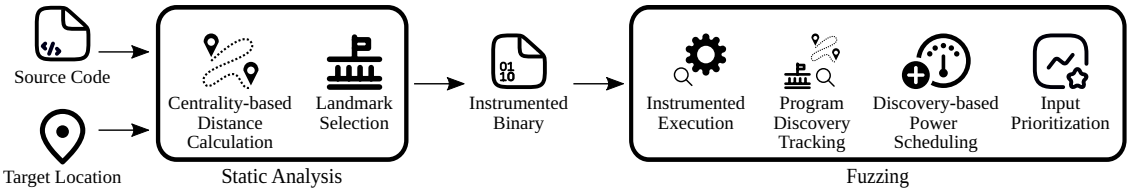


Figure 3: BULLSEYE architecture. During static analysis, landmark selection and distance calculation are performed, and the binary is instrumented with this information. During fuzzing, landmarks are used to calculate discovery, while the power scheduler leverages distance and discovery metrics to allocate seed energy. Additionally, a queue of favored seeds is maintained, prioritizing those with higher landmark hits and better distance scores.

As a one-stop solution, we deployed an LLM-augmented mutator as a microservice within our fuzzing system. The primary objective is to provide fuzzers with new mutation opportunities for stuck seeds by leveraging program semantic understanding. The service continuously monitors seed storage directories during fuzzing, tracking when new seeds are added to detect potential stagnation points. When no new seeds appear in a directory for two minutes, we identify the fuzzer as potentially stuck, and initiate LLM-assisted mutation on a recently added seed.

The service gathers comprehensive execution data for stuck seeds, including complete call stack information, source-code-level execution traces, and variable values at function entry and exit points for the deepest executed function. This information, along with current coverage data, is sent to the LLM with appropriate prompts. We find that this service is able to generate input variations that trigger new branches.

4.6 Directed Fuzzing: BULLSEYE

Motivation. Directed graybox fuzzers (DGFs) typically focus on reaching a target location as quickly as possible by pruning away paths deemed irrelevant. While effective in reducing search space, this strategy often leads to getting stuck in local minima, missing alternative routes that could expose deeper or more subtle bugs. To address this limitation, we built BULLSEYE, an in-house DGF that still prioritizes proximity to the target but is explicitly designed to explore multiple diverse paths toward it. This increases the likelihood of triggering the target under different program conditions. This design is especially valuable in the context of the AIxCC competition, as it enhances the chance of discovering multiple bug variants, a graded objective.

Target selection. For full-scan challenges, the SARIF-reported location is used as the fuzzing target. In delta-scan mode, an LLM agent is used to select one representative line per patch hunk, focusing on additions likely to contain new logic. These lines are passed to BULLSEYE as fuzzing targets.

Overview. BULLSEYE combines static analysis with runtime guidance to improve target-oriented fuzzing. As shown in Figure 3, it first performs static analysis to compute a distance score for each basic block using closeness centrality over the interprocedural control flow graph (ICFG), prioritizing blocks that are structurally close to the target. It also selects a fixed set of landmark locations across the program, which are instrumented and tracked at runtime. The number of triggered landmarks is used to compute a *discovery* metric, quantifying how much of the program space relevant to the target has been explored. During fuzzing, BULLSEYE uses these two metrics in its power scheduler to control how much energy is assigned to each seed, and in its input prioritization to favor seeds that reach more landmarks and are closer to the target in the program structure.

Implementation. We implemented BULLSEYE with over 1,500 lines of code across two main components. The static analysis module, written in C++ as an LLVM pass, performs inter-procedural distance computation and landmark selection. The fuzzing module is built on top of AFL++ with approximately 500 lines of changes to integrate BULLSEYE 's power scheduling and input prioritization. To maximize performance, we enabled AFL++'s persistent mode and shared memory fuzzing, reducing process overhead and significantly

No.	Project	Harness	Challenge Type	BULLSEYE			AFL++		
				Runs	TTE (s)	UC	Runs	TTE (s)	UC
1	asc-nginx	pov_harness	sarif	5	**39**	**94.2**	5	326.6	90.4
2				5	**100**	**35.8**	5	314.2	13.2
3				5	836.4	**8.4**	5	**398.2**	7.6
4	file	magic_fuzzer_fd	sarif	5	541	**32.8**	5	**533.2**	23.4
5	libcue	fuzz	sarif	5	**121**	8.4	5	194.4	**11**
6	libtiff	tiffcp	sarif	**4**	**11665.25**	**1**	0	–	–
7				5	**6.8**	436.6	5	10	**526**
8	libxml2	html	delta	5	96.8	**66.4**	5	**40.8**	60.8
9	libexif	exif_from_data_fuzzer exif_loader_fuzzer	delta	5	**62.6**	**5.8**	5	108.8	4.2
10				5	481	**27**	5	**386.2**	17
11				3	**4010**	**1**	3	10804.67	0.8

Table 7: Fuzzing performance across AIxCC challenges using BULLSEYE and AFL++. Metrics shown include the number of independent fuzzing runs, TTE in seconds, and UC.

improving execution throughput. To our knowledge, BULLSEYE is the first DGF to incorporate these advanced optimizations, making it well-suited for real-world deployment.

Impact. We evaluated BULLSEYE on 11 real-world targets provided by the competition organizers during preparation rounds. Each target was fuzzed for 9 hours across 5 independent runs, and we report the average time to exposure (TTE) and unique bug crashes (UC) in Table 7. BULLSEYE outperformed AFL++ in 7 out of 11 targets in terms of TTE and in 9 out of 11 in terms of UC. Notably, on target #6, BULLSEYE triggered the crash in 4 out of 5 runs, while AFL++ failed in all runs.

5 ATLANTIS-Java

ATLANTIS-Java is a subsystem of ATLANTIS that focuses on Java CPV detection in the competition. It is designed based on the observation that many Java vulnerabilities are sink-centered security issues that arise from unsafe usage of sensitive APIs. Rather than replacing traditional coverage-based fuzzing, ATLANTIS-Java augments it with sink-aware techniques to address the unique challenges of Java vulnerability detection.

Jenkins CPV Code Snippet I

```
1 public void doexecCommandUtils(...) {
2
3   byte[] sha256 = DigestUtils.sha256("breakin the law");
4
5   if (containsHeader(request.getHeaderNames(), "x-evil-backdoor")) { // condition 1
6     String backdoorValue = request.getHeader("x-evil-backdoor");
7     byte[] providedHash = DigestUtils.sha256(backdoorValue);
8     if (MessageDigest.isEqual(sha256, providedHash)) { // condition 2
9       String res_match = createUtils(cmdSeq2);
10      ...
```

Jenkins CPV Code Snippet II

```
11 String createUtils(String cmd) throws BadCommandException {
12   if (cmd == null || cmd.trim().isEmpty()) { // condition 3
13     throw new BadCommandException("Invalid command line");
14   }
15
16   String[] cmds = {cmd};
17
18   try {
19     ProcessBuilder processBuilder;
20     processBuilder = new ProcessBuilder(cmds); // The sinkpoint
21     Process process = null;
22     try {
23       process = processBuilder.start(); // cmds[0] == 'jazze' → OS Command Injection
24       ...
25     } catch (IOException e) {
26       // Handle exception
27     }
28   } catch (Exception e) {
29     // Handle exception
30   }
31 }
```

Figure 4: Example CPV from AIxCC Semifinal Jenkins CP

5.1 Motivation Example

Our core observation is that many Java vulnerabilities stem from the unsafe usage of sink APIs, and their detection process can be modeled as a sink-centered exploration and exploitation workflow. Figure 4 illustrates a Jenkins CPV that demonstrates this pattern. The vulnerability contains a backdoor that enables OS command injection when specific conditions are met. At line 20, the `ProcessBuilder` constructor serves as a sink API, a security-sensitive operation where attacker-controllable arguments can lead to command execution.

From a sink-centered perspective, detecting this vulnerability involves two distinct phases:

- **Sink exploration** (lines 3-8, 12-19): The fuzzer must satisfy path constraints to reach the sinkpoint, including the presence of header `x-evil-backdoor` with a value matching the SHA-256 hash of `"breakin the law"`, and non-empty command validation.

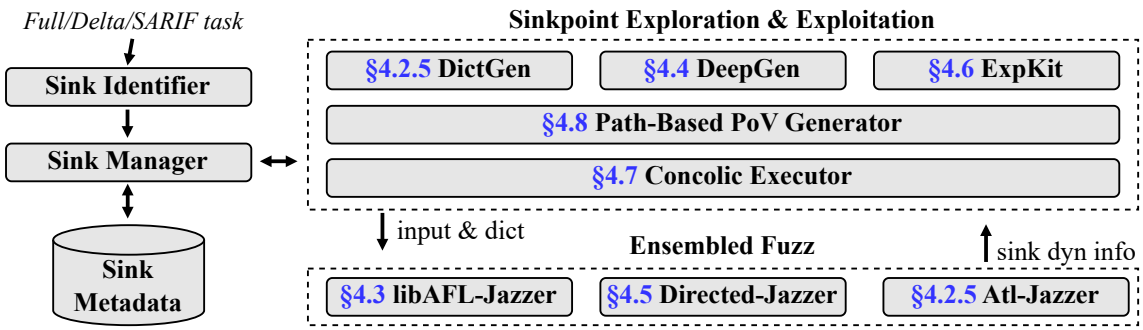


Figure 5: Overview of ATLANTIS-Java

- **Sink exploitation** (lines 20-23): Once the sinkpoint is reached, the fuzzer must generate inputs that trigger the actual vulnerability—in this case, setting `cmds[0]` to `"jazze"` to satisfy the Jazzer detection oracle.

This two-phase pattern applies broadly to Java vulnerabilities. Security issues typically manifest through dangerous API calls such as file operations, deserialization, command execution, network access, and template rendering. Each vulnerability type presents unique challenges beyond simply reaching the sink. For instance, path traversal requires constructing malicious paths that bypass sanitization; deserialization demands crafting objects that satisfy type constraints while achieving code execution; and SSRF needs URLs that bypass validation to reach external resources.

However, existing Java fuzzing solutions, which are mostly inherited from C/C++ fuzzers, are primarily coverage-centered and leverage only limited sink knowledge. They treat all code paths equally, missing opportunities to prioritize security-critical operations or generate exploitation-aware inputs. This motivates ATLANTIS-Java's design as a sink-centered framework that makes sinks first-class citizens in the fuzzing process, by combining static analysis, dynamic testing, and LLM capabilities to enhance both exploration and exploitation phases.

5.2 System Overview

Figure 5 illustrates the overall design of ATLANTIS-Java, which implements a sinkpoint-centered workflow for Java vulnerability detection. At its foundation, ATLANTIS-Java maintains an ensemble fuzzing pipeline that serves as the base infrastructure and provides continuous input generation and execution capabilities. Built upon this pipeline, the system performs sink analysis on target CPs to identify relevant sinkpoints through program analysis.

Once identified, each sinkpoint undergoes two complementary phases: *sinkpoint exploration* to discover execution paths that reach the sink, and *sinkpoint exploitation* to generate inputs that trigger vulnerabilities at the reached sink. Throughout this process, sink-aware components generate specialized inputs and dictionaries that feed back into the ensemble fuzzing pipeline, while the pipeline provides dynamic execution feedback to refine sink analysis. The system collects and maintains "beep seeds", inputs that successfully reach sinkpoints, which serve as valuable starting points for subsequent exploitation attempts.

A centralized management component tracks sinkpoint metadata, including call graphs, sink status, corpus, and crashes, and dynamically schedules system resources. Already-reached sinks skip exploration for direct exploitation, while unexploitable sinks are marked and excluded from further processing. This bidirectional interaction between the ensemble fuzzing infrastructure and sink-centered components enables ATLANTIS-Java to effectively leverage both coverage-based and sink-aware techniques for vulnerability detection.

Algorithm 1: Sinkpoint-Aware Fuzzing Loop

```
   Input: 𝒫: Target program
          𝒦: Set of sinkpoints
1  𝒮 ← {initial seeds}
2  ℬ ← ∅                                                    // Init beep seeds set
3  while not timeout do
4      s ← PickSeed(𝒮)
5      s′ ← Mutate(s)
6      result ← Execute(𝒫, s′)
7      if HasNewFeedback(result) then
8          𝒮 ← 𝒮 ∪ {s′}
9      if ReachSinkpoint(result, 𝒦) then
10         ℬ ← ℬ ∪ {s′}                                     // Add to beep seeds
11         ExploitationPhase(s′)
12     if IsCrash(result) then
13         SaveCrash(s′)
```

To align with the AIxCC competition format, ATLANTIS-Java transforms all three task types (Full Mode, Diff Mode, and SARIF tasks) into concrete sinkpoint lists for processing: ① full mode tasks target all sinkpoints within the CP source tree; ② diff mode tasks focus on sinkpoints affected by code changes; and ③ SARIF tasks prioritize sinkpoints specified in the vulnerability reports. This transformation allows ATLANTIS-Java to use one unified sinkpoint-centered technique for handling all tasks.

To support various exploration and exploitation techniques within this workflow, ATLANTIS-Java provides essential infrastructure functionalities. These components manage sinkpoint information, coordinate fuzzing efforts, and optimize resource utilization. The following paragraphs detail each infrastructure component that enables our sinkpoint-centered approach.

5.2.1 Infrastructure I - Sinkpoint-Aware Fuzzing Loop

Algorithm 1 presents our enhanced fuzzing loop that extends traditional coverage-guided fuzzing with sinkpoint awareness. Beyond tracking code coverage and crashes, our loop actively monitors whether test inputs reach any identified sinkpoints (line 13). We implement this monitoring through a custom Java agent instrumentation called `CodeMarkerInstrumentation`, which we integrated into our version of Jazzer (`Atl-Jazzer`). When the fuzzer executes an input that reaches a sinkpoint (*i.e.,* calls a sink API), the instrumentation generates a CodeMarkerHitEvent, an exception containing the complete stack trace at the sinkpoint. This runtime information is then captured and dumped to files for other components to utilize.

Upon detecting sinkpoint reaching, the loop collects "beep seeds" (line 14), inputs that successfully reach sinkpoints, and triggers targeted exploitation attempts (line 15). This design explicitly separates the exploitation phase from exploration, enabling specialized reasoning techniques to generate exploits using exact dynamic context. For vulnerabilities requiring specific exploitation conditions (*e.g.,* bypassing sanitizers or satisfying complex constraints), the beep seeds provide concrete execution contexts including call stacks, reproducible input blobs, and sinkpoint vulnerability information that guide exploitation generation. This infrastructure establishes sinkpoint reaching as a first-class feedback signal alongside coverage and crashes, forming the foundation for our sinkpoint-centered approach.

5.2.2 Infrastructure II - Ensemble Fuzzing

Ensemble fuzzing combines multiple fuzzing strategies by merging their collective efforts to improve overall bug-finding effectiveness. In ATLANTIS-Java, the ensembler serves three key functionalities: ① It collects and

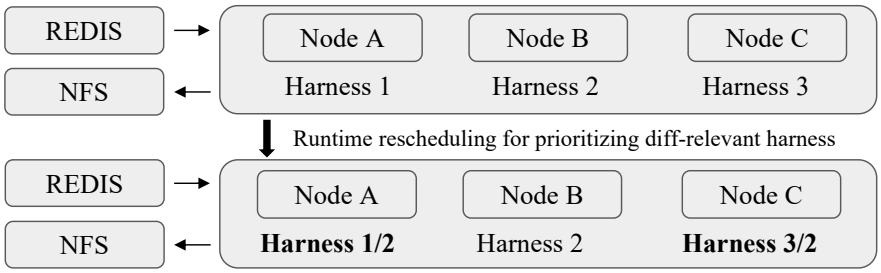


Figure 6: Runtime Rescheduling in ATLANTIS-Java for Diff Tasks

merges corpus from different fuzzer instances, performing deduplication based on coverage metrics before broadcasting the unified corpus back to all participants, enabling different strategies to benefit from each other's discoveries; ② It incorporates inputs from non-fuzzing components such as LLM-based generators or concolic execution engines into the fuzzing workflow, effectively bridging diverse input generation techniques; ③ It facilitates synchronization of sinkpoint metadata including sink reach status and beep seed collections during corpus merging, ensuring all fuzzer instances operate with up-to-date sink information. In general, the ensemble infrastructure serves as both a corpus synchronization layer and a metadata propagation channel across all fuzzing instances.

5.2.3 Infrastructure III - Sinkpoint Identification & Management

Sinkpoint identification forms the foundation of our approach by statically analyzing the target CP to locate all calls to security-sensitive sink APIs. Beyond Jazzer's built-in sink API list, we expanded our detection capabilities by collecting additional sink APIs from vulnerability benchmarks, published research papers, and static analysis tools used in the competition. Our custom sink API list is configured through YAML files in the system's CodeQL component, with support for future extension through additional configuration files.

The sinkpoint management component maintains metadata for each identified sinkpoint, including call graph information, runtime status (unreached, reached, exploited, or unexploitable), associated beep seeds, and exploitation attempts. This component dynamically schedules fuzzing resources based on sinkpoint status, such that already-reached sinks should focus more on exploitation, while sinks marked as unexploitable after thorough analysis are excluded from further processing. By doing so, the manager minimizes redundant efforts and prioritizes high-value targets such as SARIF/diff-relevant sinkpoints.

All exploration and exploitation components can access this centralized sinkpoint metadata through a shared interface, enabling informed decision-making throughout the competition. For instance, directed fuzzers only target unreached sinkpoints to guide path exploration, while exploitation agents retrieve beep seeds and stack traces for reached sinkpoints to craft targeted exploits. This unified access to sinkpoint information ensures that every component operates with consistent, up-to-date knowledge about the target program's attack surface.

5.2.4 Infrastructure IV - Distributed Design

Our distributed design employs a lightweight architecture that adapts to different competition task types. In full mode, each CP harness is assigned one and only one node, with all nodes working separately until the task deadline. During execution, NFS and Redis provide persistent caching for component outputs and system states, enabling seamless recovery from node failures or restarts. This design ensures fault tolerance while maintaining simplicity in deployment and management.

For diff mode tasks, the system initially deploys identically to full mode but incorporates a dynamic rescheduling mechanism to optimize resource allocation. As illustrated in Figure 6, after a predefined period

Component	Sinkpoint Exploration	Sinkpoint Exploitation
Directed Jazzer	✓	
LibAFL-Based Jazzer	✓	
Atl-Jazzer	✓	
Path-Based PoV Generator	✓	✓
Concolic Executor	✓	✓
DeepGen	✓	
DictGen	✓	
ExpKit		✓

Table 8: Component Overview of ATLANTIS-Java

(2 hours) or when static analysis results stabilize across multiple analyzers, the system performs a one-time rescheduling operation. This rescheduling reassigns system resources by redistributing nodes to focus on diff-relevant harnesses. For example, if Harness 2 contains critical diff-related sinkpoints, nodes originally assigned to Harnesses 1 and 3 may be reassigned to also process Harness 2, resulting in multiple nodes working on the same high-priority harness.

After rescheduling, when multiple nodes process the same harness, the sinkpoint manager leverages Redis to periodically synchronize sinkpoint metadata across all nodes. This synchronization includes sink status updates, beep seed collections, and exploitation attempts, preventing duplicate efforts while maximizing resource utilization on diff-relevant vulnerabilities. The lightweight nature of this distributed design, relying only on standard Redis and NFS infrastructure, minimizes our development and deployment efforts while providing flexibility for different competition tasks.

5.2.5 Component Overview

Beyond the infrastructure components, ATLANTIS-Java integrates multiple specialized tools for sinkpoint exploration and exploitation as shown in Table 8. These components leverage the infrastructure to work collaboratively: exploration-focused components such as Directed Jazzer and LibAFL-based Jazzer focus on reaching sinkpoints, while exploitation-focused components like ExpKit specialize in generating vulnerability-triggering inputs for reached sinks. Some components serve dual purposes, including the Path-Based PoV Generator and Concolic Executor, which perform both exploration and exploitation.

Two components deserve special mention: DictGen, adapted from ATLANTIS-Multilang, generates fuzzing dictionaries for fuzzer instances in various scenarios, including initial seed generation and beep seed exploitation phases (see §6 for details). Atl-Jazzer, our enhanced version of Jazzer, primarily adds beep seed tracking functionality alongside minor feature improvements and bug fixes. The complete feature list is documented in our code repository. Together, these components form a comprehensive toolkit that addresses different aspects of Java vulnerability detection within our sinkpoint-centered framework.

5.3 LibAFL-Based Jazzer

The LibAFL-based Jazzer component is designed to enhance mutation diversity beyond Jazzer's built-in mutators for generally improved sinkpoint exploration. While Jazzer itself provides effective libFuzzer mutations, relying solely on its default strategies can limit exploration breadth. By integrating LibAFL's advanced mutation algorithms and scheduling strategies, we aim to discover execution paths that standard Jazzer might miss, ultimately reaching more sinkpoints during the exploration phase.

Our integration leverages the LibAFL-libFuzzer project [10], which provides a LibFuzzer-compatible runtime built on LibAFL's fuzzing infrastructure. This enables a drop-in replacement architecture. While the base LibAFL-libFuzzer project provides API compatibility, integrating it with Jazzer required addressing

Original Jazzer Dependency Chain

```
Jazzer → Standard LibFuzzer → LLVM fuzzing engine
```

LibAFL-Based Jazzer Dependency Chain

```
Jazzer → libafl_libfuzzer → LibAFL fuzzing engine
```

Figure 7: Jazzer Dependency Chain Comparison

several technical challenges.

Finding Detection and Lifecycle Control. Java fuzzing requires specialized handling of findings and fuzzer lifecycle events that differ from binary fuzzing. We implemented custom observers (`JazzerFindingObserver` and `JazzerFuzzerStoppingObserver`) that hook into Jazzer's `__jazzer_set_death_callback` mechanism. This integration enables proper crash artifact dumping when Java-specific vulnerabilities are detected and supports graceful fuzzer termination based on Jazzer side's state. Without this adaptation, LibAFL would miss Java-level findings that don't manifest as traditional crashes.

Coverage Feedback Integration. Jazzer's bytecode instrumentation passes program counter (PC) information to sanitizer functions for precise coverage tracking. We extended LibAFL's coverage infrastructure with PC-aware variants (`__sanitizer_cov_trace_cmp4_with_pc`, *etc.*) that extract and forward this information to LibAFL's feedback mechanisms.

String Comparison Hooks. Jazzer relies on specialized string comparison behaviors. We implemented Jazzer-specific hooks including `__sanitizer_weak_hook_strstr` and `__sanitizer_weak_hook_compare_bytes` that integrate with LibAFL's comparison tracking while preserving Jazzer's requirements.

Mutation Strategy Control. To fully leverage LibAFL's mutation diversity, we disabled Jazzer's custom mutations through runtime flags (`-disable_custom_mutate`), allowing LibAFL's schedulers and mutators to control the fuzzing process exclusively.

Discussions. In our testing and evaluation, LibAFL-based Jazzer can show different coverage results compared to standard Jazzer in certain projects, demonstrating its complementary exploration capabilities. We think this is because LibAFL's mutation scheduler more aggressively prioritizes unexplored code regions. However, our current LibAFL-based Jazzer lacks support for value profile feedback, a mechanism Jazzer uses for simple exploitation scenarios (*e.g.,* gradually approaching string comparisons like "jazze"). Consequently, it shows weaker exploitation capabilities for vulnerabilities requiring specific value constraints.

5.4 DEEPGENERATOR

DEEPGENERATOR (previously discussed in §4.5.2) is a high-throughput framework that leverages LLM agents to generate seed mutation/generation scripts for fuzzers, thereby producing massive amounts of diverse seeds to improve coverage exploration. This framework represents a collaborative effort between the ATLANTIS-C and ATLANTIS-Java teams, designed to address the limitations of traditional mutation strategies through intelligent, context-aware seed generation. The complete system consists of three core components: LIBAGENTS, LIBDEEPGEN, and customized fuzzers with out-of-fuzzer (OOF) mutator support.

Architecture Overview. As illustrated in Figure 8, DEEPGENERATOR employs a straightforward architecture where LIBAGENTS serves as the intelligent generation layer. This component implements a deep research framework capable of analyzing target programs and generating context-aware Python mutation scripts. LIBAGENTS supports various tools including common command-line utilities, and provides wrappers for advanced AI coding assistants such as Claude Code and Codex agents. The deep research capability enables LIBAGENTS to explore the target codebase, understand API patterns, and generate scripts that

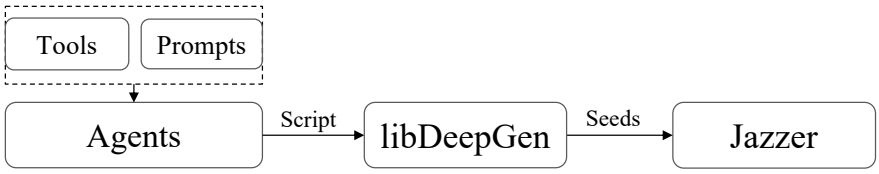


Figure 8: Overview of DEEPGENERATOR

produce semantically meaningful inputs.

The core runtime component, LIBDEEPGEN, manages the execution and scheduling of agent-generated scripts with highly optimized performance characteristics. Its design incorporates atomic operation-based ring buffers for script task scheduling and a combination of shared memory and ZeroMQ for seed storage and dispatch. This architecture enables extreme throughput, supporting thousands of script switches per second and generating hundreds of thousands of seeds at peak capacity. When LIBDEEPGEN executes scripts produced by LIBAGENTS, the generated seeds are sent to fuzzers through shared memory coordinates transmitted via ZeroMQ messages.

To integrate with existing fuzzing infrastructure, DEEPGENERATOR requires fuzzers to support OOFMutator (Out-Of-Fuzzer Mutator) functionality. This involves adding a ZeroMQ client to the fuzzer's mutation engine, enabling batch retrieval of generated seeds that are then used as mutation outputs. In ATLANTIS-Java, our Atl-Jazzer implements full OOFMutator support, allowing seamless integration with DEEPGENERATOR's high-throughput seed generation pipeline.

DEEPGENERATOR in ATLANTIS-Java Due to competition time constraints, we focused DEEPGENERATOR's deployment in ATLANTIS-Java primarily on initial corpus generation rather than its full potential for continuous mutation. For each harness, ATLANTIS-Java consolidates relevant information including harness code and task descriptions into structured prompts for LIBAGENTS. LIBAGENTS's deep research framework actively explores the CP codebase to gather harness-relevant context, analyzing existing test cases, API documentation, and code patterns to understand the harness generation specifics. This contextual understanding enables the generation of Python scripts that produce semantically valid and diverse inputs, significantly outperforming random or syntax-unaware generation strategies. ATLANTIS-Java iteratively notifies DEEPGENERATOR to generate multiple Python scripts, each producing a fixed quantity of seeds. During the execution, DEEPGENERATOR will filter out invalid or ineffective scripts, ensuring that only high-quality seeds are passed to the fuzzers.

Effectiveness. As an initial corpus generator, DEEPGENERATOR demonstrates remarkable effectiveness in jumpstarting fuzzing campaigns. Our internal benchmarks reveal substantial coverage improvements when comparing 10-minute fuzzing sessions with and without DEEPGENERATOR-generated initial corpora. In the most dramatic cases, we observed coverage increases around 2,952%, with over 20% of projects having improvements greater than 40%.

These results highlight DEEPGENERATOR's ability to quickly explore diverse program states that would require significantly longer discovery times through traditional mutation alone. These numbers validate the potential of LLM-guided seed generation for fuzzing. After the competition, we will extend DEEPGENERATOR's capabilities beyond initial corpus generation to support continuous, adaptive mutation strategies that evolve based on runtime feedback and discovered program behaviors.

5.5 Sinkpoint-Focused Directed Fuzzing

The sinkpoint-focused directed fuzzing component represents a combination of static analysis and runtime guidance, designed to efficiently navigate Java programs toward security-critical code locations (sinkpoints). Although primarily developed for reaching sinkpoints, this approach proves equally valuable for exploiting

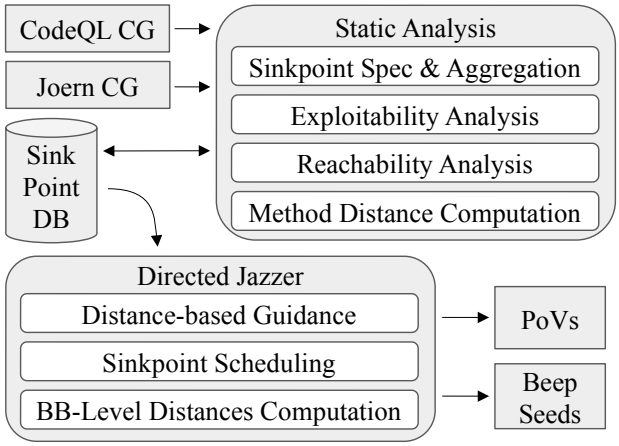


Figure 9: Overview of the Sinkpoint-focused Directed Fuzzing Architecture

sinkpoints once they are reached. The architecture orchestrates two complementary subsystems: a comprehensive static analysis pipeline combining CodeQL and Soot frameworks, and a modified Jazzer implementation that transforms distance computations into actionable fuzzing guidance.

An overview is available in Figure 9. The main inputs are call graphs from other CRS modules as well as data from the sinkpoint database. To improve performance, the system performs most analyses ahead of fuzzing, in the static analysis phase. This design choice is motivated by the constant need to compute distances from inputs to targets in directed fuzzing. By separating static analysis from the fuzzing phase, we eliminate two major computational overheads: 1) the cost of reconstructing the ICFG every time the fuzzer starts/restarts, and 2) the repeated traversal of the ICFG to calculate distances during directed fuzzing. Balancing both performance and implementation efforts, we eventually pre-compute and cache function-level distance calculations, and do basic block-level distance calculation (only requiring non-intraprocedural CFGs) during fuzzing. In addition, the fuzzer dynamically schedules sinkpoints based on their reachability, exploitability, and competition relevance.

5.5.1 CodeQL-Enhanced Sink Detection

Since the vast majority of bugs in Java applications can only happen when certain Java APIs are called, we developed a framework to define those sink APIs. While Jazzer has a list of such APIs that it sanitizes, we identified additional APIs that are likely to trigger a sanitizer. Our CodeQL component addresses this challenge by establishing a framework for specifying security-sensitive APIs that extend beyond Jazzer's built-in sanitizers. In particular, we added sinkpoints for calls to methods from the following classes:

- `java.math.BigDecimal`: The use of the `BigDecimal` class may result in a denial of service.
- `java.net.URL` and `java.net.URI`: Additional networking APIs, which may lead to an SSRF.
- `javax.validation.Validator`: May lead to an expression language injection, resulting in remote code execution.
- `javax.xml.parsers.SAXParser`: Parsing an XML may result in an SSRF.
- `org.apache.batik.transcoder.TranscoderInput`: Opening an SVG stream may lead to an SSRF.

Rather than exhaustively analyzing every library dependency, our CRS extends the list of sink APIs allowing us to only scan the challenge problem code instead of all libraries. The strategic advantage of

Example Sink Definition

```yaml
- model: # URL(String spec)
    package: "java.net"
    type: "URL"
    subtypes: false
    name: "URL"
    signature: "(String)"
    ext: ""
    input: "Argument[0]"
    kind: "sink-ServerSideRequestForgery"
    provenance: "manual"
  metadata:
    description: "SSRF by URL"
```

Figure 10: Example sink definition for `java.net.URL`.

this approach becomes apparent when considering the resource constraints of the competition. On our benchmarks, analyzing not only the challenge problem code but also the dependencies would take hours with Soot, which may be longer than our CRS has to analyze the challenge problem, without even starting the directed fuzzer. However, using our custom framework, we can skip analyzing the dependencies and reduce analysis time to a matter of minutes (depending on the size of the challenge problem) on our benchmark set, leaving most of the analysis time for the directed fuzzer to run.

Beyond identification, the CodeQL analysis incorporates an exploitability assessment that distinguishes between theoretically reachable sinks and practically exploitable ones. By analyzing data flow patterns, we determine whether sink API arguments may be influenced by attacker-controlled input. When our system finds strong evidence that a sinkpoint is not exploitable, we remove it from the analysis. We consider a sinkpoint not exploitable if, *e.g.,* the relevant argument is hard-coded or not tainted by attacker-controlled input. This filtering transforms a large list of potential targets (hundreds to thousands) into a manageable set of high-value sinkpoints (<100 sinkpoints for 95% of cases in our benchmark set).

The implementation leverages a centralized sink definition architecture built around YAML configuration files that separate CodeQL model specifications from descriptive metadata. An example specification is given in Figure 10 for the `java.net.URL` class, which may result in an SSRF.

5.5.2 Static Analysis for Distance Computation

The static analysis foundation tackles one of directed fuzzing's most challenging problems: accurately measuring the distance from program entry points to sinkpoints. Our approach begins with per-harness reachability analysis, ensuring that computational resources focus exclusively on sinkpoints that are theoretically accessible from specific fuzzing entry points. This preliminary filtering eliminates unreachable targets that would otherwise consume valuable analysis time or guide the fuzzer towards unreachable code locations.

Recognizing that no single static analysis tool provides complete call graph coverage, we merge call graphs from multiple other CRS components, based on analysis frameworks such as Joern and CodeQL and extended with our own analyses to resolve calls. This multi-tool approach proves particularly valuable for handling Java's complex object-oriented features, where interface calls and reflective invocations often confound individual analysis tools. Each framework contributes unique strengths: CodeQL excels at pattern-based detection, while Joern provides flexible code property graph analysis.

The merging process confronts technical challenges arising from inconsistent naming conventions across analysis tools. Lambda expressions and nested classes receive different internal representations in various frameworks, requiring normalization and heuristics to establish consistent target identification. Our system resolves these discrepancies by mapping all call sites to specific Jimple instructions, *i.e.,* Soot's intermediate representation, creating a unified foundation for subsequent distance calculations.

Distance computation itself employs a hierarchical approach that balances precision with computational efficiency. Call graphs from static analysis provide coarse-grained method-level distances, while control flow graphs within Jazzer enable fine-grained basic block-level precision. We only compute the call graph ahead of time and not the control flow graphs, since there are a number of limiting factors that would come with such an approach. Statically computing distances on a basic block level would require a stable mapping of basic blocks to distances. However, the instrumentation of Jazzer (JaCoCo) changes the bytecode and would require us to rely on heuristics to match basic blocks, which we want to avoid. Another approach would be to map the coverage IDs to distances. However, coverage IDs are not deterministic by default. Coverage IDs can be made deterministic by instrumenting jar files ahead of time, but we want to avoid modifying the test artifacts as this would be a threat to stability. The reason why we cannot directly map bytecode offsets of basic blocks from the test artifacts to the instrumented classes is because the number of bytes that the bytecode instrumentation adds is difficult to predict. This is because every call to record a triggered coverage ID comes with a constant, which is part of the run-time constant pool. The resulting increase in the number of constants may require certain instructions to use a different variant to support wider indices, resulting in changes even to instructions that are present in the code before the instrumentation happens. Thus, we resort to generating the CFG at runtime of the fuzzer as the most reliable and precise approach.

An example result of the static analysis is illustrated in Figure 11. It contains a regex sinkpoint in the Apache Tika library and specifies the location in the code (class name, method name, descriptor, bytecode offset) as well as metadata. The analysis adds the reachability and exploitability results for each harness. The sinkpoint also contains information about whether it was reached and if it is derived from a diff or SARIF report, which will be used for prioritization.

5.5.3 Directed Jazzer

Our version of Jazzer contains a component to guide the fuzzer towards specified code locations. This new component transforms static analysis insights into dynamic fuzzing guidance through an adaptive scheduling system that schedules up to 15 concurrent sinkpoint targets at a time. The scheduler continuously monitors target status, automatically removing reached sinkpoints from active consideration while promoting newly discovered targets to active status.

Target prioritization reflects the competitive nature of the AIxCC environment through a prioritizing round-robin algorithm that allocates double scheduling time to sinkpoints derived from diff analysis and SARIF reports. This prioritization ensures that competition-relevant vulnerabilities receive appropriate attention while maintaining broad coverage of the overall attack surface. The dynamic nature of this scheduling adapts to changing analysis results, automatically adjusting priorities as new data becomes available.

The distance computation engine confronts the practical reality that instrumentation identifiers change between execution runs, making pre-computed mappings unreliable. Rather than modifying target JARs—which would compromise system stability and reliability—the implementation employs dominator analysis to establish runtime correlations between coverage identifiers and Soot basic blocks. This approach proves essential because Jazzer's coverage instrumentation does not instrument every basic block, requiring intelligent inference of distances for uninstrumented code regions.

Our system maintains comprehensive situational awareness through continuous monitoring of distance files, automatically detecting and incorporating updates from ongoing static analysis. This dynamic adaptation capability enables the fuzzer to respond to evolving analysis results. Target metadata encompasses reachability status, exploitability assessments, and priority classifications, providing the scheduler with rich information for making optimal resource allocation decisions throughout extended fuzzing campaigns.

Example Static Analysis Result


```
{
  "coord": {
    "class_name": "org.apache.tika.utils",
    "method_name": "<clinit>",
    "method_desc": "()V",
    "bytecode_offset": 2,
    "mark_desc": "sink-RegexInjection",
    "file_name": "ExceptionUtils.java",
    "line_num": 31
  },
  "type": [
    "sink-RegexInjection"
  ],
  "in_diff": false,
  "sarif_reports": [],
  "beepseeds": [],
  "ana_reachability": {
    "TikaAppRUnpackerFuzzer": true,
    "HtmlParserFuzzer": true,
    "RTFParserFuzzer": true,
    "TextAndCSvVParserFuzzer": true,
    "ThreeDXMLParserFuzzer": true,
    "TikaAppUnpackerFuzzer": true,
    "M3U8ParserFuzzer": true
  },
  "ana_exploitability": {
    "TikaAppRUnpackerFuzzer": false,
    "HtmlParserFuzzer": false,
    "TikaAppUntarringFuzzer": false,
    "RTFParserFuzzer": false,
    "TextAndCSVParserFuzzer": false,
    "XliffParserFuzzer": false,
    "ThreeDXMLParserFuzzer": false,
    "TikaAppUnpackerFuzzer": false,
    "M3U8ParserFuzzer": false
  },
  "reached": false,
  "exploited": false
}
```


Figure 11: Example of the static analysis result for a regex sinkpoint in Apache Tika.

5.6 ExpKit

ExpKit is a specialized exploitation component designed to address the "last mile" challenge in Java vulnerability detection: cases where fuzzers successfully reach sinkpoints but fail to trigger actual exploits. Through large-scale analysis and targeted LLM-based exploitation generation, ExpKit transforms reached-but-unexploited sinkpoints into successful vulnerability discoveries.

5.6.1 Motivation: The Last Mile Challenge

We conducted a large-scale fuzzing experiment across 42 CPVs in our benchmark, allocating 8 hours and over 100 CPU cores per harness. The results revealed a critical gap in traditional fuzzing approaches.

As shown in Table 9, while 73.8% of CPVs had their sinkpoints reached through fuzzing, only 35.7% resulted in successful exploits. This 38.1% gap represents the "last mile" challenge: cases where traditional value profile feedback mechanisms prove insufficient for triggering vulnerabilities despite reaching sinkpoints.

Total CPVs	Not Reached	Reached Only	Exploited
42 (100%)	11 (26.2%)	16 (38.1%)	15 (35.7%)

Table 9: Large-scale Fuzzing Results Showing the Last Mile Challenge

5.6.2 Analysis of Exploitation Gaps

The root causes of those 16 reached-but-unexploited CPVs include:

- **Distracted by seed explosion (4/16).** Fuzzers generated overwhelming amounts of inputs reaching sinkpoints, but failed to focus on exploitation-relevant mutations.
- **Missing sink API instrumentation (2/16).** Some sinkpoints used APIs not covered by standard value profile instrumentation (there are significant gaps between the sink API and the Jazzer-hooked functions which can trigger value profile feedback), leaving fuzzers without exploitation guidance.
- **Insufficient value profile depth (1/16).** Complex sanitizers required longer input sequences than value profile mechanisms could effectively guide.
- **Complex exploitation logic (9/16).** This is the most common case, where exploitation required reasoning about complex conditions, multiple API interactions, or specific input formats that traditional fuzzing strategies could not handle even with the help of value profile feedback. For instance, to trigger a XML deserialization vulnerability, the fuzzer must generate a specific XML payload that satisfies type constraints and bypasses validation checks.

These findings motivated ExpKit's design as a specialized exploitation agent that leverages beep seeds' rich execution context combined with LLM reasoning to bridge the semantic gap between reaching and exploiting sinkpoints.

5.6.3 Key Design

ExpKit operates as an autonomous exploitation agent that processes beep seeds, *i.e.,* inputs that successfully reach sinkpoints, to generate targeted exploits. The tool is implemented with several key design principles.

Beep Seed Scheduling and Exploitation Loop. ExpKit groups beep seeds by their execution context, defined as the complete stack trace at the sinkpoint. This grouping enables fair scheduling across different execution paths. The scheduler prioritizes contexts with fewer exploitation attempts, implementing a two-dimensional fair scheduling algorithm based on the tuple $\langle \text{attempt\_count}, \text{context\_id} \rangle$. For each exploitation round, ExpKit selects the context with minimum attempts and randomly picks a beep seed from that context for processing. During competition runs, ExpKit operates continuously, processing new beep seeds as they arrive from fuzzing components and immediately testing generated exploit attempts.

Context Collection and Prompt Generation. For each selected beep seed, ExpKit collects context information and transforms it into structured prompts that guide LLMs through exploitation reasoning:

- Complete source code of all files appearing in the stack trace
- Sinkpoint details including line number, sink API specification, vulnerability type, vulnerability descriptions, exploitation specifics, *etc.*
- Hex dump of the reaching input and full stack frames
- Explicit exploitation task requirements and output format specifications for direct input blob generation

The generated exploit blobs are then passed to a dedicated fuzzing phase without coverage feedback (since the sinkpoint is already reached), allowing near-successful PoCs to undergo further mutations for successful exploitation. Subsequently, both successful exploits and failed attempts, along with their associated seeds, are shared with all fuzzer instances through the ensemble infrastructure.

5.6.4 Effectiveness

ExpKit demonstrated remarkable effectiveness in addressing the last mile challenge. Of the 16 reached-but-unexploited CPVs, ExpKit successfully generated exploits for 13 cases (81.3% success rate). The success cases span diverse vulnerability types such as command injection, path traversal, and deserialization vulnerabilities, validating ExpKit's generality across different exploitation scenarios. Particularly noteworthy is that 6 of these vulnerabilities were exploited with a single LLM query (using the OpenAI o1-preview model), demonstrating the power of combining precise execution context with language model reasoning.

The three remaining cases required complex multi-round iteration and specialized tools beyond ExpKit's current single-query design. While ExpKit proves highly effective for straightforward exploitation scenarios, post-competition development will focus on evolving it into a full-featured exploitation agent with tool access (debuggers, analyzers, fuzzers), multi-round reasoning capabilities, and diverse blob generation methods including script-based approaches. This evolution will address the remaining complex cases while improving efficiency for simpler exploitations.

5.7 Concolic Execution

We utilize a Java concolic execution engine (*i.e.,* concolic executor) to serve two primary roles in our workflow. The executor acts as a *coverage amplifier* to explore deep and complex code paths that fuzzers cannot easily reach, and as an *exploit assistant* to help turn a *reached-but-unexploited* sink into a proof-of-vulnerability (PoV). To this end, we developed a new concolic execution engine on GraalVM Espresso [23], a JIT-compiled JVM execution environment running on GraalVM.

5.7.1 Motivation

Existing Java concolic engines are limited in supporting generic vulnerability discovery because they either support only specific Java versions of the JVM (version 8 or 11) for their binary compatibility—*e.g.,* Java PathFinder, jCUTE, DART, JBSE—or often fail to run production-grade open-source software due to collisions in their bytecode instrumentation method (`asm`)—*e.g.,* SWAT and COASTAL. For the latter in particular, execution on critical areas such as class initialization (`<clinit>`), use of Mockito, lambda expressions, *etc.*, which are essential to explore program paths with fuzzing and testcase harnesses, were shown to be divergent when running with the framework, rendering them infeasible to apply to production-grade software.

Such obstacles motivated us to develop a Java concolic executor at the bytecode interpreter level (*i.e.,* emulate bytecode instructions themselves symbolically under-the-hood) to achieve *binary compatibility* up to the latest version of Java (25 and onward). This way, we can *run all critical executions in Java*, and *scale to production-level applications* robustly.

5.7.2 Our Approach: Interpreter-based Symbolic Emulation

We built our concolic execution engine on *GraalVM Espresso*, a high-performance (JIT-compiled) bytecode interpreter. This approach offers several key advantages over existing frameworks:

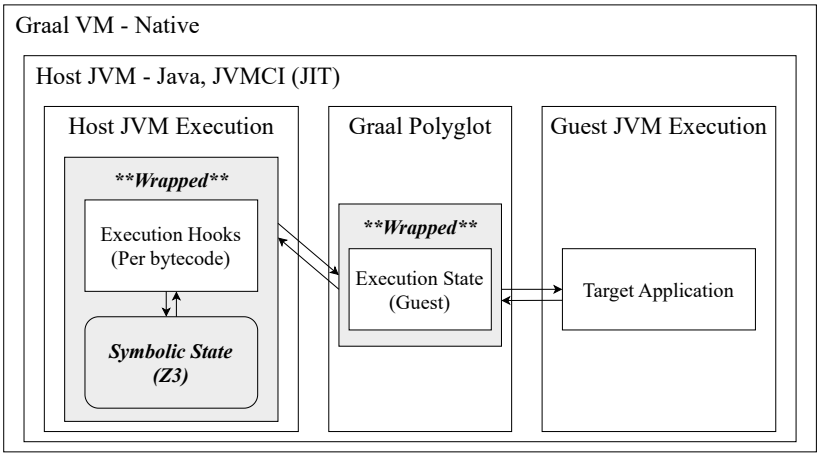


Figure 12: Concolic Executor Overview

- **Compatibility without Instrumentation:** Dynamically instrumenting bytecode allows us to hook every execution at the smallest possible granularity (*i.e., bytecode*), not losing any tracking of the runtime symbolic state of the target program.
- **State Isolation:** Espresso runs the target application in an isolated context. The dynamic instrumentation as well as shadow symbolic state are completely invisible to the target runtime, unlike `asm`-based static instrumentation that suffers symbolic/concrete execution state contamination.
- **JIT-Accelerated Performance:** Symbolic state emulation incurs nontrivial overhead in the target execution. To mitigate this overhead, our engine leverages Espresso's Just-In-Time (JIT) compilation for shadow symbolic execution. After the first visit of a code path, the code will be JIT-compiled, and thereby, subsequent visits to the same path are executed a near-native speed (for both the target execution as well as symbolic state emulation).

In our approach, the core concept is to "wrap" all Java data types—from primitives like `int` to complex `Object` instances—in custom container classes. These containers hold both the concrete value and its corresponding symbolic representation, allowing symbolic state to be propagated naturally as the program executes. Since this management is performed at the JVM level, no modifications to the target code are required, and the original program runs normally without knowing that symbolic execution is happening. Z3 expressions are used to represent symbolic states, with calculations and operations performed only when necessary for efficiency.

Core Modifications. To extend the framework so that both concrete values and symbolic states can be managed during the execution, three core modifications were performed. First, data types used by the JVM (such as `Object`, `int`, `long`, *etc.*) are wrapped to allow simultaneous management of both concrete values and symbolic states. Second, data storages in the JVM, such as local variables and the operand stack, are made compatible with these wrapped data types. Third, with data types and storage now supporting symbolic states, the bytecode handlers and other operations are extended so that operations can be performed not only on concrete values but also on symbolic values.

Fallback. However, the structure of Espresso and the JVM is complex and extensive, and there are some challenging bytecodes, such as those for floating-point operations. Therefore, it was not possible to wrap and extend all functionalities. When unsupported functionalities are encountered, the symbolic information is discarded and the value is downgraded to its original, unwrapped concrete form. Although this results in the loss of symbolic state, it ensures compatibility and robustness in program execution. In particular,

Concolic Execution Example

```
public static int test(byte[] input) {
    int a = input[0]; // Symbolic
    int b = 5;        // Concrete
    if (a > 100) {
        return a + b; // Path 1
    }
    return a - b;     // Path 2
}
```

Type	Extended Class
byte	ConcolicByte
short	ConcolicShort
char	ConcolicChar
int	ConcolicInt
long	ConcolicLong
float	ConcolicFloat
double	ConcolicDouble

Table 10: Primitive Types and their Extended Classes

this strategy was highly beneficial for ensuring compatibility with internal framework code in Espresso. During program execution, data is not only processed by bytecode handlers, but also interacts with various Espresso internal framework components, which provide optimizations and extended functionalities. These interactions are often complex and diverse, making them difficult to fully support, yet they are largely unrelated to the symbolic states encountered during target program execution. By downgrading in situations involving framework-internal interactions that are irrelevant to symbolic state management, it was possible to reduce implementation effort while achieving robust compatibility.

5.7.3 Overall Workflow

We adopt the fuzzer entry point, `fuzzerTestOneInput()`, as our execution root. The input is typically provided either directly as a byte array or via a `FuzzedDataProvider`. The symbolic state is initialized by wrapping the input data. We hook bytecode execution, symbolizing the input and tracking symbolic propagation through methods, objects, fields, *etc.* To illustrate, consider the above simple Java method:

When our concolic executor runs this code, it operates as follows:

1. **Initialization:** The value `input[0]` is loaded. Its concrete value (*e.g.,* '42') is stored, and it is assigned a symbolic variable, let's call it S_0. The integer 'a' is now represented as a wrapped type holding (concrete: 42, symbolic: S_0). The integer 'b' is also wrapped but initially has no symbolic state: (concrete: 5, symbolic: null).

2. **Constraint Collection:** At the `if (a > 100)` branch, the executor records the path constraint. Since 'a' has a symbolic state S_0, the condition $S_0 > 100$ is added to a list of constraints for the current execution path. Concretely, $42 > 100$ is false, so the 'else' branch (Path 2) is taken.

3. **Symbolic Propagation:** The operation `a - b` is performed. The executor calculates the new concrete value ($42 - 5 = 37$) and also computes the new symbolic expression by combining the symbolic states of the operands: $S_0 - 5$.

4. **New Input Generation:** After the execution finishes, the solver can negate the collected path constraint. It asks Z3 to solve for $\neg(S_0 > 100)$, which simplifies to $S_0 \leq 100$. The solver might also be asked to find an input that satisfies the other path, $S_0 > 100$. Z3 might return a solution like $S_0 = 120$. The executor then crafts a new `input` byte array where the first byte is '120' to explore Path 1 in the next run.

This process of lazily converting values to symbolic expressions only when they interact with other symbolic data avoids generating bloated and unnecessary Z3 terms, significantly improving solver performance.

5.7.4 Symbolic State Management

Class Definitions

```
public class ConcolicValueWrapper<T> {
  protected T concrete_value;
  protected Expr<?> expr = null;
  ...
}

public class ConcolicInt
  extends ConcolicValueWrapper<Integer> {
  ...
}
```

JVM Bytecode Handler Comparison

```
@@ Original vs. Wrapped Bytecode Handler @@
 case IADD: {
-  int value1 = popInt(frame, top - 2);
-  int value2 = popInt(frame, top - 1);
-  int result = value1 + value2;
-  putInt(frame, top - 2, result);
+  ConcolicInt value1 = popConcolicInt(frame, top - 2);
+  ConcolicInt value2 = popConcolicInt(frame, top - 1);
+  ConcolicInt result = value1.Add(value2);
+  putConcolicInt(frame, top - 2, result);
   break;
 }
```

Figure 13: JVM bytecode handler for `IADD`: Original vs. Wrapped

To accurately track symbolic state across complex Java applications, we implement comprehensive handling for key JVM features, combining wrapper-based value modeling for JVM values (primitives, arrays/objects, `String`) with boundary summaries (function hooks, boxed types, file I/O).

Primitives. All primitive types (`byte`, `short`, `char`, `int`, `long`, `float`, `double`) are extended with `Concolic*` wrappers (Figure 10) that store a concrete value (`concrete_value`) and a symbolic expression (`expr`). These extended classes also expose arithmetic and, when necessary, bitwise operations that update both the concrete and symbolic parts during computation. The bytecode handlers were updated accordingly; Figure 13 shows `IADD`, whose handler pops/pushes concolic values (`popConcolicInt()`, `putConcolicInt()`), invokes the corresponding concolic operation (*e.g.,*`value1.Add()`), and updates both states.

Arrays and Objects. Java programs are object-heavy, with frequent cross-object interactions; therefore, precise and efficient state management is essential. We treat objects as field containers and arrays as element containers, where each slot may carry its own concrete value and symbolic state. To keep overhead low—even for large, nested, or multidimensional structures—we lazily materialize member states on first read or write along the executed path. Static members are tracked in a class-level symbolic map to maintain per-class semantics consistently across instances.

java.lang.String. We treat `String` as a special object, since modeling symbolic state for strings is particularly challenging. Two approaches are viable: (i) encode strings with dedicated string theories (*e.g.,* Z3Str/Seq) to reason directly at the symbolic-string level, or (ii) track the backing store (*i.e.,*`String.value` as a `byte[]`) and link the underlying bytes to their symbolic states. We adopt (ii), recording the bytes for each instance and associating them with symbolic state. While (i) can express richer, more abstract properties, it proved difficult to implement correctly given Java's string semantics; the backing-store approach is less expressive but substantially more robust and performant in practice.

Function hooks and summaries. For native or semantically dense methods, we instrument call boundaries with pre/post hooks and attach compact summaries that capture only essential relations (e.g., prefixes, lengths, bounds, aliasing) rather than expanding full method bodies. In practice, we provide manual summaries for frequently invoked native methods—`System.arraycopy()`, `Object.clone()`, and `Unsafe` methods—that preserve copy length, bounds checks, and element-wise aliasing. To ensure full compatibility with Jazzer, we also model the `FuzzedDataProvider` APIs, propagating symbolic provenance through calls such as `consumeBytes`, `consumeRemainingAsBytes`, and `consumeString`.

Boxed Types (Autoboxing/Unboxing). Java provides wrapper classes for primitives, each of which encapsulates its corresponding primitive value. At autoboxing/unboxing boundaries (*e.g.,*`Integer.valueOf` / `intValue`), we preserve the link to the underlying primitive's concolic state. Because boxed values may be cached, we re-inject the symbolic state after `valueOf` to avoid stale or missing expressions.

Concolic Solving Example

```
} else if (marker == JpegConstants.DQT_MARKER) {
  final DqtSegment dqtSegment = new DqtSegment(marker, segmentData);
  for (final QuantizationTable table : dqtSegment.quantizationTables) {
    if (0 > table.destinationIdentifier || table.destinationIdentifier >= quantizationTables.length) {  // (1)
      throw new ImagingException("Invalid quantization table identifier " + table.destinationIdentifier);
    }
    // not explored under fuzzing-only
    ...
  }
} else if (marker == JpegConstants.DHT_MARKER) {    // (2)
  // not explored under fuzzing-only
  final DhtSegment dhtSegment = new DhtSegment(marker, segmentData);
  ...
}
return true;
```

File I/O. To maintain symbolic integrity across file operations, we extended `FileInputStream`, `FileOutputStream`, and `RandomAccessFile`, together with the required dispatchers. These wrappers preserve the symbolic state of file contents, file descriptors, and memory-mapped regions, and they include symbolic handling for file-pointer adjustments. This allows the executor to track whether symbolic input data is written to or read from files, ensuring that symbolic reasoning remains intact throughout I/O interactions.

5.7.5 Constraint Solving

During execution, whenever the program reaches an exploration target, the executor collects the associated path constraints. After each execution, it solves these collected constraints to synthesize new inputs that either steer execution into previously unexplored branches or bias execution toward exploitation-oriented behaviors at vulnerability-related sinks.

(A) Constraint Solving for Exploration. When a new branch predicate is observed, the executor negates it and solves a *minimal related slice* of constraints—*i.e.,* only those constraints that influence the predicate—to produce an input taking the opposite outcome. This raises coverage and input diversity while limiting incidental changes to unrelated bytes.

To check the contribution of concolic execution to exploration, we ran it on top of a corpus collected after 1 hour of fuzzing `org.apache.commons.imaging.Imaging.getBufferedImage()` in *Apache Commons Imaging*. This run showed an increase of about 9.5% branch coverage over the fuzzing-only baseline. The following excerpt illustrates part of this improvement.

In the excerpt above, guard (1) is a range check on `destinationIdentifier`, and guard (2) checks whether `marker == JpegConstants.DHT_MARKER`. With fuzzing alone, the corpus repeatedly triggers the exception at (1), leaving the following block unexecuted; similarly, (2) never holds, so the `DhtSegment` path is uncovered. This reflects a common trajectory in mature fuzzing campaigns: once inputs are locally "good enough" to reach a region, the chance of stumbling into *new* branches inside that region drops, especially when deeper progress requires coordinated changes across multiple bytes and checks.

With concolic execution, condition (1) was negated to satisfy the range check by solving only the bytes influencing `destinationIdentifier`. This ensured $0 \leq$ `destinationIdentifier` $<$ `quantizationTables.length`, and allowed us to visit a block that had remained unexecuted under fuzzing alone. Likewise, condition (2) was flipped by solving the marker bytes so that `marker == DHT_MARKER`, which enabled the `DhtSegment` branch to be explored for the first time.

(B) Constraint Solving for Exploitation. We further target *sink-aware* generation: when symbolic values are present at a sink, the executor solves constraints to bias inputs toward exploit-relevant behaviors—for

Out-Of-Memory in LZWInputStream.initializeTables()

```
protected void initializeTables(final int maxCodeSize) {
    ...
    if (1 << maxCodeSize < 256 || getCodeSize() > maxCodeSize) {
        // TODO test against prefixes.length and characters.length?
        throw new IllegalArgumentException("maxCodeSize " + maxCodeSize + " is out of bounds.");
    }
    final int maxTableSize = 1 << maxCodeSize;
    prefixes = new int[maxTableSize];
    characters = new byte[maxTableSize];
    outputStack = new byte[maxTableSize];
    outputStackLocation = maxTableSize;
```

example, allocating arrays large enough to induce an `OutOfMemoryError` or constructing values that trigger Jazzer's detectors. This is motivated by cases where fuzzers reached a vulnerability-related sink but failed to trigger actual exploitation.

Jazzer can also guide mutations toward such sinks via compare/substring feedback (*e.g.,* `traceMemcmp`, `traceStrstr`). This works when the compared value is derived almost directly from raw input, but it weakens after non-linear transforms (hashes/CRCs/bit-twiddling), multi-step encodings, or table lookups; and because the signal is local, it struggles to satisfy multiple, distant preconditions.

To address these limits, our executor adds minimal payload constraints and solves them after reachability is established. We primarily leverage Jazzer's sentinels and model constraints for hooked methods such as `java.lang.ProcessBuilder.start()` and `java.net.Socket.connect()`. we also target methods not directly hooked by Jazzer with early, non-sanitizer constraints, increasing the overall potential for exploitation. There remains ample opportunity to extend these ideas to other forms of exploit-oriented generation.

Out-Of-Memory via Oversized Array Allocation. We identify array allocations through the `NEWARRAY` and `ANEWARRAY` bytecode instructions, and use Z3's `Optimize` to maximize the allocation length under the feasible path conditions. This steers execution toward inputs most likely to provoke an `OutOfMemoryError`.

Assume the program computes `len` from four input bytes at offset p and allocates `new byte[len]`. Here, the four bytes $b_p, \ldots, b_{p+3}$ are combined into a 32-bit `int` in big-endian order. We then solve:

$$
\begin{aligned}
& b_i \in [0, 255] \quad (i \in \{p, \ldots, p{+}3\}), \\
& \texttt{len} = (b_p \ll 24) \mid (b_{p+1} \ll 16) \mid (b_{p+2} \ll 8) \mid b_{p+3}, \\
& \textbf{path conditions:}\ \ C_1(b_\cdot) \wedge \cdots \wedge C_m(b_\cdot), \\
& \textbf{objective:}\ \ \max \texttt{len}.
\end{aligned}
$$

where each C_i denotes the instantiated outcome of the i-th branch predicate on the chosen path, such as checksum/range checks that gate the allocation. The dependency slice includes only the bytes influencing `len` and the relevant guards, keeping solving fast and localized.

This case is drawn from `Apache Commons Compress` in competition round 3. As in the exploration section, the array allocation is guarded by conditions that the executor should satisfy. Beyond satisfying these guards, the executor synthesizes inputs that maximize the subsequent allocation size where `maxTableSize = 1 << maxCodeSize`, thereby provoking an `OutOfMemoryError`. Although the path is relatively deep, we found that the executor collected the constraints along the way and synthesized an input that reached the allocation site and triggered the issue.

OS Command Injection. For command injection, Jazzer instruments `java.lang.ProcessBuilder.start()` and reports an `OsCommandInjection` finding when the executable filename is recognized as the sentinel `"jazze"`. In practice, multiple preconditions must be satisfied before this sink is reached, and in our experiments a coverage-guided fuzzer alone often struggled to trigger the detector—even on paths that met the validations—and in some cases failed to discover it at all.

OS Command Injection in GzipCompressorInputStream.init()

```
final DataInput inData = new DataInputStream(in);              // Input
final int method = inData.readUnsignedByte();
if (method != Deflater.DEFLATED) {                             // Validation 1
    throw new IOException("Unsupported compression method " + method + " in the .gz header");
}
final int flg = inData.readUnsignedByte();
if ((flg & GzipUtils.FRESERVED) != 0) {                        // Validation 2
    throw new IOException("Reserved flags are set in the .gz header.");
}
long modTime = ByteUtils.fromLittleEndian(inData, 4);
...
String fname = null;
// Original file name
if ((flg & GzipUtils.FNAME) != 0) {                            // Validation 3
    fname = new String(readToNull(inData), parameters.getFileNameCharset());
    parameters.setFileName(fname);
}
if (modTime == 1731695077L && fname != null) {                 // Validation 4
    new ProcessBuilder(fname).start();                         // Sink
}
```

In another case from `Apache Commons Compress` in competition round 3, once validations 1-4 are satisfied and `new ProcessBuilder(fname).start()` is invoked with a filename recognized as the sentinel, Jazzer reports an OS command-injection finding. On the executor side, we use a simple sufficient precondition for solving: `fname` equals `"jazze"` or ends with `"/jazze"`. This accommodates path-shaped inputs (*e.g.,*`"/tmp/jazze"`) while leaving filename resolution to the runtime and preserving the detector's semantics.

In addition to OS command injection, we apply a similar approach to other Jazzer-hooked sinks—such as Path Traversal or Reflection—tailoring the constraints to each vulnerability signature.

5.7.6 Optimization

In concolic execution, solving all collected constraints can be highly time-consuming, and in practice it is rarely feasible to resolve every constraint within a limited budget. As execution progresses, constraints not only become more complex but also grow in number, which can lead to severe inefficiency or solver failures. To mitigate these challenges and make constraint solving more effective, we introduce several optimization strategies. The following describes the primary approaches employed in our system.

Constraint Limiting. The concolic execution handles symbolic variables that may appear anywhere in the code, including as comparison conditions inside loops executed hundreds of thousands of times. Each comparison generates constraints, resulting in an enormous number of constraints that complicate later solving and may even cause memory exhaustion, halting execution.

To address this, we limit the number of constraints that can be generated from the same program counter (PC) to a small number (*e.g.,* 5). This was highly effective because it prevents the constraints from being created as many times as the number of iterations; in one test case, it reduced the number of SAT constraints from over 5,000, preventing an `OutOfMemoryError`. The trade-off is a potential loss of information, as the same PC might be reachable via different contexts, but this was crucial for stability.

Package Blacklisting. Since symbolic states can propagate through most JVM operations, constraints can also be generated by common Java packages and libraries. However, this sometimes results in the creation of excessive constraints, making constraint solving difficult due to overconstraint. To address this, we implemented filtering for certain packages to prevent the generation of unnecessary constraints.

Coverage-Guided Prioritization. The number of constraints could be substantial, and considering the limited computational time and resources available, solving all constraints may be infeasible. We observed cases where the solver attempted to solve constraints that did not directly increase block coverage because their successor blocks had already been visited. While such efforts may indirectly contribute to coverage

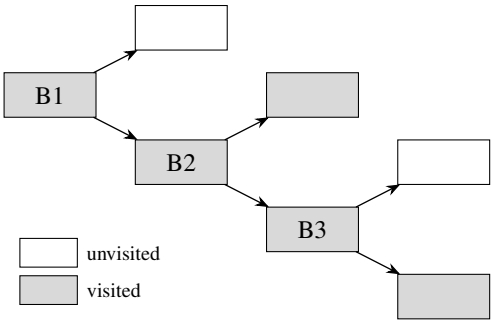


Figure 14: Sample branch blocks

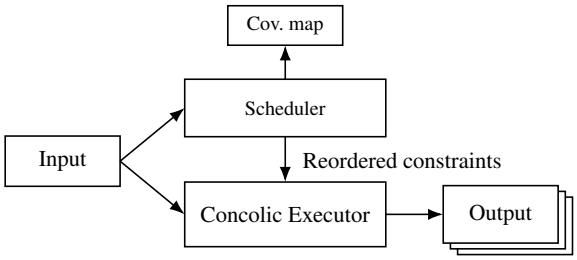


Figure 15: Scheduler

improvement, it is more efficient to prioritize constraints that directly lead to new block coverage given resource constraints. Therefore, before constraint solving, we employ a heuristic-based prioritization strategy to maximize new block coverage:

1. Prioritize branches that have never been attempted before, as their successful solving has the potential to yield new coverage gains since their feasibility and coverage impact remain unexplored.

2. Next, prioritize branches whose successor blocks have not been visited, as solving these constraints can directly contribute to block coverage expansion by enabling access to previously unreachable code blocks.

3. Finally, prioritize deeper branches, as they are more likely to have been recently discovered, thus representing untapped opportunities for coverage enhancement.

This reordering does not guarantee optimality. For example, branches that have been attempted previously may contribute differently to coverage depending on the execution context. Our strategy does not account for this contextual variation, potentially under-prioritizing such constraints. Nevertheless, it effectively directs solving efforts toward the most promising opportunities for direct coverage improvement.

Figure 14 shows the executed (visited) blocks and the order in which the constraints are solved. The branches were executed in the order `B1`, `B2`, and `B3`; however, with our strategy, the constraints are solved in a different order: `B3`, `B1`, and `B2`.

Figure 15 illustrates the scheduler functionality designed to prioritize constraints. The scheduler maintains up-to-date block coverage information by updating the coverage map based on the given inputs. With this up-to-date coverage data, it assigns priorities to constraints in order to guide the concolic executor more effectively.

5.8 Path-Based PoV Generator

The Path-Based PoV Generator is an AI agent that produces inputs capable of triggering vulnerabilities in Java programs through a given harness. It operates by identifying sinkpoints within the provided code, finding a path from the harness to each of them, and then generating a PoV for each path.

Figure 16 is an example code snippet to explain how this tool works. This code is a simplified version of a Jenkins example, originally presented for the ASC, where command injection can be triggered at lines 33 and 36. To address this vulnerability, the tool first identifies all function calls in the code that are related to command injection, which is line 36 in this example. Then it statically explores the execution path from the parameters of the harness entry point, `fuzzerTestOneInput`, to these calls. In the example, this call path is $Line\ 1 \rightarrow Line\ 5 \rightarrow Line\ 19 \rightarrow Line\ 29 \rightarrow Line\ 36$. Once a path is identified, the tool collects all relevant code along it by concatenating related method bodies, and provides it to the LLM. The LLM then generates an input value capable of triggering command injection. While traditional fuzzing would struggle

Running Example for Path-Based PoV Generator

```
public static void fuzzerTestOneInput(byte[] data) throws Exception {
  new JenkinsTwo().fuzz(data);
}

private void fuzz(byte[] data) {
  ByteBuffer buf = ByteBuffer.wrap(data);
  int picker = buf.getInt();
  if (count > 255) return;
  String whole = new String(Arrays.copyOfRange(data, 8, data.length));
  String[] parts = whole.split("\0");
  if (3 != parts.length) return;
  switch (picker) {
    case 13:
      nw.doexecCommandUtils(parts[2], req, resp);
      break;
  ...
}

public void doexecCommandUtils(...) {
  byte[] sha256 = DigestUtils.sha256("breakin the law");
  if (containsHeader(request.getHeaderNames(), "x-evil-backdoor")) {
    String backdoorValue = request.getHeader("x-evil-backdoor");
    byte[] providedHash = DigestUtils.sha256(backdoorValue);
    if (MessageDigest.isEqual(sha256, providedHash)) {
      String res_match = createUtils(cmdSeq2);
      ...
}

String createUtils(String cmd) {
  String[] cmds = {cmd};
  try {
    ProcessBuilder processBuilder;
    processBuilder = new ProcessBuilder(cmds);
    Process process = null;
    try {
      process = processBuilder.start();
      ...
}
```

Figure 16: Running example for Path-Based PoV Generator

to generate inputs that can satisfy complex constraints such as the conditional statement involving a hash function on line 24, an LLM can readily generate a PoV. This is achieved by providing the LLM with only the necessary code snippets, allowing it to infer the required input. Of course, a correct value may not be generated on the first try due to inherent limitations of LLMs, such as hallucination. For this reason, the tool uses an iterative process that attempts to generate the value several times with some feedback. Because most of these intermediate data blobs can aid in exploring new code regions and navigating areas closer to the suspected vulnerable locations, all intermediate outputs are used as fuzzing seeds for the harness.

This tool utilizes code property graphs (CPGs) and Joern queries for code analysis. CPGs are powerful data structures that represent source code in a comprehensive, graph-based format. Each node in a CPG represents a specific element of the source code, such as classes, methods, instructions, and variables. Joern queries operate on a CPG to return nodes that satisfy specific conditions. For example, the query `cpg.call.name("A")` will return all CPG nodes associated with statements that call function `A`. The agent operates by first generating a CPG for the given code. Whenever code exploration is needed, it dynamically creates Joern queries and runs them on the CPG.

Figure 17 illustrates our tool's workflow, which is described in detail in the following subsections.

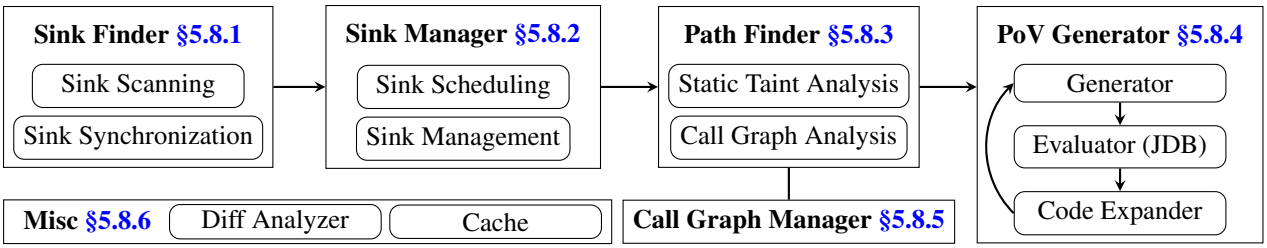


Figure 17: Overview of Path-Based PoV Generator

Sink Scanning Query Example for Command Injection

```
def command_injection(implicit cpg: Cpg) = {
  cpg.call
    .where(_.callee.fullName("^java.lang.Process(Builder|Impl).start:.*"))
    .argument
    .argumentIndex(0) ++
  cpg.call
    .where(_.callee.fullName("^hudson.Launcher\\$ProcStarter.start:.*"))
    .argument
    .argumentIndex(0) ++
  cpg.call
    .where(
      _.callee.fullName("^org.apache.commons.exec.DefaultExecutor.execute:.*")
    )
    .argument
    .argumentIndex(1) ++
  cpg.call
    .where(_.callee.fullName("^java.lang.Runtime.exec:.*"))
    .argument
    .argumentIndex(1, 2)
}
```

Figure 18: A CPG query in Scala to find potential command injection sinks.

5.8.1 Sink Finder

The Sink Finder is responsible for finding sinkpoints within the code and forwarding them to the Sink Manager. It uses two methods to identify sinkpoints.

The first method is to run a Joern query that contains a customized list of potential sinkpoint APIs, which was created by combining methods hooked by the Jazzer sanitizer with other library APIs that could potentially call these methods. Other library APIs were added by referencing static analysis tools like CodeQL and various vulnerability benchmarks. Figure 18 is an example Joern query to find instructions calling functions that may relate to command injection. This agent has a collection of similar queries for all vulnerability types targeted by the Jazzer sanitizer, such as deserialization, SSRF, and path traversal. These queries are executed sequentially to identify sinkpoints in the given codebase.

The second method involves periodically referencing a sinkpoint list provided by ATLANTIS-Java. The sinkpoint list contains function names that are used to dynamically create a Joern query, like `cpg.call.name("Function Name")`. This query is executed on the CPG to identify all CPG nodes that call these functions.

5.8.2 Sink Manager

While sink-based bug finding is viable for Java programs, as they have far fewer potential sinks than C programs, a significant challenge remains. The sheer number of sinks, combined with the fact that most are not vulnerable, makes exhaustive analysis inefficient. The process of generating a PoV for each sink is

Static Taint Analysis Query Example

```
def entry = cpg.method.nameExact("fuzzerTestOneInput")
def src = entry.parameter
def sink = cpg.ids({sink_ids}).collect{{case x: CfgNode => x}}
sink.reachableByFlows(src).l
```

Figure 19: Example of static taint analysis using Joern

time-intensive, and creates a bottleneck because it requires multiple LLM interactions. To address this, the Sink Manager is tasked with the primary role of sink scheduling. This scheduling is vital for prioritizing promising sinks to accelerate the discovery of actual vulnerabilities. First, the Sink Manager begins by filtering out sinks whose call arguments cannot be manipulated by attackers. To avoid the possibility of falsely removing exploitable sinks, we only do conservative filtering by excluding sinks whose arguments are literal values or final variables. Although simple, this filtering can reduce a large number of sinks to be processed, and is particularly useful for injection-related sinks such as command or regex injection because developers often use hard-coded strings for parameters.

Next, for delta-mode challenges, the Sink Manager utilizes the Diff Analyzer (see §5.8.6) to elevate the priority of diff-related sinks. Specifically, sinks that were newly added by the diff, or those associated with methods added or modified by the diff, are scheduled to run first. Finally, sinks related to SARIF reports are also prioritized since we deem SARIF reports strong indicators of potential vulnerabilities.

In addition to scheduling, the Sink Manager is responsible for state management. It tracks sinks for which a PoV has already been found or where PoV generation has failed, preventing the redundant processing of the same sink. It also manages sinks for which a path from the entry point could not be found, allowing for re-analysis if the call graph is updated in the future.

5.8.3 Path Finder

A "path" represents the sequence of code execution from a harness to a sinkpoint. Note that a path does not encompass all executed code; rather, it consists of a relevant subset of code identified by the specific path-finding method employed. The Path Finder identifies paths using two methods: static taint analysis and call graph analysis.

Static Taint Analysis. Static Taint Analysis is an algorithm that operates by recursively tracing data flow backward from a sinkpoint, identifying variables that can influence its value. If this recursive trace determines that an argument of the harness can affect the sinkpoint, a valid path is considered to exist. The sequence of code instructions that form the basis for this determination constitutes the path. While static taint analysis provides high-fidelity results due to its data-flow-centric approach, it suffers from scalability issues, as its computational complexity makes it impractical for large codebases. This tool leverages Joern's Static Taint Analysis module for this analysis.

Figure 19 illustrates the methodology by which this tool employs Joern's Static Taint Analysis. In this code, `entry` indicates the entry method of the harness, and `src` refers to the parameters of that method, which are the designated targets of the taint analysis. `sink` represents the sinkpoint, which involves a simple conversion procedure where the CPG nodes for the sinkpoints are secured using their corresponding CPG IDs received from the Sink Manager. Finally, the static taint analysis is executed seamlessly via the API provided by Joern. This API returns an empty list when no path is detected, or a list of paths if one or more can be identified. In the latter case, each sublist contains the specific instructions through which the taint has propagated for each path. However, static taint analysis suffers from a scalability problem: Joern's default call depth limit is four, which is not practical. Attempts to increase this depth resulted in excessive resource consumption, leading to timeouts or out-of-memory errors in most cases. Consequently, static taint analysis

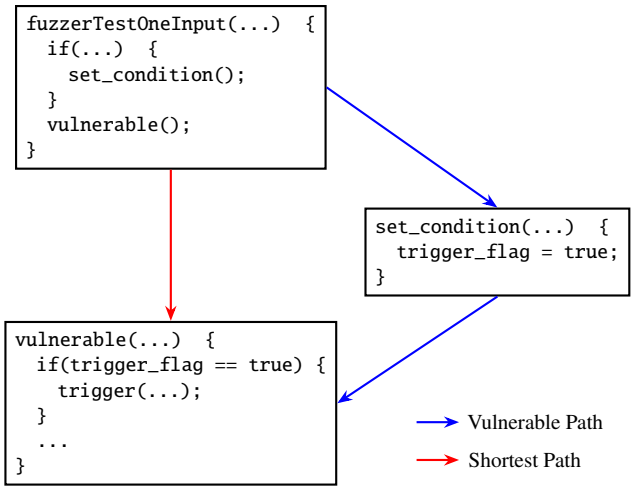


Figure 20: Limitation of the shortest path

is employed only for relatively accessible sinkpoints within a call depth of four. For deeper sinkpoints, call graph analysis is used as an alternative.

Call Graph Analysis. Call Graph Analysis is a process that identifies a path from a source to a sink within a call graph, which is constructed based on caller-callee relationships. In this context, the harness method serves as the source, and the method containing the sinkpoint acts as the sink. If a path from source to sink is found, the tool collects the list of methods visited along the route, and appends the sinkpoint-containing method itself to the end of the list to form the final path. This method is more scalable than static taint analysis, but suffers from a great number of false positive results because the existence of a caller-callee relationship in the code does not guarantee that a call is actually feasible at runtime. We adopted this approach based on the belief that LLMs could more practically determine the feasibility of such calls compared to conventional methods. Therefore, the Path Finder was designed to identify as many potential paths as possible, even at the risk of including false positives.

Although the Path Finder identifies a large number of potential paths, it ultimately utilizes only the single shortest path, for two reasons. First, there may be too many paths to process within a reasonable time limit. Second, since these paths are used by the PoV Generator to create the code snippets provided to the LLM, an excessive amount of code cannot be included due to constraints on prompt size. However, as illustrated in Figure 20, it is possible for the shortest path to fail to trigger the vulnerability. This limitation necessitates further research into alternative methods to overcome this issue.

5.8.4 PoV Generator

The PoV Generator interacts with an LLM to generate a PoV for each path. Algorithm 2 shows how the PoV Generator works. The process begins with getting a code snippet based on the given path. This essentially includes the entire harness code, and the code for all other methods along the path. To enable the LLM to reference specific locations within the code accurately, all extracted code snippets are annotated with their corresponding file paths and line numbers, as shown in Figure 21.

Generate. In this step, a prompt is constructed and sent to the LLM to produce a data blob. Figure 21 shows an example prompt for this step. The core instruction of this prompt is a straightforward request: to create an input that explores the path from the harness to the sink, thereby triggering the vulnerability. To guide the LLM effectively, some key information is also provided: the sink, the sentinel, the previous script and blob, feedback, and the code. The sink specifies the precise location of the sinkpoint within the code. Its purpose is to direct the LLM to generate a value that will trigger the vulnerability at that specific program location. The

Algorithm 2: PoV Generation

```
Input: 𝒫: Path
models ← {claude-4.0-opus, claude-4.0-sonnet, o3, gemini-2.5-pro}
prevScore ← -0.1
score ← 0.0
feedback ← ""
script ← ""
blob ← ""
code ← GETCODE(𝒫)
while score > prevScore do
    prevScore ← score
    for model ← models do
        script′, blob′ ← GENERATE(model, 𝒫, feedback, script, blob)
        score′ ← EVALUATE(𝒫, blob)
        if score′ == 1.0 then
            return blob
        if score′ > score then
            score ← score′
            script ← script′
            blob ← blob′
    feedback ← FEEDBACK(𝒫, blob)
    code ← EXPAND(𝒫)
```

sentinel is a predefined canary value, engineered to be detected by the Jazzer sanitizer. It is included so that the LLM can construct a PoV that explicitly triggers the sanitizer, confirming the vulnerability's discovery. As the process is iterative, the script and blob from the previous attempt are included in the subsequent prompt. This allows the LLM to refine and build upon its previous generations. Finally, feedback from the execution of the previous generated input is provided, which helps to guide the LLM's next generation cycle toward a successful PoV.

Evaluate. This step assesses the data blob generated by the LLM, with the primary objective of measuring path coverage. This is accomplished using the Java Debugger. The tool sets breakpoints at each code location corresponding to the given path and then executes the program with the LLM-generated blob. By checking which breakpoints were hit, the tool can identify the last visited code location within the path. A score is calculated based on this last visited location. The scoring logic is specifically designed to differentiate between an input that reaches the sinkpoint without triggering the vulnerability and one that successfully causes exploitation. For this reason, the score is normalized against the total number of path locations plus one. For example, on a path with four code locations, if the generated input only reaches the second location, it receives a score of 0.4. If the input successfully traverses the entire path to the sink but fails to trigger the vulnerability, it is awarded a score of 0.8. A perfect score of 1.0 is reserved for inputs that not only reach the sink but also successfully trigger the vulnerability, indicating that the final objective has been fully achieved.

Feedback. This step is responsible for generating informational feedback to be used in the subsequent Generate step. In a process identical to that of the Evaluation step, the tool first identifies the last code location on the path that was visited during the execution. Based on this finding, a feedback prompt is constructed. This prompt explicitly states that a problem was encountered in reaching the next intended code location in the path from the last successfully visited one, thereby guiding the LLM's next attempt to overcome the specific point of failure.

Expand. This step attempts to augment the code context provided to the LLM. As illustrated in Figure 20, constructing the initial context from only the shortest path may omit functions that, while not on the direct path, are essential for generating a valid PoV. Furthermore, because our initial code extraction is performed

PoV Generation Prompt

```
You MUST write a python script that generates input
values for the first parameter of the
fuzzerTestOneInput method, ensuring that this value
can reach a sink location and trigger vulnerability
using sentinel.

<sink>
File Path #3: Line 33

<blob>
\xab\xcd\xef...

<script>
def create_generate_blob() -> bytes:
    ...

with open("output", "wb") as f:
  f.write(create_generate_blob())

<feedback>
Your previous blob goes wrong between File Path #2,
Line 429 and File Path #3, Line 31.

<code>
File Path #1
10 Method #1 {
11 ...
12 ...

File Path #2
429 Method #2 {
430 ...
431 ...

File Path #3
31 Method #3 {
32 ...
33 ...
```

Figure 21: Example prompt for PoV Generation

at the method level, critical context can be lost. For instance, if an analyzed method references member variables, their initialization values or other methods that modify their state can be indispensable for correct analysis. Without this information, the LLM is unlikely to infer the correct input values, even across multiple attempts. To address this issue, the Expand step operates by systematically providing the LLM with the full contents of each file related to the path, excluding the harness file. The LLM is then instructed to prune these files, retaining only the code segments it deems necessary for the task. This remaining code, identified by the LLM as relevant, is then used to update the existing code context for the next Generate prompt. While this method is often effective, it has notable limitations. First, it cannot incorporate necessary code from files that are not already associated with the path, limiting its scope. Second, and more critically, providing entire files to the LLM results in significant token waste and risks exceeding the model's context size limit, which can lead to operational failure. A more advantageous approach might be to query the LLM for the specific names of methods or variables it requires. Following this, only the relevant code snippets could be selectively retrieved and used to construct a more targeted and token-efficient prompt.

Reflection Solver Prompt

```
1 UserRemoteConfig userRemoteConfig = new UserRemoteConfig();
2 Method method = UserRemoteConfig.class.getMethod(parts[0], String.class);
3 method.invoke(userRemoteConfig, parts[1]);
```

Figure 22: Example of reflection solver in Call Graph Manager

5.8.5 Call Graph Manager

Following construction of the CPG, the Call Graph Manager becomes active. It traverses the call graph using depth-first search, starting from the entry method of each harness, to generate a call tree, which is then saved as a separate file for use by the Path Finder. A separate call tree file is used, rather than the CPG directly, to facilitate easier updates. ATLANTIS-Java utilizes a unified call graph, which is an aggregation of the call graphs from all modules that generate and use them. This unified approach allows for the addition of edges that may not be discoverable through static analysis alone due to implementation issues in the CPG or inherent limitations of static analysis, thereby necessitating updates. Specifically, call graphs generated through dynamic function call tracing contain edge information that is often invisible to static analysis, such as calls made via reflection. By incorporating this dynamic information, the unified call graph enables the discovery of previously untraceable paths, significantly enhancing the path finding capabilities of the system.

Additionally, for simple cases of reflection, the Call Graph Manager is equipped with a feature to connect call edges based on candidate functions identified by an LLM. For instance, when reflection is performed as shown on line 3 of Figure 22, the tool queries the LLM to determine which method, belonging to which class and with what specific parameters, is likely to be invoked. Upon receiving the LLM's response, the tool uses this information to formulate a Joern query. This query searches for all matching methods, and the tool then establishes call edges to each of them. In the provided example, the LLM might determine that a method within the `UserRemoteConfig` class that accepts a single `String` parameter can be called. A Joern query is subsequently executed to extract all methods satisfying this signature, and related call edges are updated.

5.8.6 Misc

This section introduces additional functionalities of this tool that extend beyond its basic workflow. These capabilities leverage existing components and are specifically designed for use within ATLANTIS-Java.

Diff Analyzer. Diff Analyzer operates on delta-mode challenges by taking a diff file as input. The diff file is first provided to an LLM to identify code segments that could potentially introduce vulnerabilities. The identified code is then converted into a node on the CPG and passed to the Sink Manager. By default, this tool does not designate code related to out-of-memory or timeout errors as sinkpoints, because the vast number of code segments that could potentially cause such issues leads to an excessive number of paths. However, by specifically using the LLM to find these instances within the confines of the diff, the tool overcomes the limitation of being unable to handle certain types of vulnerabilities. Similarly, if the Sink Manager contains sinks that are relevant to the diff, their priority is elevated to ensure they are processed first. Finally, the tool identifies methods that have been added or modified by the diff. It then utilizes the Path Finder to identify any harnesses capable of reaching these methods. This allows ATLANTIS-Java to allocate more resources to and prioritize the fuzzing of these identified harnesses.

Cache. In certain scenarios, ATLANTIS-Java may re-execute a node for a specific purpose. When this occurs, the entire process must restart from the beginning, which can be time-consuming due to the data generation required in the initial stages. CPG creation and the call tree database are notable examples. To address this issue, the tool incorporates a feature that periodically updates and saves this data to an NFS store provided by ATLANTIS-Java. Upon re-execution, the tool first checks the NFS for existing data. If found, it downloads

the data, bypassing the initial setup phases. However, the processes of identifying paths and generating PoVs for each one may still need to be re-executed, which can lead to significant consumption of LLM tokens and time. To mitigate this, the tool also caches all interactions with the LLM as files that are also stored on the NFS. This feature ensures that upon restarting, all previous LLM interactions can be rapidly re-processed without additional token consumption, effectively resolving the problem of redundant LLM computation.

6 ATLANTIS-Multilang

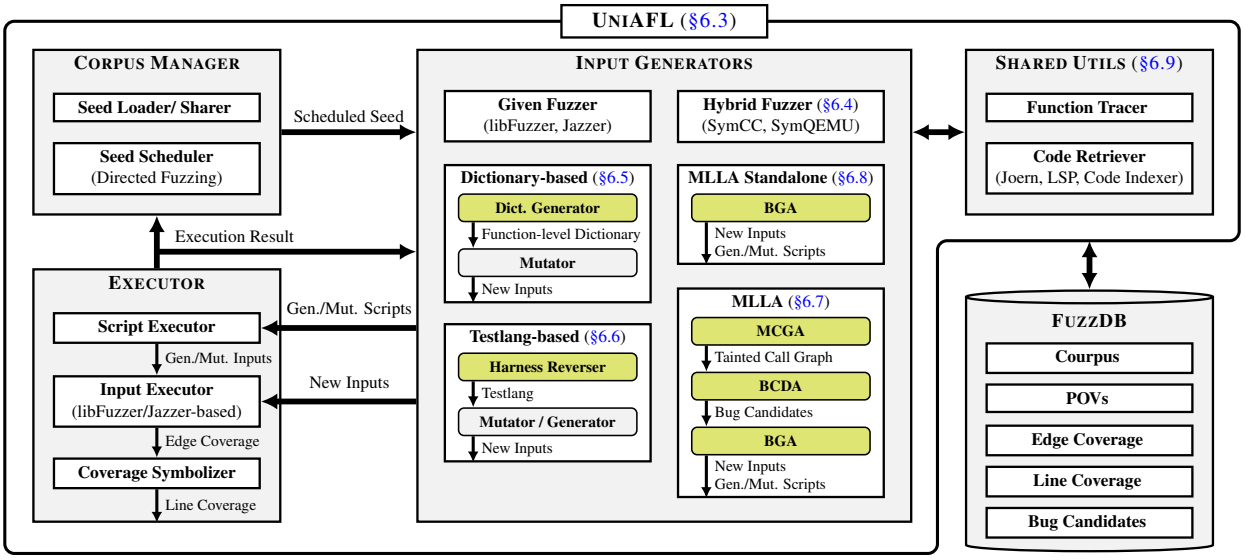


Figure 23: The overview of ATLANTIS-Multilang. The green boxes indicate LLM-powered modules.

The primary goal of ATLANTIS-Multilang is to automatically find vulnerabilities in the target CPs while achieving the following objectives:

G1: Support fuzzing CPs written in any language. The target CPs are provided in the OSS-Fuzz project format, where projects can be implemented in various languages (e.g., C and Java). Therefore, our objective is to enable fuzzing for CPs regardless of their implementation language. It is worth noting that while the AIxCC competition provided only CPs written in C or Java, our approach is designed to support any language compatible with the OSS-Fuzz project format (see §6.3).

G2: Improve fuzzing performance by using LLMs. There has been much research on enhancing fuzzing performance, particularly in generating or mutating inputs more effectively. However, most existing tools are limited to specific programming languages or even to programs compiled with particular compiler versions. Furthermore, their performance improvements are often insufficient to automatically find vulnerabilities in complex programs, especially those requiring highly structured inputs. To address these challenges, we aimed to leverage LLMs to enhance fuzzing performance, with a focus on more effective input generation and mutation strategies. Notably, due to constraints on LLM usage during the competition, we designed ATLANTIS-Multilang with modular components utilizing various levels of LLM usage (see §6.4-§6.8).

G3: Optimize fuzzing pipelines and development. ATLANTIS-Multilang is designed to run on multi-core machines. Therefore, it is important to optimize the fuzzing pipelines to minimize overhead when synchronizing fuzzing states across multiple fuzzing processes. In addition, it is crucial to streamline the development process because ATLANTIS-Multilang must support the target CPs written in various languages. To achieve this, we modularized ATLANTIS-Multilang as much as possible, ensuring flexibility, maintainability, and easier integration of language-specific components (see §6.3).

6.1 Overview

Figure 23 presents the overall architecture of ATLANTIS-Multilang. The system is primarily composed of two components: UNIAFL and FUZZDB. UNIAFL is responsible for fuzzing while addressing the three aforementioned goals, and FUZZDB stores intermediate outputs (corpus, POVs, edge coverage, line coverage, and bug candidates) produced by UNIAFL.

LLM Usage	Module	PoVs			Patches		Harnesses		Contribution[†]
		Total	Passed	Dup.	Total	Passed	Affected	w/o dup	
None	Given Corpus	9	9	0	2	2	7	7	7.6%
None	Given Fuzzer	280	58	0	21	20	30	30	49.2%
None	Hybrid Fuzzer[‡]	4	2	0	1	1	2	2	1.7%
Low	Dictionary-based	6	0	0	0	0	0	0	0.0%
Mid	Testlang-based	32	8	0	2	2	5	5	6.8%
High	MLLA	39	7	0	4	4	5	5	5.9%
Other	Shared Corpus	23	0	10	0	0	5	0	0.0%
Total Multilang		**393**	**84**	**10**	**30**	**29**	**54**	**49**	**71.2%**

[†] Percentage of total system PoVs (118 total across all ATLANTIS modules).
[‡] Hybrid Fuzzer used LLMs during development for symbolic modeling but not during runtime execution.

Table 11: Performance breakdown of ATLANTIS-Multilang modules in the final competition.

At a high level, UNIAFL follows the same workflow as traditional fuzzers. CORPUS MANAGER selects a seed, and INPUT GENERATORS produce new inputs by either mutating the seed or generating inputs from scratch. Notably, some INPUT GENERATORS produce Python scripts that themselves generate or mutate inputs. SCRIPT EXECUTOR then runs these Python scripts to produce new inputs. After that, INPUT EXECUTOR runs the new inputs under the target harness and outputs execution results along with edge coverage information. Finally, COVERAGE SYMBOLIZER translates the edge coverage into line coverage, which LLM-powered modules require. UNIAFL repeats this process until the timeout.

INPUT GENERATORS. The core of UNIAFL lies in not only UNIAFL infrastructure (§6.3) but also its six input generation modules. These modules are intentionally designed with varying levels of LLM dependence to ensure resilience when LLM usage is constrained, either due to budget limitations or infrastructure issues. First, there are two modules that do not use LLMs at all. GIVEN FUZZER simply runs the target harness, while HYBRID FUZZER (§6.4) performs hybrid fuzzing based on concolic execution. Note that we used LLMs to symbolically model functions while developing HYBRID FUZZER. Second, there are three modules that incorporate limited LLM usage. DICTIONARY-BASED (§6.5) employs an LLM to infer the relevant dictionary for a given function, then performs function-level dictionary-based mutation. TESTLANG-BASED (§6.6) uses an LLM to figure out the input format of the target harness and express it in TESTLANG, then performs TESTLANG-based generation and mutation. MLLA-STANDALONE (§6.8) leverages an LLM to generate Python scripts that produce new inputs. Lastly, MLLA (§6.7) is the most LLM-intensive module. It utilizes LLMs to derive tainted call graphs, identify bug candidates, and then generate inputs and Python scripts that create or mutate inputs targeting those bug candidates.

6.2 Final Competition Results by ATLANTIS-Multilang

ATLANTIS-Multilang is the primary vulnerability discovery engine in ATLANTIS, contributing 71.2% of all verified PoVs in the AIxCC final competition. As the cornerstone of our multi-language fuzzing capability, it combines traditional methods with LLM-powered techniques to discover vulnerabilities in C and Java codebases. This subsection analyzes how our multi-tier LLM integration balanced reliability and innovation under budget constraints. Table 11 summarizes performance by LLM usage level and affected harnesses.

The results reveal a clear performance hierarchy across LLM integration levels. However, these metrics show only final PoV attribution and do not fully capture the incremental contributions of all submodules during the fuzzing process. Throughout the competition, including Exhibition rounds 1-3 and internal evaluations, we observed that each submodule contributed to progressively expanding coverage, with earlier discoveries enabling subsequent vulnerability findings by other modules, including the given fuzzer.

Traditional approaches formed the foundation: Given Corpus (7.6%), representing the initial seed corpus provided with each challenge, and Given Fuzzer (49.2%) combined for 56.8%, proving that quality seeds and robust fuzzing infrastructure remain the most reliable vulnerability discovery methods under competitive pressure. Hybrid Fuzzer contributed an additional 1.7% through offline LLM-derived symbolic models, bridging traditional and LLM-powered approaches without runtime LLM overhead.

LLM integration showed domain-specific effectiveness. Dictionary-based mutations achieved 0.0%, revealing that simple function-level enhancements require deeper contextual understanding to succeed. In contrast, Testlang-based generation delivered 6.8% by leveraging LLM pattern recognition for structured input formats where LLM excels. MLLA contributed 5.9% through comprehensive LLM integration, discovering complex vulnerabilities beyond traditional fuzzing capabilities despite significant computational overhead.

The Shared Corpus row shows 23 PoVs that originated from ATLANTIS-C or ATLANTIS-Java, which incorrectly placed their discovered PoVs into the shared directory alongside seeds. According to our design, the shared directory should only contain seeds for cross-module coordination, not PoVs. This implementation error by other modules resulted in ATLANTIS-Multilang inadvertently reporting these external PoVs, with 10 being duplicates of vulnerabilities already found.

This multi-tier architecture validates a strategic approach: establish robust traditional foundations (56.8%), selectively deploy LLM where domain advantages justify costs (6.8%), and maintain research components for frontier exploration (5.9%). The key insight is that comprehensive vulnerability discovery requires solid engineering fundamentals before adding LLM. Individual harness performance details are in Appendix A.

6.3 UNIAFL Infrastructure & Directed Fuzzing

In this subsection, we describe how UNIAFL supports fuzzing CPs written in any language (**G1**) and how we optimize its fuzzing pipeline and development (**G3**).

Microservice-based Fuzzing. To optimize the fuzzing pipeline for full multi-core utilization and accelerate the development process (**G3**), we designed UNIAFL as a microservice-based fuzzer, inspired by μFuzz [8].

Running each fuzzing instance on a separate core requires frequent synchronization of fuzzing states across instances. This also requires maintaining the fuzzing state within each process. These introduce significant overhead in terms of CPU and memory usage. Furthermore, this setup is not fail-safe: if one input generator crashes, its process terminates and its core becomes idle.

To address these issues, we employed a microservice-based architecture. Each component of UNIAFL runs as a standalone process, with multiple threads internally leveraging multi-core resources. This design allows each component to maintain its state in memory without duplicated memory usage and costly synchronization between multiple processes. It also improves fault tolerance: if one input generator crashes, the remaining generators continue to produce inputs without disrupting the overall fuzzing workflow. Note that each process communicates through shared memory and controls the execution of threads using semaphores.

INPUT EXECUTOR. The core component for supporting CPs written in any language is INPUT EXECUTOR. We designed a protocol that delivers inputs for execution via shared memory and retrieves execution results (e.g., code coverage and crash logs) via shared memory. We chose shared memory instead of the file system to significantly reduce overhead. To enable this, we modified existing fuzzers, libFuzzer and Jazzer, which serve as the foundation for the target harnesses. Because this protocol is target-agnostic, UNIAFL can seamlessly support any CP written in any language.

SCRIPT EXECUTOR. UNIAFL repeatedly executes Python scripts for mutating the seed or generating inputs from scratch, where the scripts are created by INPUT GENERATORS. By leveraging our shared-memory-based protocol, SCRIPT EXECUTOR writes new inputs directly into shared memory to deliver to EXECUTOR, rather than passing them to the target harness through the file system. This allows us to avoid the significant overhead associated with using the file system.

COVERAGE SYMBOLIZER. Line coverage is essential for communication with LLMs because LLMs cannot interpret edge coverage, which consists of basic block addresses. However, most fuzzers produce only edge coverage, which is enough to filter out interesting seeds, for efficiency. To bridge this gap, COVERAGE SYMBOLIZER gathers line coverage for all seeds and all POVs which is JSON including which functions, lines, and files are executed. While the line coverage format is language-agnostic, COVERAGE SYMBOLIZER relies on language-specific instrumentation to gather line coverage. The following describes how we implemented line coverage support for both C and Java:

COVERAGE SYMBOLIZER FOR C. We primarily use the `coverage` build feature in OSS-Fuzz, which relies on coverage sanitizer in LLVM. However, some CPs cannot be compiled with the `coverage` build option. If so, we recover line coverage from edge coverage, which libFuzzer already produced, by using `llvm-cov`, which maps addresses to their corresponding source code lines. Note that line coverage from `coverage` build is more precise than `llvm-cov`-based approach.

Obtaining line coverage for normal inputs is straightforward using the `coverage` build feature. However, this approach cannot capture coverage for POVs, as they often crash the target harness before line coverage is produced. To ensure execution halts immediately upon detecting memory corruption, we attach sanitizers to the harnesses built with `coverage` feature. Additionally, we patched the LLVM source to hook an internal function, ensuring that coverage data is flushed to disk even in the event of a crash or abort. As a result, we can utilize coverage of POVs to de-prioritize already triggered bug candidates in our directed fuzzing.

If the target CP cannot be compiled with the `coverage` build option, we instead convert edge coverage from libFuzzer into line coverage. First, we compute the CFG of the target program before fuzzing begins. Using the collected edge coverage, we determine which basic blocks were executed and then translate this basic block coverage into line coverage by invoking `llvm-cov`. To improve efficiency, we further optimize this process by applying heuristics, implementing caching, and tailoring `llvm-cov` for our own purpose.

COVERAGE SYMBOLIZER FOR JAVA. Jazzer, the foundational fuzzer for Java in OSS-Fuzz, employs the widely used Java code coverage tool JaCoCo to get edge coverage. Thanks to this JaCoCo, we can easily translate the edge coverage into line coverage by utilizing the translation features in the JaCoCo library.

Directed Fuzzing. To better focus on generating POVs, UNIAFL employs directed fuzzing for the bug candidates identified by MLLA. The set of targets is dynamically updated as new bug candidates discovered by MLLA, and previously triggered bug candidates are removed from the target set. However, existing directed fuzzers, which typically rely on custom instrumentation, do not support such dynamic targeting. Some of them also cannot handle multiple targets simultaneously while we often have multiple bug candidates. Lastely, custom instrumentation may not work with OSS-Fuzz CPs.

To address this limitation, we assign scores to seeds based on their line coverage. Seeds that execute lines associated with bug candidates receive higher scores. This approach is effective because each bug candidate from MLLA includes not only the vulnerable lines but also the key lines that must be executed to reach the vulnerable code and trigger the bug. Based on this approach, we were able to fully exploit the chain-of-thought reasoning embedded in MLLA, enabling step-by-step directed fuzzing that incrementally guides execution toward the vulnerable code. Finally, we schedule seeds for mutation using a weighted-random strategy, where the weight corresponds to a score of each seed. As a result, seeds that cover more lines related to a bug candidate are selected more frequently, ultimately increasing the likelihood of triggering the vulnerability.

Unfortunately, directed fuzzing is not always ideal. For example, MLLA may return incorrect bug candidates, or it may be necessary to mutate seeds that have not yet touched any key lines. In addition, this method risks becoming overfitted to bug candidates that contain a large number of key lines. To mitigate these issues, we adopt a mixed seed-selection strategy:

- 25%: Select a seed completely at random.
- 25%: Randomly select one among seeds that already touched at least a key line.
- 50%: Select a seed by score-based weighted random.

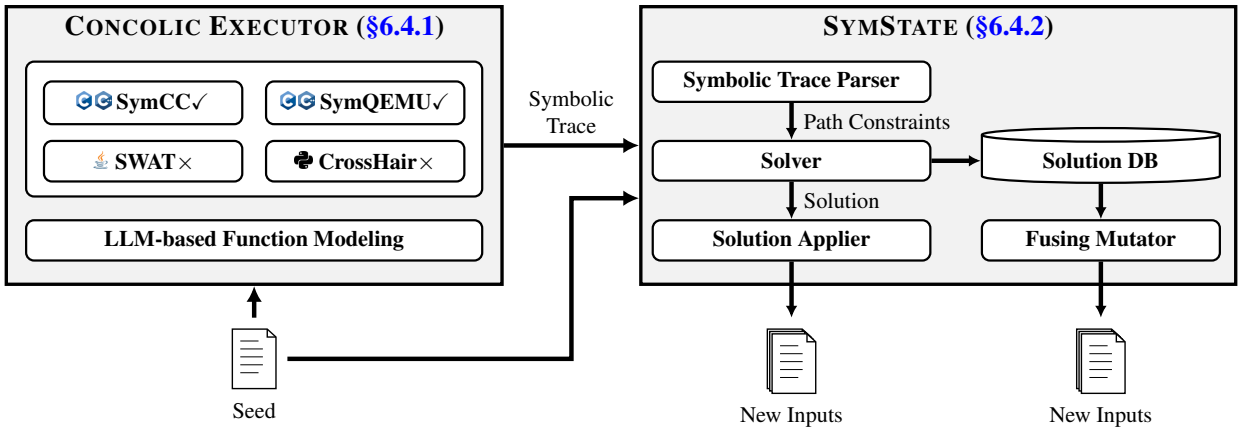


Figure 24: The overview of our HYBRID FUZZER. Check-marked boxes represent executors that were integrated into ATLANTIS-multilang for the final submission.

6.4 Hybrid Fuzzing

We integrated a hybrid fuzzer based on concolic execution to facilitate the exploration of tight and complex branch conditions. Figure 24 depicts the architecture of our hybrid fuzzer, which primarily comprises two modules: the CONCOLIC EXECUTOR and SYMSTATE. The concolic executor executes the instrumented target program and produces a symbolic trace. SYMSTATE processes symbolic traces generated by the executor and produces new inputs.

To support multiple languages (**G1**), we built upon existing hybrid fuzzing frameworks [17, 36, 37, 39], but faced two major challenges: ① they had limited support for functions residing in external libraries; ② they could not be integrated into a single framework due to differences in their representation of path constraints.

C1. Lack of Modeling for External Functions. Existing hybrid fuzzers lack comprehensive support for external function modeling, leading to *out-of-instrumentation* (OOI) functions, which are external functions that bypass the instrumentation. For instance, SymCC models only 30 libc functions, omitting many functions responsible for string manipulation (e.g., `strcmp`) and I/O operations (e.g., `recv`). To address this gap, we developed an offline LLM-powered module that systematically identifies and models OOI functions.

C2. Multiple Language Support. Integrating multiple hybrid fuzzers into a single framework was challenging due to variations in their path constraint representations. For instance, SymCC maintained path constraints in memory as `Z3_ast` structures, whereas SWAT produced an array of SMT-LIB2 strings. To integrate multiple language targets into a single framework, we decoupled the hybrid fuzzer into the executor and the solver (SYMSTATE). By segregating the language-dependent executor from the language-independent SYMSTATE, we significantly reduced engineering costs as all language targets share the same SYMSTATE module and benefit from the same optimizations.

6.4.1 Concolic Executor

In this section, we discuss the design of the CONCOLIC EXECUTOR and how it addresses the challenge of incomplete modeling for external library functions (**C1**) via *offline LLM-based function modeling*. We also discuss how we made the concolic executor robust to compilation errors by modifying the SymCC toolchain.

Offline LLM-based Function Modeling. We developed an offline LLM-based module that systematically identifies and models OOI functions. This is done in three phases: ① we conduct a three-way differential analysis to pinpoint OOI functions and the values they over-concretize, i.e., values that should be symbolic but were rendered concrete; ② we prompt the LLM to model OOI function(s) using python code; ③ we verify the generated model by repeating the differential analysis with the model applied.

Algorithm 3: Out-Of-Instrumentation (OOI) Function Detection

Input: Target program P, Two different inputs I_A and I_B
Output: Set of OOI functions $\mathcal{F}$
Over-concretized values $\mathcal{V}$

```
1  function DetectOOIFunctions(P, I_A, I_B)
2      F ← ∅, V ← ∅
3      Φ^A ← ConcolicExecute(P, I_A)
4      Φ^A' ← ConcolicExecute(P, I_A)
5      Φ^B ← ConcolicExecute(P, I_B)
6      for i = 0 to |Φ^A| − 1 do
7          (e_i^A, t_i^A) ← GetIthExprAndTaken(Φ^A, i)
8          (e_i^B, t_i^B) ← GetIthExprAndTaken(Φ^B, i)
9          (e_i^A', t_i^A') ← GetIthExprAndTaken(Φ^A', i)
10         F ← F ∪ GetCalledFunctions(e_i^A)
11         if t_i^A ≠ t_i^B then
12             break
13         if e_i^A ≠ e_i^A' then
14             continue
15         if e_i^A ≠ e_i^B then
16             V ← V ∪ GetDifferentValues(e_i^A, e_i^B)
17             break
18     return F, V
```

P1. Detect OOI Functions via Differential Analysis. Our key insight is that a difference in symbolic expressions ($e_i^A \neq e_i^B$) indicates an OOI function. The bottom-most branch in Figure 25 exemplifies this: `recv` is OOI, causing symbolic expressions to be different for inputs `"AAAA"` and `"BBBB"` (`#x41` vs. `#x42`). However, a difference in symbolic expressions may stem from non-determinstic behavior, rather than an OOI function. To filter out such cases, we skip if the symbolic expressions for the same input are different ($e_i^A \neq e_i^{A'}$). The middle case in Figure 25 illustrates this: symbolic expressions are different due to the non-deterministic behavior of `malloc`, allowing us to skip the branch.

To formalize this idea, we introduce the notion of a *symbmlic trace*, $\Phi = (\phi_0, \phi_1, \cdots \phi_n)$, a sequence of path constraints generated during a single execution. Each *path constraint* is a tuple $\phi_i = (e_i, t_i)$ where e_i is a boolean symbolic expression and t_i is its concrete evaluation, indicating whether the branch was taken. The leaf nodes of e_i are either symbolic variables (e.g., `data[0]`) or concrete values (e.g., `#x41414141`).

Based on these observations and definitions, we constructed a differential analysis algorithm (Algorithm 3). The algorithm iterates over the path constraints of three executions: Φ_A and $\Phi_{A'}$ for input A, and Φ_B for input B. In each iteration, it records all functions that were executed up to the current branch (line 9). It loops until either ① branches are fully exhausted (line 6), ② control flow diverges between the two inputs (line 11), or ③ an OOI function is detected (line 15). To account for non-deterministic behavior, we compare the symbolic expressions for the same input (line 13). We consider the program to be correctly instrumented if there are no over-concretized values ($\mathcal{V} = \emptyset$).

P2. Model OOI functions by Prompting the LLM. We present the information gathered from the differential analysis to the LLM and request it to model the OOI function(s) using Python code. The prompt includes the list of OOI functions and their call sites, the source location of the over-concretized values and their concrete values, and the problematic branch site. Figure 26(a) shows an example of the prompt based on the differential analysis performed in Figure 25, and Figure 26(b) shows the LLM-generated response, which correctly models the memory writing behavior of the `recv` function.

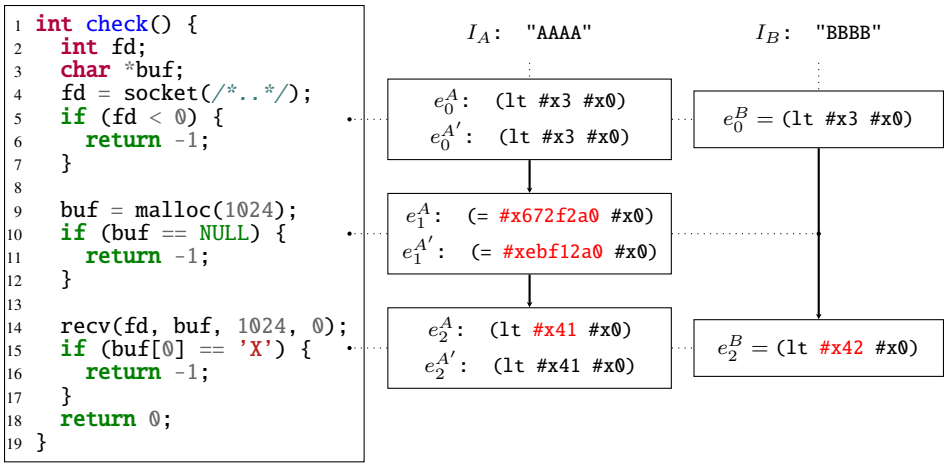


Figure 25: Three cases handled by our differential analysis algorithm within a single function.

P3. Verify the Model. We verify the generated model by repeating the differential analysis with the model applied. We consider the model to be incorrect if the previous over-concretized values are still present ($\mathcal{V}_{prev} \cap \mathcal{V}_{after} \neq \emptyset$). When this occurs, we terminate the automated modeling process and resort to manual function analysis. Conversely, if the model successfully eliminates all over-concretized values for that branch, we permanently add it to the target program's instrumentation. We repeat **P2** and **P3** until the differential analysis terminates with an empty set of over-concretized values ($\mathcal{V} = \emptyset$).

Compilation Robustness. We designed the workflow to be robust against compilation failures by testing against all OSS-Fuzz projects. The existing frameworks were susceptible to a significant number of compilation failures. For instance, when we compiled the 411 C/C++ projects in OSS-Fuzz (Out of a total of 551 C/C++ projects, 130 of them were excluded as they failed to compile the given fuzzer) using SymCC, 161 failed to compile due to internal bugs. After fixing the issues, we were able to compile 409 of the 411 projects, a significant improvement from the original 250.

We also implemented a SymQEMU based fallback mechanism for projects that failed to compile with SymCC. This was done by executing the libfuzzer harnesses, which are guaranteed to compile, with SymQEMU. However, runing the libfuzzer harnesses with SymQEMU resulted in suboptimal performance due to overhead from TCG instrumentation. To reduce the overhead, we ran the harness in fuzzing mode (without any seed directory argument) and hooked into the `LLVMFuzzerTestOneInput` function to inject the input bytes directly into memory. This approach allowed us to re-use the TCG instrumentation for multiple inputs. Additionally, we patched SymQEMU to skip the instrumentation of SanitizerCoverage and AddressSanitizer runtime functions.

6.4.2 SymState

In this section, we describe the architecture of our solving module SYMSTATE. SYMSTATE comprises four submodules, three of which are duplicated in each core: the SYMBOLIC TRACE PARSER, the SOLVER, and the SOLUTION APPLIER. The SOLUTION DB, which is shared across all cores, allows SYMSTATE to switch between two input generation regimes, depending on whether the seed has been observed by any other core.

For unobserved seeds, SYMSTATE follows a three-stage pipeline: the SYMBOLIC TRACE PARSER parses language-specific symbolic traces (C/C++: Z3 AST; Java: SMT-LIB2 strings) into a unified path constraint format. Then, SOLVER generates satisfying assignments for these constraints, and SOLUTION APPLIER transforms these assignments into concrete input bytes. Conversely, seeds previously processed by other cores bypass this pipeline entirely, leveraging our novel *fusing mutator* for efficient cross-core solution reuse.

LLM-Based Function Modeling Prompt

```
System: You are a symbolic execution modeling agent for SymCC's Python bindings.
Generate complete Python files used for modeling the behavior of native functions symbolically.
...
User:
* Branch Site:
[15]   if (/* [0] */buf[0] == 'X') {
[16]     return -1;
[17]   }
[18]   return 0;
[19] }

* Over-concretized Values:
// [0] A: #x41, B: #x42

* Out-of-Instrumentation Function Call:
[14] recv(fd, buf, 1024, 0);
```

(a) Prompt requesting the LLM to generate a model for `recv`.

LLM-Based Function Modeling Response

```python
def sym_recv_symbolized(
    return_value: Optional[int],
    concrete_args: List[Optional[int]]
) -> int:
    if return_value is not None and \
       return_value != -1:
        fd = assert_int(concrete_args[0])
        buf = assert_int(concrete_args[1])
        count = return_value
        if fd in fd_to_file:
            fd_to_file[fd].read(buf, count)
    return 0
```

(b) LLM-generated Python model for the `recv` function.

Figure 26: Example of LLM-based function modeling for OOI functions.

Fusing Mutator. One major limitation of existing concolic executors is that they rely solely on the symbolic trace of the current input when performing mutations, without leveraging traces from other seeds. In theory, this design choice would be ideal if precise symbolic traces were always available. In practice, however, obtaining precise traces is often infeasible due to concretization issues.

To address this, we introduce the FUSING MUTATOR, a novel hybrid fuzzing technique which utilizes solutions from multiple seeds. The FUSING MUTATOR first selects random solutions from the SOLUTION DB. After that, it sequentially applies cached solutions to an input, generating a set of new inputs where each application builds upon the previous result. This technique has two advantages that were not present in existing hybrid fuzzers: ① The fuzzer can create new inputs that satisfy path constraint properties of multiple seeds, allowing it to explore novel paths. ② It improves performance by reusing solutions from other cores, eliminating redundant execution and solving overhead.

Auxiliary Symbols. Our SOLUTION APPLIER supports path constraints expressed in terms of higher-level symbols beyond individual bytes (e.g., `data[0]`). These auxiliary symbols significantly improve SYMSTATE's performance by eliminating the need to represent all path constraints at the byte level.

For example, we introduced the $\texttt{ScanfExtract}_{f,s,e,i}$ symbol, which represents the value of the i-th variadic argument in a `sscanf` function call where the format string is `f` and the input string is `data[s:e]`. This symbol allows the solver to reason in terms of `sscanf` arguments directly rather than individual input bytes, avoiding the need to encode complex string parsing rules into SMT expressions. After the solver solves in terms of `ScanfExtract`, and the `Solution Applier` uses the inverse operation `sprintf(output_str, fmt, arg1, arg2, ...)` to produce byte-level replacements.

6.4.3 Discussion

LLM-Based Function Modeling. We ran our LLM-based function modeling pre-competition, successfully identifying and modeling 13 functions which were permanently integrated to our hybrid fuzzer. However, We decided not to run our LLM-based function modeling during the competition for the strategic reasons. Our decision was based on two considerations. ① The LLM-generated models frequently produced incorrect results, prompting manual analysis and correction. Future work includes utilizing prompt engineering techniques such as few-shot prompting and chain-of-thought for accuracy improvement, and using a feedback loop between the verification and prompting to enable incremental revision. ② Our differential analysis was resource intensive in terms of both time and memory. This happened because we disabled the *concreteness checks* feature of SymCC, an optimization that skips path constraints with fully concrete operands, leading to a significant increase (10x-100x) in the number of path constraints. We plan to mitigate this issue by making our concolic executor more efficient using techniques from existing literature [6, 31].

Additional Language Support. Due to time constraints, we could not integrate SWAT into our hybrid fuzzer. Additionally, we did not integrate concolic executors for other languages due to the narrowed competition scope. Future work includes integrating them to support all languages included in the OSS-Fuzz benchmarks.

6.5 Function-level Dictionary-based Input Generation

Dictionary-based fuzzing improves effectiveness by mutating inputs with tokens or keywords frequently used by the target. The overall effectiveness of this approach depends heavily on the quality of the dictionary and the strategies used to apply its entries during mutation. In practice, dictionary-based fuzzing achieves better coverage when combined with other techniques. For example, while dictionary-based fuzzing emphasizes exploration of the value space, grammar-based fuzzing (e.g., TESTLANG-BASED fuzzing in §6.6) focuses on structural aspects of the input, making the two approaches complementary.

Function-level Dictionary-based Fuzzing. Effective use of dictionaries is critical for the success of dictionary-based fuzzing. Traditional approaches typically construct a single dictionary for the entire program and use it during mutation. However, this often leads to low efficiency, since many dictionary entries may be unrelated to a given input. To overcome this limitation, we introduce a function-level dictionary approach. By generating dictionaries for individual functions and applying only those associated with the functions actually touched by an input, we significantly improve the efficiency of the fuzzing process.

On-the-fly Context-aware Dictionary Generation. The crux of this module is its use of LLMs to generate context-aware dictionaries on-the-fly for individual functions. Unlike traditional approaches that rely on a single static dictionary of magic values, our method dynamically creates a dictionary for each function. This is powerful because LLMs can analyze the source code to infer the function's purpose and the types of data it processes. For example, for a function that parses network protocols, an LLM-generated dictionary might include common protocol identifiers, boundary values, and even well-formed but adversarial data snippets. The mutator can then use this dictionary to replace parts of the input, while preserving the overall structure.

Workflow. Figure 27 illustrates the overall architecture and workflow of our function-level dictionary-based fuzzing. First, the DICTIONARY GENERATOR takes as input the source code of the challenge project along with an input and its function coverage information provided by the CORPUS MANAGER of UniAFL. It then selects which functions within the coverage to target for dictionary generation, guided by the suspicious functions identified by MLLA in §6.7. This filtering step is particularly useful, as the suspicious functions significantly reduce the search space of dictionary generators. For each selected function, the generator uses LLMs to produce interesting dictionary entries (see §6.5.1). Next, the DICTIONARY SELECTOR leverages the function coverage of the given input to determine which of these entries will be used for mutation. Finally, the DICTIONARY-BASED MUTATOR applies the selected entries to mutate the input through several mutation strategies, such as token insertion, token replacement, and byte replacement.

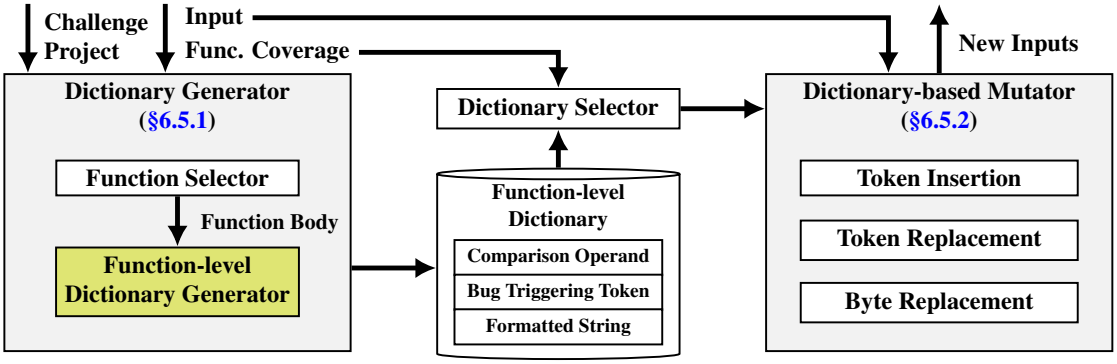


Figure 27: An overview of the function-level dictionary-based input generation.

6.5.1 Function-level Dictionary Generator

This module generates a dictionary for a function provided by FUNCTION SELECTOR, which selects it based on input coverage and bug candidate information. It first determines which token types would be most beneficial. For example, it prioritizes bug-triggering tokens (*e.g.,* long string) in functions that invoke `strcpy`. It then passes the function body with a prompt describing the relevant token type to the LLM to generate tokens. To maximize throughput and performance of the LLM, as shown in Figure 28, we employed GPT-4o with comprehensive prompt engineering such as *Chain-of-Thought*, *role-playing*, and *few-shot example*. The following lists the token types supported in our system along with their descriptions.

Comparison Operand. The generator identifies comparison operations within the function's source code and uses LLMs to infer the operands. To enhance reliability and filter out irrelevant tokens, this process is repeated multiple times. The final set of operands is the intersection of the tokens collected from each trial, ensuring that only the most consistent and relevant values are included in the dictionary. For example, if three trials for a function handling HTTP headers yield {`"Host"`, `"User-Agent"`}, {`"Host"`, `"User-Agent"`}, and {`"Host"`, `"User-Agent"`, `"Referer"`}, the final dictionary would contain {`"Host"`, `"User-Agent"`}. In this example, the token `"Referer"` is spurious, generated as an artifact of the LLM's stochastic nature.

Bug Triggering Token. This module prompts the LLM to generate inputs that are likely to trigger common vulnerabilities. For example, if the target function handles SQL queries, LLMs will produce SQL injection payloads (*e.g.,* '' OR '1'='1'') as tokens. This allows the fuzzer to proactively test for known bug classes.

Formatted String. For functions that perform parsing, this module leverages the LLM's understanding of common data formats to generate strings that are syntactically valid and likely to be successfully parsed (*e.g.,* for a function that parses User-Agent strings, it might generate `"Mozilla/5.0 (X11; Linux x86_64)"`). This helps the fuzzer bypass initial validation checks and explore deeper logic within the parsing function.

6.5.2 Function-level Dictionary-based Mutator

Based on the function coverage of the input, DICTIONARY SELECTOR filters out the dictionary used for mutating the input. To maximize the chances of discovering new coverage and vulnerabilities, the mutator employs a hybrid approach that combines dictionary-based strategies with traditional random mutations. With a high probability, it applies one of the dictionary-based strategies. With a small probability, however, it falls back to a traditional fuzzing strategy, such as performing a random bit or byte flip. The primary mutation strategies are as follows:

Token Insertion. The mutator randomly selects a token from the dictionary and inserts it at a random location in the input. This strategy is effective at testing how the program handles inputs including unexpected data or additional tokens.

Comparison Operand Extraction Prompt

```
Task: Enumerate constants defined globally or as class/struct members.

Key Analysis Rules:
- A constant is assigned once and never changed.
- Do not enumerate constants defined in functions.
- Find definitions at specific line numbers.
- Calculate the resulting value if possible.

Output Format (JSON):
{
"EXPLANATION": "{Your explanation}"
"CONSTANT_A": { "name": "class.CONSTANT_A", "expression": "1+2", "value": 3, "line": "5" },
... (more constants) ...
}
```

Bug Triggering Token Extraction Prompt

````
Task: Given <FUNCTION_NAME>(), what values would trigger vulnerabilities?

Key Analysis Rules:
- Generate values likely to reveal vulnerabilities.
- Assume vulnerability might exist in given function only.

Vulnerability Examples:
- C: Buffer overflow, Use-after-free, Integer under/overflow, ...
- Java: SQL Injection, XPath Injection, Command Injection, SSRF, ...

Example Analysis:
```c
void foo(char *data, size_t size) {
  char buf[10];
  if (size > 0)
    memcpy(buf, data, size);  // Potential buffer overflow at line 4
}
```

Output Format:
- Answer: "A" * 11 <DELIMITER> `data` - `buffer overflow`
````

Formatted String Extraction Prompt

```
Task: Given <FUNCTION_NAME>() (parsing function), generate an accepted string?

Key Analysis Rules:
- Parsing function: takes string input, validates against rules/grammar.
- Generate complex, well-formed strings.
- Ensure string is valid and would be accepted.

Output Format:
{Reason why the string would be accepted by the function}
- Answer: "Hello, World!"
```

Figure 28: Summarized dictionary generation prompts

Token Replacement. A random chunk of the input is replaced with a randomly selected token from the dictionary. This produces structurally valid but semantically different data, which might be helpful for testing whether the program handles unexpected tokens correctly.

Byte Replacement. This is a more granular version of token replacement, where a small, random number of bytes in the input are replaced by a token from the dictionary. This can create subtle changes that may trigger edge-case behaviors.

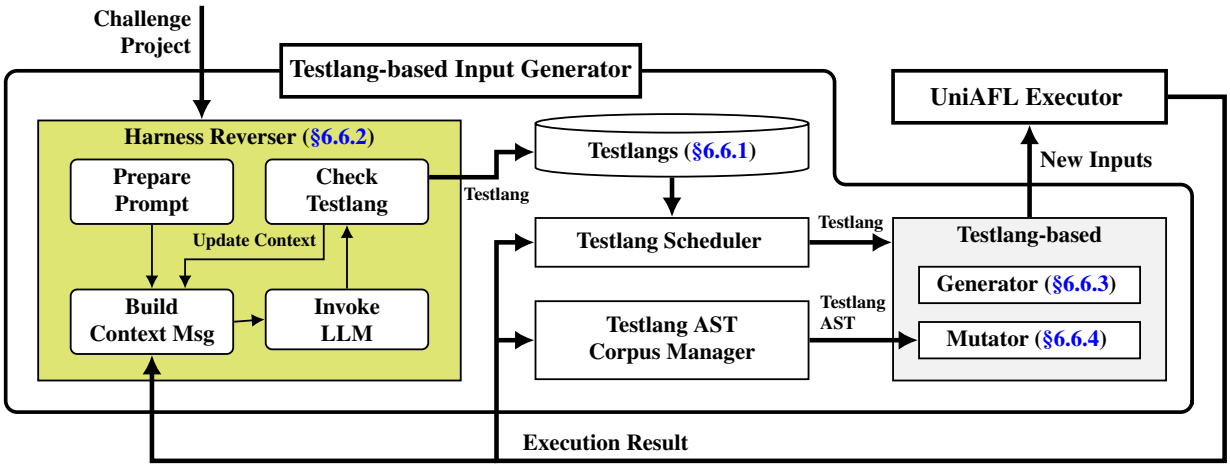


Figure 29: Overview of the Testlang-based input generator.

6.6 Testlang-based Input Generation

One of the key challenges in fuzzing is how to randomly generate good inputs that are highly structured, so that the target program is more likely to accept them. Existing tools and research mainly focus on reverse-engineering the target program to recover its input format. In this subsection, we introduce our Testlang-based input generator and explain how it addresses this challenge. Our approach goes one big step further by not only considering input format inference but also the following:

Optimizing structural specification complexity. Some input formats are extremely complex, making it impractical to fully specify them. By leveraging well-known format specifications and LLM-generated Python scripts, we can balance specification completeness with manageability. Also, partial specifications can kick in to make the LLM focus on the most interesting parts of the input format.

Handling complex codebases. Static analysis often struggles with complex or large codebases. By a simple touch, utilizing LLMs to reverse-engineer code behavior, we can reduce the computational cost of interpreting complex logic and reasoning about input structure identification.

Guiding fuzzing with LLM. LLMs can be used not only for reversing the input format, but also for evaluating fuzzing status using fuzzer feedback. This allows us to systematically prioritize fuzzing directions (e.g., Testlang variants) that align with the analysis strategy of the Harness Reverser. We call this *LLM-opinionated*.

As shown in Figure 29, the Testlang-based input generator contains four major components: the Harness Reverser that extracts input formats, Testlang and Python files which are outputs from the Harness Reverser, and generation and mutation engines that operate on these Testlang-based structured representations. When the challenge project (CP) is given, the Harness Reverser analyzes the target harness code and the broader CP codebase to understand how the harness processes input data, producing a comprehensive Testlang as output. In addition, the Harness Reverser may also write Python scripts that contain tailored generators and encoders to handle complex formats. This will be explained further in §6.6.2. Testlangs, the outputs of the Harness Reverser, capture the hierarchical structure, field types, constraints, and relationships discovered during this analysis process. Testlangs are managed with extra metadata such as CP source coverage target lines, deprioritization, and usage metrics, which are used in the Testlang Scheduler. These will be explained further in §6.6.1. The Testlang and Python files are then fed into the Testlang-based generation and mutation engines to produce new test inputs that are semantically meaningful and structurally valid. The generated inputs are then executed on the target harness by the UniAFL Executor, and the resulting code coverage and program behaviors are monitored and fed back to evaluate and improve the effectiveness of the module. Detailed processes will be explained in §6.6.3 and §6.6.4.

6.6.1 Testlang

Testlang is a domain-specific language designed to formally describe arbitrary data structures and input formats for fuzzing purposes. Grammar-based generation and mutation itself is not a new idea, and there are several existing tools and research that utilize grammar-based approaches for structured fuzzing [5, 26]. However, Testlang introduces several novel features and capabilities that distinguish it from prior works.

LLM-friendly design for on-the-fly generation. Testlang is designed to be easily generated and modified by LLMs. Its syntax is based on JSON schema, which is both human-readable and machine-parsable, with strong support for hierarchical structures, field types, constraints, and relationships. By using popular formats like JSON, we make it straightforward for LLMs to produce valid Testlang. This JSON-based design also provides additional benefits. It enables seamless integration with Testlang-based generation and mutation because these structured representations can be directly interpreted. It also reduces development overhead when modifying features of Testlang as the changes can be easily reflected in the schema. As a result, Testlang achieves both flexibility and maintainability while remaining accessible to both humans and machines.

Semantics of data fields. Testlang can describe not only basic data types but also semantic relationships between fields, thereby ensuring that generated inputs remain structurally valid. Specifically, it supports data types (*e.g.,* integers, strings, arrays), size constraints (fixed, variable), value constraints (ranges, enumerations), inter-field relationships (*e.g.,* dependency, record reuse), and hierarchical structures (nested objects, arrays of objects). For example, in the simple Testlang shown in Figure 30(a), the record named "`Lookup`" contains a field named "`table`" which refers to the preceding field named "`table_size`" as its size.

Generation extensions using external tools. Fields can be associated with custom Python generators that produce semantically valid content, and encoders that transform data during serialization. This enables support for complex formats like compressed data, encrypted content, or domain-specific encodings. Some well-known text-based formats (e.g., XML, HTTP, HTML) may also be handled with a bundled grammar-based generator called Grammarinator [25], inducing the LLM to focus more on finding interesting parts of the input than on the complex input format itself. For example, in Figure 30(b), the record named "`CSVCustom`" contains a field named "`csv_data`" which uses a custom generator that leverages Grammarinator to produce valid CSV content. Another example is shown in Figure 31(a), where the record named "`CSVData`" contains a field named "`csv_data`" which uses a Python-based generator named "`CSVGenerator`" in the corresponding Python code. Details about generating Python-based custom generators will be explained in §6.6.2.

Oracle provision for output quality evaluation. Testlang holds metadata that indicate which lines in the CP source code are expected to be covered when inputs generated from this Testlang are executed. This allows the system to evaluate the quality of a Testlang based on how well it achieves the intended coverage targets. This feedback can be used to refine and improve Testlang over time.

Partial update support. Testlang supports partial specifications that can be incrementally refined. This allows LLMs to start with a basic structure and progressively add details, constraints, and relationships as they gain a deeper understanding of the input format. Also, this enables the LLM to focus on the most interesting parts of the input format by reducing analysis context rather than attempting to fully specify every aspect. For example, in Figure 31(b), the Testlang is marked as a partial specification with the field "`is_partial`" set to true. If this partial Testlang is used with Figure 30(a), the final output Testlang will be a combination of both Testlangs, where the "`INPUT`" record is taken from Figure 30(a), and the others are taken from Figure 31(b).

Focus alignment of LLM analysis and fuzzing. Testlang storages have metadata that guides both the LLM's analysis strategy and the prioritization of fuzzing tasks: (1) deprioritization tags for cases where a Testlang does not align with the current analysis strategy and should therefore be downweighted in future fuzzing cycles; (2) usage metrics such as the frequency of use and the coverage achieved. Leveraging this information, the Testlang Selector can make informed choices about which Testlangs to generate or mutate, ensuring that the fuzzing process remains aligned with the analysis objectives defined by the Harness Reverser.

```
{
  // ... header fields ...
  "records": [
    {
      "name": "INPUT",
      "kind": "struct",
      "fields": [
        {
          "name": "lookups",
          "kind": "array",
          "len": 2,
          "items": {
            "kind": "record",
            "name": "Lookup"
          }
        }
      ]
    },
    {
      "name": "Lookup",
      "kind": "struct",
      "fields": [
        {
          "name": "table_size",
          "kind": "int",
          "byte_size": 4
        },
        {
          "name": "table",
          "kind": "string",
          "len": {
            "kind": "field",
            "name": "table_size"
          }
        }
      ]
    }
  ]
}
```

(a) Simple Testlang Example

```
{
  // ... header fields ...
  "records": [
    // ... previous contents ...
    {
      "name": "CSVDataUnion",
      "kind": "union",
      "fields": [
        {
          "name": "structured_csv",
          "kind": "record",
          "items": {
            "kind": "record",
            "name": "StructuredCSV"
          }
        },
        {
          "name": "csv_custom",
          "kind": "record",
          "items": {
            "kind": "record",
            "name": "CSVCustom"
          }
        }
      ]
    },
    {
      "name": "CSVCustom",
      "kind": "struct",
      "fields": [
        {
          "name": "csv_data",
          "kind": {
            "custom": "CSV"
          }
        }
      ]
    },
    // ... next contents ...
  ]
}
```

(b) Grammar-based Custom Generator

Figure 30: Examples of Testlang output formats.

6.6.2 Harness Reverser

The Harness Reverser is an LLM-powered system that automatically analyzes target programs to extract input format specifications and generate corresponding Testlang descriptions. This component addresses the fundamental challenge of manual format reverse engineering, which is both time-consuming and error-prone. As LLMs have demonstrated strong capabilities in code understanding and generation, the Harness Reverser leverages these strengths to systematically analyze input processing logic within target programs. Also, as the correctness and level of detail of the generated Testlang are crucial for effective testing, the Harness Reverser employs a rigorous analysis process to ensure high-quality output.

By using LLMs, this system systematically analyzes how the harness processes input data, producing a comprehensive Testlang as output. It takes the target harness code and the CP codebase as input. Starting from the harness entry point, the reverser traces data flow through parsing routines, validation logic, and data structure transformations to understand the expected input format. The Harness Reverser also proposes auxiliary Python generators to handle complex field types and encodings, ensuring that the generated Testlang captures both structural and semantic aspects of the input format. This process is iterative, with the LLM actively selecting and refining the code scope to analyze (functions, files, and specific regions) as the investigation progresses. In addition, by leveraging runtime feedback (coverage and crashes), the Harness Reverser refines the generated Testlang, enhancing accuracy and effectiveness in vulnerability discovery.

```
{
  // ... header fields ...
  "records": [
    // ... previous contents ...
    {
      "name": "CSVData",
      "kind": "struct",
      "fields": [
        {
          "name": "csv_data",
          "kind": "bytes",
          "generator": "CSVGenerator"
        }
      ],
    },
    // ... next contents ...
  ]
}
```

```python
class CSVGenerator:
    def generate(self) -> bytes:
        # Generate various CSV structures to
        # trigger different parsing behaviors
        csv_type = random.choice([
            'simple_csv',
            'quoted_fields',
            # ... many other types ...
        ])

        if csv_type == 'simple_csv':
            return self._generate_simple_csv()
        elif csv_type == 'quoted_fields':
            return self._generate_quoted_fields()
        # ... many other generation types ...
        else:
            return self._generate_simple_csv()

    # ... GENERATION METHODS ...

    def validate(self, input_bytes: bytes):
        # Basic validation - ensure valid bytes
        if not isinstance(input_bytes, bytes):
            raise ValueError("Generated data must be bytes")
        # ...

        # Ensure it has some CSV-like structure
        lines = content.split('\n')
        if len(lines) < 1:
            raise ValueError("CSV must have at least one line")
```

(a) Python-based Custom Generator

```
{
  // ... header fields ...
  "is_partial": true,
  "records": [
    {
      "name": "Lookup",
      "kind": "struct",
      "fields": [
        {
          "name": "table_size",
          "kind": "int",
          "byte_size": 4
        },
        {
          "name": "table",
          "kind": "record",
          "len": {
            "kind": "field",
            "name": "table_size"
          },
          "items": {
            "kind": "record",
            "name": "TableEntry"
          }
        }
      ]
    },
    {
      "name": "TableEntry",
      "kind": "struct",
      "fields": [
        {
          "name": "key",
          "kind": "string",
          "terminator": ":"
        },
        {
          "name": "value",
          "kind": "string"
        }
      ]
    }
  ]
}
```

(b) Partial Update Example

Figure 31: Advanced Testlang features.

The Harness Reverser incorporates several key features that improve both the accuracy and effectiveness of Testlang generation:

- **Automated Input Format Extraction:** It automatically identifies and extracts input format specifications from the target harness code and CP code base, achieving advanced structural fuzzing capabilities without requiring manual intervention.
- **LLM-Based Custom Generator Support:** Leveraging LLMs' capability, it recognizes well-known formats and attaches existing custom fuzzers instead of re-specifying them in Testlang. Additionally, for formats that are hard or impractical to model in Testlang alone (e.g., stateful protocols, nested compression, checksums), the LLM synthesizes Python generators that encapsulate the semantics and expose them as Testlang nodes, boosting expressiveness, validity, and coverage.
- **LLM-opinionated Analysis with Runtime Feedback:** It employs an *LLM-opinionated* analysis strategy where the LLM actively directs its own investigation, targeting code sections most likely to yield security-relevant insights. Also, the system integrates runtime feedback from fuzzing (code coverage and crash logs) to evaluate and refine the Testlang iteratively.

Overview. The green box in Figure 29 illustrates the architecture of the Harness Reverser, which operates through a sophisticated multi-stage analysis pipeline that progressively evolves the Testlang through continuous LLM interaction and context refinement. The Harness Reverser consists of five primary stages: Prepare Prompt, Build Context Messages, Invoke LLM, Check Testlang, and Update Context. Each stage plays a critical role in establishing the analysis context, guiding LLM behavior, validating outputs, and refining the understanding of the target program's input format. The following subsections provide a detailed explanation of each stage and its function within the overall analysis workflow.

Prepare Prompt. In the Prepare Prompt stage, the system constructs a base prompt that defines the LLM's role as a reverse-engineering expert and establishes the foundational context for analysis. This prompt includes instructions on how to request code snippets, analyze input processing logic, and express findings in Testlang format. The prompt also incorporates representative examples of analysis patterns and expected output formats to guide the LLM's reasoning process.

This step begins by loading and preprocessing the harness code from the specified file path, creating a structured `Code` object that includes full location metadata, such as file path and line number ranges. For `Code` objects, the system applies code filtering based on coverage information to identify and remove unused code sections. The filtering stage extracts compiled line information from the target binary to determine which lines are actually compiled and executed, and systematically eliminates preprocessor conditional blocks that can never be reached given the build configuration. This preprocessing ensures the LLM focuses its analysis only on relevant, executable code paths.

In delta mode, the system processes *diff* information to extract additional code blocks that represent changes in the broader codebase, providing context about modifications that may affect input processing logics. The diff processing identifies modified functions, added data structures, and changed parsing logic that could impact input format requirements. This integration allows the reverser to understand how recent changes to the codebase might have introduced new input format requirements or modified existing parsing logics. Consequently, this stage constructs a comprehensive message consisting of four primary components:

- **System Message:** Defines the LLM's role and reverse engineering capabilities.
- **Grammar Specification Message:** Contains the complete Testlang JSON schema dynamically populated with available custom generator definitions from the `customgen` module.
- **Examples Message:** Provides representative analysis patterns and expected output formats.
- **Target Message:** Contains the preprocessed and filtered harness code with any relevant diff information.

To optimize token usage and response latency across multiple analysis iterations, the system applies strategic message caching to the examples and target messages. The caching strategy leverages Anthropic's extended cache-TTL beta features to reduce API costs and improve response times for repeated analyses of similar code patterns. This optimization is particularly important for iterative analysis workflows where similar context is repeatedly processed. Additionally, when the Harness Reverser is invoked multiple times during a single analysis session, it loads previously generated Testlang from prior iterations to provide continuity and context.

Build Context Messages. This stage builds the context messages that enable the LLM to make the next analysis and generation decisions. The context contains (1) the current Testlang with its associated Python generator code; (2) LLM-requested code blocks with file/line metadata; (3) runtime feedback (code coverage and crash logs); and (4) warnings and error messages to guide the LLM. The output is a set of structured messages that provide situational awareness for the LLM's next invocation. This adaptive context building process operates as the critical bridge between the static code analysis phase and the dynamic fuzzing execution phase, ensuring LLMs receive complete situational awareness for informed decision-making.

The context construction process begins with Testlang state management. For initial analysis sessions, the system presents an empty Testlang context with explicit guidance to create a default *INPUT* record structure with appropriate mode and endianness settings. For ongoing sessions, the method formats the current validated Testlang with its versioned identifier, enabling LLMs to understand the evolution of their analysis. Associated Python generator codes are included as structured code blocks with names and implementations, allowing LLMs to reference and modify existing generators during iterative refinement.

Next, the system processes LLM code search requests to retrieve specific code blocks for analysis. It interprets structured queries from the LLM that specify function names, file paths, and line numbers, and extracts the corresponding code snippets from the preprocessed harness and project code. Each retrieved code block is annotated with full location metadata to provide context for the LLM's analysis.

Furthermore, Testlang is refined using runtime feedback such as code coverage and crash logs. As the LLM selects target lines to cover in the Testlang metadata, the system correlates coverage data with these targets to determine which lines were successfully exercised by generated inputs. This provides precise feedback on Testlang's effectiveness in meeting coverage goals. In addition, crash logs are parsed to extract details about discovered vulnerabilities, allowing the system to guide the LLM away from retargeting known issues and toward unexplored code paths.

Lastly, the system synthesizes various warnings and error messages to guide the LLM's analysis strategy. These messages include validation warnings from the Testlang checking stage, execution errors of Python generators, and strategic advisories based on runtime feedback. These messages are structured to provide clear, actionable insights that help the LLM refine its analysis and generation approach. Notably, the LLM may ignore certain warnings if it deems them irrelevant to its current analysis strategy, allowing for flexible decision-making. If the LLM chooses to disregard specific warnings, the system respects this decision and does not reintroduce those warnings in subsequent iterations, preventing redundant feedback loops.

Invoke LLM. In this stage, the system invokes the LLM with the prepared base prompt and dynamically constructed context messages. The invocation process implements a robust architecture that handles transient errors and maintains operational continuity during extended analysis sessions. Since the system operates in automated fuzzing environments without human supervision, robustness is essential due to the unpredictability of LLM responses, which may include incomplete outputs, inconsistent formatting, or unexpected analytical directions that could disrupt the iterative analysis pipeline.

The invocation architecture uses a context-based interaction model that reconstructs complete context for each LLM invocation rather than maintaining conversation history. Each call provides base prompts defining the analysis objectives along with comprehensive context messages containing current analysis state. This approach prevents information bloat by filtering out irrelevant historical exchanges while ensuring the LLM receives the information needed for informed decision-making. The conversation-history-free design enables deterministic analysis outcomes, reduces token consumption through strategic information curation, and maintains control over analytical direction by presenting only refined, contextually relevant information.

The invocation process employs error recovery and resource management mechanisms for LLM interaction robustness during extended analysis sessions:

- **Context window overflow:** Content reduction that replaces detailed code implementations with standardized placeholders while preserving essential metadata such as function names, file paths, and locations.
- **Token limit violations:** Reallocation of output tokens based on available context capacity, with automatic content reduction when limits are exceeded.
- **Model switching failures:** Automatic switching between different LLM models when encountering failures or timeouts to maintain continuous operation.
- **LLM infrastructure issues:** Handling of LLM errors, API compatibility changes, and caching optimizations to ensure continuous operation despite external service constraints.

Check Testlang. In this stage, the system validates LLM-generated Testlang to ensure syntactic correctness, semantic validity, and effectiveness in producing test inputs for vulnerability discovery. This validation also checks the quality of both the Testlang structure and its associated Python generator code, guaranteeing compatibility with Testlang-based input generation and mutation.

First, the generated Testlang undergoes basic JSON schema validation to ensure syntactic correctness and semantic validity. The system checks for required fields, data types, and hierarchical relationships defined in the Testlang schema. Any deviations from the schema result in immediate rejection of the Testlang, with detailed error messages provided to guide LLM corrections. Furthermore, during the validation process, the system generates various warnings to guide LLMs in refining the Testlang. These warnings may include suggestions for improving clarity, enhancing coverage, or addressing potential ambiguities in the generated Testlang. For example, if a field lacks sufficient constraints, the system may issue a warning recommending additional analyses to define appropriate value ranges or dependencies. In addition, depending on the field types and constraints specified in the Testlang, the system may suggest additional code analysis to identify relevant parsing logic or validation routines that can inform more precise field definitions.

Second, Python generator code associated with Testlang fields also undergoes rigorous validation to ensure correctness and effectiveness. The system checks for syntax errors, runtime exceptions, and logical consistency within the generator implementations. Each generator is executed in a controlled environment with randomized input data to verify that it produces valid outputs without errors. This execution-based validation helps identify issues that static analysis alone may miss, such as unhandled edge cases or incorrect assumptions about input data. These validation results are used to generate warnings that inform the LLM about potential issues in the generator code, guiding improvements and refinements of Python generators.

Since Python generators play a crucial role in producing actual bug-triggering inputs, the system has additional guidance and validation mechanisms specifically for them. The LLM assigns security severity scores to the vulnerability classes that each generator targets, so that the system can prioritize generators that focus on high-severity vulnerabilities. It also assigns trigger probabilities indicating how likely the generator is to produce inputs that can trigger the targeted vulnerabilities. These scores are used to prioritize generators during fuzzing and to guide LLM analysis toward the most promising directions.

Update Context. Lastly, the system updates its internal context state to reflect the outcomes of the current analysis iteration and prepare for subsequent cycles. This stage involves persisting validated Testlang, managing warning states, and implementing error recovery mechanisms to ensure continuous progress through the iterative analysis pipeline. For critical errors that prevent further analysis (e.g., LLM failures, validation errors, code search failures), the system preserves the previous LLM output and error context while clearing warnings. This prevents cascading failures from terminating analysis prematurely and provides LLMs with sufficient context to understand and resolve errors prior to the next iteration.

6.6.3 Testlang-based Generation

Based on the Testlang generated by the Harness Reverser, this component generates new inputs focusing on comprehensive exploration of the specified input format.

Semantics-aware. Rather than generating random byte sequences, the system produces inputs that are semantically meaningful within the context of the target application. This includes respecting harnesses that use FuzzedDataProvider (FDP) by incorporating LIBFDP (see §6.9.3) so that generated inputs are consumed by the target harness properly.

Balancing coverage and error findings. The generation process employs two complementary strategies. Coverage-targeted generation produces valid inputs that adhere to Testlang constraints and explore the input space to maximize code coverage. Crash-targeted generation intentionally applies out-of-range values to trigger exceptions and expose vulnerabilities. Together, these strategies balance breadth and depth in testing.

Integrating Custom Generators. The system seamlessly integrates custom grammar-based and Python-based generators that can generate inputs with complex formats. These generators leverage existing parsing libraries to ensure format validity while introducing controlled variations to explore edge cases. Furthermore, they can incorporate domain knowledge (*e.g.,* known file formats) to create realistic test cases.

Contributing to the Main Seed Pool. As described in §6.6.4, we maintain separate seed pools for Testlang-based inputs to enhance mutation effectiveness. However, rather than using those inputs for Testlang-based mutations only, the generated inputs are also added to the UniAFL corpus if they are interesting (e.g., discover new coverage) to further enhance the overall fuzzing process.

Focus alignment of LLM analysis and fuzzing. Before generating new inputs based on Testlang, the Testlang Selector prioritizes Testlangs using metadata and achieved coverage. The Harness Reverser may deprioritize Testlangs that do not align with the current analysis strategy or fail to deliver sufficient coverage, while recently created Testlangs are favored as they better reflect the current analysis focus and help avoid rediscovering already found crashes. Usage metrics such as how frequently a Testlang has been used are also considered to promote diversity in exploration. By leveraging this information, the Testlang Selector improves fuzzing performance and ensures that the process stays aligned with the overall analysis goals.

6.6.4 Testlang-based Mutation

Based on the Testlang from the Harness Reverser, this component mutates inputs while preserving their structural validity and effectively exploring the input space.

Seed Pool Separation. To enhance mutation effectiveness, the system maintains separate seed pools for Testlang-based inputs and byte-level inputs from other input generators. It is very challenging to restore the structure of an input by only using the input definition described in Testlang. Thus, we separated seed pools to ensure that Testlang-based mutations are applied to inputs that are already structured and semantically meaningful according to the Testlang. Mutated inputs are also added to the main UniAFL seed pool if they are interesting. We still allow Testlang-based mutations to be applied to inputs from other input generators, which is called *AST-Free Mutations*, to seek more opportunities in exploring the codebase.

Structure-aware Mutations. Unlike byte-level mutations that can easily break input format constraints, which is not a mainly desired behavior for this module, Testlang-based mutation operates at the semantic level. Mutations respect field types, size constraints, and inter-field relationships, ensuring that mutated inputs remain structurally valid while exploring different value combinations. The system implements multiple mutation strategies tailored to different field types:

- **Boundary Value Mutation:** Replace numeric fields with values near boundaries (*e.g.,* INT_MAX).
- **Type-Aware Mutation:** Apply domain-specific mutations (e.g., path traversal strings for filename fields, SQL injection patterns for text fields).
- **Constraint Violation Mutation:** Deliberately violate specified constraints to test error handling paths.
- **Cross-Field Mutation:** Modify relationships between fields to test validation logic.
- **AST-Free Mutation:** Mutate inputs from non-Testlang based input generators by inserting or replacing part of the inputs with Testlang-based structures.

The Testlang-based approach demonstrates significant advantages over traditional fuzzing methods, particularly for programs requiring structured inputs and domain-specific formats. By operating at the semantic level rather than the byte level, the system achieves higher code coverage, discovers vulnerabilities more efficiently, and reduces the time spent on invalid inputs that are immediately rejected by input validation routines. LLM-powered analysis is especially crucial for understanding complex domain-specific schemas such as HTML/XML, SQL queries, configuration file formats, and protocol specifications, where traditional fuzzing approaches struggle to generate valid inputs that can penetrate deep into application logic.

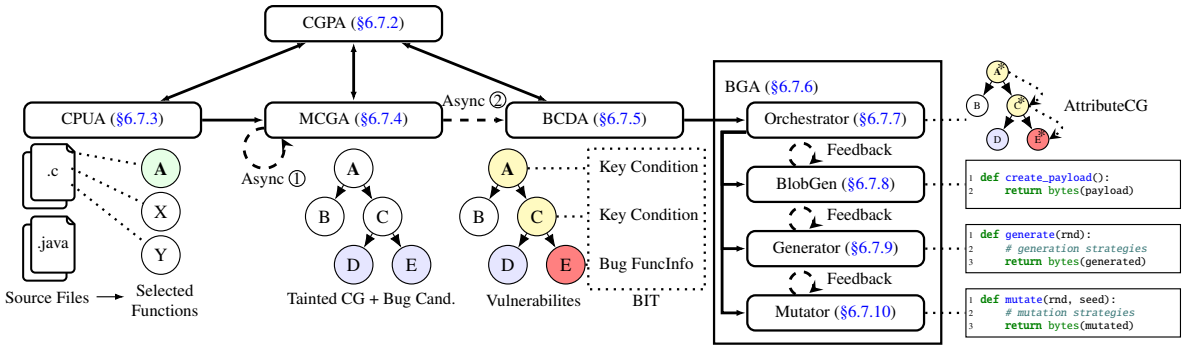


Figure 32: The overall architecture of MLLA.

6.7 Multilang-LLM-Agent (MLLA)

This section describes the Multi-Language LLM Agent (MLLA), which is the most LLM-intensive input generation module within the ATLANTIS-Multilang system (§6).

Traditional fuzzers rely on syntax-unaware mutations that fail to generate the semantically-valid inputs required for modern vulnerabilities which are often hidden behind complex data structures, format validation, and are incapable of handling stateful execution sequences. MLLA addresses this semantic gap by leveraging LLMs' code comprehension capabilities, employing a coordinated multi-agent architecture that transforms vulnerability discovery from blind mutation to intelligent, state-aware, and targeted exploitation. While LLMs offer semantic understanding, applying them to vulnerability discovery introduces fundamental challenges that MLLA addresses through specialized agent design:

C1: Scale and Context Limitations. Analyzing large, multi-language codebases strains LLM context windows and computational resources. Complex call chains and deep function hierarchies exceed token limits, while function resolution ambiguity creates uncertainty when multiple candidates match queries.

C2: Precision and Validation Challenges. Non-determinism and hallucinations of LLMs make reliable vulnerability detection difficult. Distinguishing genuine vulnerabilities from hallucinations requires validation mechanisms, while target-reaching accuracy depends on understanding complex triggering conditions.

C3: Integration and Coordination Complexity. Coordinating many tools such as code retrievers, static analyzers, script execution environment and input executor introduces systemic complexity. Furthermore, it requires robust design to orchestrate agents with different strategies while maintaining fault tolerance and resource management.

6.7.1 Overview

MLLA operates in two primary modes: a lightweight standalone mode for rapid seed generation, and a full pipeline mode for systematic vulnerability exploitation. The standalone mode leverages LLMs to generate diverse fuzzing seeds directly from harness analysis (detailed in §6.8). The full pipeline mode orchestrates specialized agents through the workflow illustrated in Figure 32, where each agent addresses traditional fuzzing limitations while overcoming LLM-specific challenges (**C1**-**C3**).

Call Graph Parser Agent (CGPA). This agent resolves ambiguous or incomplete function information into precise, structured representations (`FuncInfo`), leveraging Joern, LSP, code indexer, and AST-based search. This mitigates inaccuracies in LLM-based function resolution while addressing **C1** by ensuring consistent function resolution across large, multi-language codebases. Therefore, preventing propagation of hallucinated or incorrect function definitions into downstream agents (see §6.7.2).

CP Understanding Agent (CPUA). This agent analyzes harness files to identify functions that need to be analyzed with taint flow information. This enables semantic code scoping beyond traditional fuzzing capabilities while managing **C1** by transforming overwhelming codebase complexity into context-window-compatible analysis targets (see §6.7.3).

Make Call Graph Agent (MCGA). This agent constructs interprocedural call graphs while detecting vulnerable sinks within each function. This overcomes cross-language call relationship mapping limitations while addressing **C1** through recursive decomposition, caching, and tool integration that validates LLM responses and prevents hallucinated function calls (see §6.7.4).

Bug Candidate Detection Agent (BCDA). This agent validates vulnerabilities by analyzing execution paths to distinguish genuine issues from false positives, extracting concrete triggering conditions. This provides semantic vulnerability understanding beyond traditional capabilities while solving **C2** through multi-stage analysis that combines LLM reasoning with execution verification (see §6.7.5).

Blob Generation Agent (BGA). This agent generates targeted payloads through script-based creation of format-compliant exploits. This transcends syntax-unaware mutations with semantically-valid inputs while addressing **C2** and **C3**. Basically, BGA consists of four sub-agents where the *Orchestrator Agent* coordinates three sub-agents: *BlobGen* creates single payloads through iterative refinement with coverage feedback, *Generator* produces multiple payloads to increase target-reaching probability, and *Mutator* applies focused mutations for individual function transitions when context is complex (see §6.7.6–§6.7.10).

These agents operate with execution validation, where some operate asynchronously, mitigating non-determinism of LLM by feedback loops, tool verification, and adaptive refinement.

Tool Integration. The architecture incorporates sophisticated tool integration enabling capabilities beyond pure LLM reasoning. MCGA and BCDA utilizes the CGPA in a self-loop configuration for precise function analysis, while all agents benefit from shared tooling infrastructure including static analysis tools, language servers, and state management systems (see §6.7.2 and §6.9).

Diff-aware Analysis. When provided with diff files, MLLA leverages LSP-based function-level diff analysis to intelligently prioritize modified functions. The system maps diff hunks to specific function boundaries using Language Server Protocol (LSP), enabling targeted vulnerability analysis of code changes that are most likely to contain newly introduced vulnerabilities. For instance, MCGA integrates diff context directly into function analysis, treating modified functions as high-priority targets with comprehensive taint tracking.

6.7.2 Call Graph Parser Agent (CGPA)

CGPA addresses the challenge of resolving ambiguous or incomplete function information in large, multi-language codebases. Because CPUA and MCGA often operate on partial or uncertain function metadata (*e.g.,* a function name without full signature, or a callsite without definition), CGPA provides a reliable mechanism for mapping such partial information to precise, structured function representations. This capability is essential to prevent hallucinated callees and ensure downstream agents operate on accurate code contexts.

Functionality and Workflow. Given partial input such as a function name, callsite, or file path, CGPA queries multiple backends described in §6.9 to collect candidate definitions: Joern for graph-based static queries, LSP servers for symbol resolution, a lightweight code indexer for efficient search, and AST-grep for syntax-level matching. CGPA aggregates these results, removes duplicates, and applies heuristic or LLM-based selection to choose the most relevant definition. The output is a structured `FuncInfo` object that contains the file path, function signature, code body, and other metadata required for accurate call graph construction. This standardized output format enables consistency across agents, allowing MCGA and BCDA to consume function information seamlessly without having to implement redundant resolution logic. When multiple definitions remain ambiguous, CGPA invokes LLM-based reasoning to rank candidates, leveraging context such as caller function body or surrounding imports. Results are cached in Redis to minimize repeated computation and accelerate future queries, especially in iterative or recursive analyses.

Integration and Benefits. CGPA serves as a shared utility within MLLA, supporting both CPUA and MCGA by providing precise function resolution. By combining multiple backends with LLM validation, CGPA mitigates limitations of purely static or purely LLM-based reasoning, ensuring accurate resolution in the face of overloads, or reflection. This prevents error propagation in call graph expansion and strengthens downstream vulnerability detection. Ultimately, CGPA transforms incomplete and noisy code references into consistent, analyzable function objects, allowing the overall system to maintain robustness across diverse languages and complex codebases.

6.7.3 Challenge Project Understanding Agent (CPUA)

CPUA addresses the fundamental scoping problem in automated vulnerability analysis and **C1** by determining which functions in a large codebase deserve detailed investigation. Instead of attempting to analyze all functions indiscriminately, CPUA focuses the analysis pipeline by interpreting harness files and extracting high-value entry points that are most likely to process fuzzed input or expose vulnerabilities. This targeted approach enables MLLA to allocate resources efficiently and reduce noise in downstream analyses.

Functionality and Workflow. Given the contents of a harness file, CPUA applies LLM-based reasoning to identify interesting candidate functions. These include functions that (1) directly receive fuzzer-provided input, or (2) propagate data to downstream libraries. For each candidate, CPUA generates rich metadata including priority level, call sites, and tainted arguments. The output is a prioritized list of target functions annotated with metadata, which downstream agents such as MCGA consume to construct detailed call graphs.

Integration and Benefits. A major strength of CPUA lies in its language-agnostic design. By relying on the LLM's semantic understanding rather than on language-specific ASTs or hardcoded parsing rules, CPUA can operate seamlessly across C, C++, Java, and other supported languages without modification. This abstraction makes CPUA effective even in projects that rely on reflection or dynamic dispatch, where traditional static analyzers struggle. However, its effectiveness depends on both the quality of the harness and the reliability of the LLM's reasoning. When harnesses expose limited or indirect entry points, CPUA may yield incomplete coverage. For example, in projects like `nginx`, communication often occurs through sockets rather than direct function calls. In such cases, the fuzzing harness can interact with the program externally without ever invoking its internal functions, leaving CPUA with little semantic signal to analyze.

6.7.4 Make Call Graph Agent (MCGA)

MCGA addresses the fundamental challenge of mapping function call relationships in large, multi-language codebases. Based on interesting candidate functions identified by CPUA, MCGA constructs precise call graphs that capture all direct and indirect callees while simultaneously detecting vulnerable sinks. To enhance accuracy beyond static analysis alone, MCGA optionally integrates FUNCTION TRACER for dynamic callsite discovery. While static analysis provides comprehensive coverage of potential call relationships, FUNCTION TRACER supplies runtime execution traces that validate actual function calls (see §6.9), enabling MCGA to distinguish between theoretically possible and practically reachable paths. This hybrid approach significantly improves call graph precision by incorporating real execution behavior into the analysis. This dual responsibility forms the backbone of the static analysis layer, enabling downstream components to reason about interprocedural control flow, identify vulnerable sinks, and guide payload generation strategies toward security-relevant code paths.

Functionality and Workflow. Given a function and its metadata (*e.g.,* name, location, tainted arguments), MCGA performs a recursive analysis to build a call graph where each node represents a function and edges denote callsite relationships. Algorithm 4 describes the recursive process of the MCGA, which builds a call graph rooted at a given target function.

Algorithm 4: MCGA: Make Call Graph Agent

```
Input: Target function f with metadata (name, file, code, tainted args)
Output: Call graph rooted at f with sink annotations
1  node ← ResolveFuncInfo(f) via CGPA
2  if node is already visited then
3      return cached result
4  sinks ← DetectVulnSink(node)
5  if sinks ≠ ∅ then
6      DispatchAsync(BCDA, node)
7  callees ← ExtractCallees(node.code)
8  foreach callee ∈ callees do
9      calleeInfo ← ResolveFuncInfo(callee)
10     DispatchAsync(MCGA, calleeInfo)
11     Add calleeInfo as child of node
12 Cache(node)
13 return node
```

For every visited function, MCGA performs two tasks: (1) determine whether the function contains vulnerable sinks, and (2) identify its callees. MCGA begins by invoking CGPA to resolve the exact definition of each function (line 1), using a code retriever described in §6.9.2. Once resolved, MCGA checks each function for vulnerability patterns such as unsafe deserialization, unchecked system calls, or improper input validation. Sink detection is performed via LLM prompts augmented with static context (line 4). Simultaneously, the function body is parsed to extract all call sites (line 7). For each callee, MCGA repeats the resolution process and expands the graph recursively (line 10). It incorporates cycle detection and depth limits to ensure termination. For functions with recent modifications, MCGA integrates diff context directly into the function body analysis, treating all function arguments as tainted when diff information is present to ensure comprehensive vulnerability detection in recently changed code. If a function is marked as potentially vulnerable, it is asynchronously forwarded to BCDA for further analysis.

Asynchronous and Cached Execution. MCGA invokes both itself (① in Figure 32) and BCDA (② in Figure 32) asynchronously to improve analysis throughput and reduce latency in vulnerability detection. When MCGA encounters a function during call graph expansion, it may reinvoke itself recursively as a non-blocking task, enabling parallel exploration of multiple call paths. Additionally, if a function is found to contain a potential vulnerability, it is immediately dispatched to BCDA without waiting for the full call graph to complete. This asynchronous design allows MLLA to prioritize and analyze high risk regions earlier, overlap LLM I/O and compute-intensive tasks, and avoid bottlenecks that would arise from sequential traversal, especially in large, deeply nested codebases.

6.7.5 Bug Candidate Detection Agent (BCDA)

BCDA solves the critical challenge of distinguishing true vulnerabilities from false positives in the call graphs from MCGA. When MCGA identifies functions with potential vulnerable sinks, the key question is whether these sinks represent exploitable vulnerabilities and, if so, under what conditions they can be triggered.

BCDA addresses this challenge through an LLM-powered vulnerability analysis system that processes extracted execution paths into structured Bug-Inducing Things (BITs). By taking as input a function previously identified in MCGA as potentially containing a vulnerable sink, BCDA performs a multi-stage analysis to determine whether a vulnerability exists within a given code path. Beyond simple detection, BCDA identifies the specific conditions (*e.g.,* if-else branches or exception handling) necessary to trigger the discovered vulnerabilities, providing concrete guidance for directed fuzzing and payload generation stages.

Functionality and Workflow. BCDA operates in four main phases. First, it receives candidate sinks and corresponding call graphs from MCGA and forms source-to-sink execution paths. It then performs path expansion and pruning to enrich the call path with relevant functions while discarding irrelevant ones. Next, BCDA applies vulnerability classification, using sanitizer-specific prompts and domain knowledge to decide whether the path indeed contains a vulnerability. If a vulnerability is detected, BCDA performs key condition extraction to identify the critical branching conditions that must be satisfied to reach the sink. Finally, BCDA compiles the results into structured Bug Inducing Things (BITs), which encapsulate vulnerability type, location, trigger conditions, and priority. These BITs serve as actionable vulnerability candidates for downstream fuzzing agents.

Path Expansion and Pruning. Initially, after MCGA detects a potential sink, BCDA receives the complete call graph. Before BCDA's main procedure begins, it filters out paths that have already been analyzed or are currently under analysis, retaining only the unexplored ones. Then, BCDA starts its analysis by forming a call path from the source function (typically the harness) to the sink function.

BCDA expands execution paths to address the issue of incomplete information in call stack–like paths, which often lack details about out-of-stack functions. To enrich the execution path with the required functions, we provide all available functions to the LLM and retain only the necessary ones. Using Tree-sitter–based code parsing, BCDA extracts all function calls within each node of the execution path. This expansion process ensures that the analysis has access to the full context of data transformations and validation logic that occur along the vulnerability path.

To avoid analyzing irrelevant code, BCDA uses LLM-powered pruning to selectively retain only the functions likely to be relevant for vulnerability detection. To improve the effectiveness of this pruning, each additionally attached function is tagged with its own ID, and the LLM is prompted to output the tags corresponding to the necessary functions. As a result, the expanded path includes more context needed to determine the vulnerability, ensuring that subsequent analysis remains focused on vulnerability-relevant code.

Vulnerability Classification. The subsequent step for BCDA involves classifying the expanded path to determine the presence of the vulnerability. The classification is guided by the potential sanitizer type (i.e., bug type) identified through the sink classification in MCGA. BCDA then formulates a sanitizer-specific prompt that provides detailed explanations of relevant vulnerability information, common patterns, and effective detection strategies (leveraging the domain knowledge described in §6.7.11). This prompt provides the LLM with concrete guidance to analyze the expanded code path and accurately determine whether it contains the specified vulnerability or not.

Key Condition and Trigger Path Extraction. The key condition extraction step is performed on the expanded path if it is determined to contain a vulnerability. This step identifies important conditional statements along the expanded path that must be satisfied to reach the vulnerable location. For example, an if statement may need to evaluate to true to reach the vulnerability, while a try-catch block might need to fail in order to enter the error-handling code.

BCDA analyzes each function-level transition individually, rather than extracting conditions from the entire path. This allows the LLM to focus on a single transition between functions, making it easier to accurately identify critical decision points. For each transition, BCDA constructs detailed prompts that include call flow information, relevant additionally expanded functions, and context of the source and target locations. This enables the LLM to understand the specific conditions required to trigger a vulnerability.

BIT Generation. The final output of the BCDA analysis is a Bug Inducing Thing (BIT) data structure, which includes all relevant information about identified vulnerabilities in a format optimized for subsequent LLM-based analysis and fuzzing stages. Each BIT contains the vulnerability type, location of vulnerability, priority level, all identified key conditions, and analysis messages from each step. Priority levels are assigned based on factors such as the presence of interesting code changes (*e.g.,* recent modifications that may have introduced vulnerabilities). Each BIT is then forwarded to the next agent and to the fuzzers.

6.7.6 Blob Generation Agent (BGA) Framework

BGA framework comprises the payload generation stage of MLLA, transforming LLM-based vulnerability exploitation into a coordinated multi-agent system. The fundamental insight driving our design is that effective vulnerability exploitation requires not just generating payloads, but creating an intelligent feedback loop between LLM reasoning, code execution, and domain knowledge (see §6.7.11).

Key Innovation: Script-Based Payload Generation. Traditional fuzzing approaches lack semantic understanding of code paths and data structures required to trigger vulnerabilities, while single-shot LLM payload generation often fails due to non-determinism and complexity. The BGA framework addresses this by having LLMs generate Python scripts that serve as "executable exploit recipes", which programmatically construct precise payloads while documenting the reasoning process. This script-based approach enables complex payload construction, format-aware generation, and systematic exploration of vulnerability-triggering conditions.

Multi-Agent Architecture. The framework employs four specialized agents that work synergistically to overcome distinct challenges in automated exploitation: The *Orchestrator Agent* serves as the coordination hub, receiving Bug Inducing Things (BITs) and call graphs from BCDA, applying intelligent filtering to eliminate redundant work, and dispatching contexts to specialized agents through concurrent `asyncio` execution. The *BlobGen Agent* creates single targeted payloads through iterative refinement, generating `create_payload() -> bytes` functions that evolve based on coverage feedback to systematically overcome obstacles preventing successful exploitation. The *Generator Agent* employs probabilistic target-reaching, producing `generate(rnd: random.Random) -> bytes` functions that create multiple payloads to dramatically increase the likelihood of reaching target vulnerabilities. The *Mutator Agent* achieves surgical precision in complex call paths, generating `mutate(rnd: random.Random, seed: bytes) -> bytes` functions that focus on individual function transitions when full context would exceed LLM limitations.

Synergistic Integration. This multi-agent design creates complementary exploitation strategies: BlobGen provides depth through iterative single-payload refinement, Generator offers breadth through probabilistic variation, and Mutator enables precision through focused transition analysis. The Orchestrator unifies these approaches, ensuring that each vulnerability receives the most appropriate exploitation strategy based on its characteristics.

The framework seamlessly integrates with the UNIAFL infrastructure (described in §6.3). BlobGen's binary blobs feed directly to the INPUT EXECUTOR, while Generator and Mutator functions execute via the SCRIPT EXECUTOR to produce continuous streams of payloads. Successful payloads that explore new paths become seeds for broader fuzzing campaigns, enabling the BGA framework to contribute alongside other input generators in the ATLANTIS-Multilang system.

All agents utilize Claude Sonnet 4 (`claude-sonnet-4-20250514`) with temperature `0.4` for consistent code generation, operating through Docker-based execution environments with comprehensive sanitizer integration including Jazzer for JVM targets and AddressSanitizer for native code. We selected Claude Sonnet 4 after evaluating performance-cost trade-offs: during our internal evaluation, we found that while Claude Opus 4 demonstrated slightly better performance, its 5× higher cost rendered Claude Sonnet 4 the optimal choice for our multi-agent vulnerability analysis pipeline. We also compared Claude models with other LLMs; for detailed evaluation results across different vulnerability types and languages, see Table 12.

6.7.7 BGA: Orchestrator Agent

The Orchestrator Agent serves as the coordination hub for the BGA framework, receiving *Bug Inducing Things (BITs)* and call graphs from BCDA and intelligently dispatching them to the three specialized payload generation agents. Its primary role is to maximize exploitation effectiveness through strategic work distribution and resource management.

Intelligent Filtering and Prioritization. The Orchestrator applies structured filtering logic before dispatching work to downstream agents. It eliminates redundant efforts by filtering out: (i) transitions already covered by previous fuzzing inputs, ensuring agents focus on unexplored vulnerability paths; (ii) duplicated transitions across call graphs, preventing multiple agents from targeting identical code paths; and (iii) transitions lacking conditional branches, as these offer limited mutation opportunities. Additionally, the Orchestrator duplicates high-priority BITs to increase the probability of successful exploitation for critical vulnerabilities. Historical crash data is reintegrated to avoid redundant exploration of previously triggered vulnerabilities.

Concurrent Execution Architecture. Implemented using `asyncio`, the Orchestrator enables concurrent execution of all three payload generation agents while maintaining system stability. It employs semaphore-based concurrency control to prevent resource exhaustion, logs detailed performance metrics for analysis, and implements fault isolation to ensure that failures in one agent do not cascade to others.

Context Transformation. The Orchestrator transforms raw BITs and call graphs into structured execution contexts tailored for each agent's specialization. For BlobGen, it provides complete vulnerability context for iterative refinement; for Generator, it supplies source and destination function information for probabilistic exploration; and for Mutator, it identifies specific function transitions for surgical precision. This context-aware dispatch ensures that each agent receives the information format best suited to its exploitation strategy.

6.7.8 BGA: BlobGen Agent

Key Idea. The BlobGen Agent makes a fundamental shift in automated payload generation by adopting a script-based approach. Its core innovation is *generating Python scripts to create payloads*, rather than generating the raw payloads directly. The agent prompts an LLM to create a Python function that programmatically constructs the exploit payload. This method offers several advantages: it allows for precise control over complex data structures and formats, enables incorporation of dynamic values, provides a reproducible recipe for exploit generation, and facilitates intelligent feedback loops through readable code that documents the exploitation logic. The generated scripts serve as "executable exploit recipes" that capture both the payload construction logic and the reasoning behind it, enabling iterative refinement based on execution feedback.

Workflow. The BlobGen Agent follows a systematic 5-step workflow for payload generation: (1) *Sanitizer Selection* (optional): chooses appropriate sanitizers if not pre-configured based on target language and vulnerability context by leveraging the vulnerability categorization described in §6.7.11, (2) *Payload Script Generation*: creates a Python function `create_payload()` using LLM-powered analysis of vulnerability context, incorporating domain knowledge about exploit guides and data structures (§6.7.11) to define the logic for building the exploit payload, (3) *Coverage Collection*: executes the generated script in a Docker sandbox environment to produce binary payloads and collects execution data, (4) *Failure Analysis*: analyzes coverage data and crash information to assess effectiveness and understand why payloads failed to trigger vulnerabilities, and (5) *Iterative Refinement*: uses execution coverage feedback to generate improved scripts, progressively refining the exploitation strategy through multiple iterations.

System Prompt Design. The BlobGen system prompt follows key design principles that maximize LLM effectiveness for vulnerability exploitation. Our approach employs hierarchical structure, context-before-complexity, annotation clarity, and XML organization [2, 3, 33]. Figure 33 demonstrates these principles in practice. The `<role>` section establishes the LLM as a security researcher, priming it for technical exploitation challenges. The `<final_objective>` enforces critical constraints, mandating a `create_payload` function that returns only single bytes objects using built-in Python libraries, ensuring seamless integration with fuzzing infrastructure. The `<context>` section provides upfront target binding with project name and sanitizer type before introducing code complexity, preventing focus dilution across irrelevant possibilities. The `<code_annotations>` section explains our annotation system: `@BUG_HERE` and `@KEY_CONDITION` markers derived from BCDA's BITs, with `@` prefix chosen to avoid confusion with developer comments.

BlobGen System Prompt

```
<role>
You are an expert security researcher specializing in vulnerability analysis and exploit development
for an oss-fuzz project. Your mission is to analyze code for security vulnerabilities and
demonstrate them through carefully crafted payloads that trigger sanitizers.
</role>
<expertise>
You possess specialized knowledge in:
- Vulnerability analysis in large codebases
- Endianness handling
- Sanitizer-based vulnerability detection
...
</expertise>
<final_objective>
Your ultimate goal is to implement a Python 'create_payload() -> bytes' function that:
- Returns ONLY a single bytes object (no tuples/dicts)
- Handles loop iterations and state when needed
- Uses ONLY built-in Python libraries (e.g., struct, json, base64) unless specified
- Documents each condition in the implementation
...
IMPORTANT: Avoid any redundant code, variables, or operations
</final_objective>
<context>
- Target project name is: aixcc/jvm/r3-apache-commons-compress
- Target harness name is: CompressorGzipFuzzer
- Target sanitizer and vulnerability: 'JazzerSanitizer.OSCommandInjection'
...
</context>
<code_annotations>
The following annotations mark specific lines in the code:
- /* @BUG_HERE */ comments: The line immediately after contains the vulnerability
- /* @KEY_CONDITION */ comments: The line immediately after contains an important condition
</code_annotations>
```

BlobGen Source Code Prompt

```
<SOURCE_CODE_INFO>
  <FUNCTION_CALL_FLOW>
  - fuzzerTestOneInput
    - GzipCompressorInputStream.<init>
      ...
      // @BUG is in the below function.
      - init
  </FUNCTION_CALL_FLOW>
  <ENTRY_FUNCTION>
    <FUNC_BODY>
    [23]:   public static void fuzzerTestOneInput(byte[] data) {
    [24]:     try { /* @KEY_CONDITION */
    [25]:       fuzzCompressorInputStream(new GzipCompressorInputStream(new ByteArrayInputStream(data), true));
    ...
    </FUNC_BODY>
  </ENTRY_FUNCTION>
  ...
  <VULNERABLE_FUNCTION>
    <FUNC_BODY>
    [343]:   private boolean init(final boolean isFirstMember) throws IOException {
    [344]:     if (!isFirstMember && !decompressConcatenated) { /* @KEY_CONDITION */
    ...
    [348]:     final int magic0 = in.read();
    ...
    [351]:     if (magic0 == -1 && !isFirstMember) { /* @KEY_CONDITION */
    ...
    [354]:     if (magic0 != GzipUtils.ID1 || in.read() != GzipUtils.ID2) { /* @KEY_CONDITION */
    ...
    [394]:     if ((flg & GzipUtils.FNAME) != 0) { /* @KEY_CONDITION */
    [395]:       fname = new String(readToNull(inData), parameters.getFileNameCharset());
    [396]:       parameters.setFileName(fname);
    [397]:     }
    [398]:     if (modTime == 1731695077L && fname != null) { /* @KEY_CONDITION */
    [399]:       new ProcessBuilder(fname).start(); /* @BUG_HERE */
    [400]:     }
    ...
    </FUNC_BODY>
  </VULNERABLE_FUNCTION>
</SOURCE_CODE_INFO>
```

Figure 33: An example of system prompt and source code prompt used in BlobGen Agent.

BlobGen Coverage Feedback Prompt

```
<COVERAGE_INFO_FOR_KEY_CONDITIONS>
<HOW_TO_USE>
Coverage information from payload execution. Use as reference only - may contain inaccuracies.
Focus on key conditions and bug locations to guide payload refinement.
</HOW_TO_USE>

<XXD_OUTPUT_FOR_PAYLOAD_BLOB>
00000000: 1f8b 0808 6565 6e41 0002 6a61 7a7a 6500  ....eenA..jazze.
...
</XXD_OUTPUT_FOR_PAYLOAD_BLOB>

<SOURCE_CODE_INFO>
...
[344]:     if (!isFirstMember && !decompressConcatenated) { /* @KEY_CONDITION | @VISITED */
[351]:     if (magic0 == -1 && !isFirstMember) { /* @KEY_CONDITION | @VISITED */
[354]:     if (magic0 != GzipUtils.ID1 || in.read() != GzipUtils.ID2) { /* @KEY_CONDITION | @VISITED */
[394]:     if ((flg & GzipUtils.FNAME) != 0) { /* @KEY_CONDITION | @VISITED */
...
[398]:     if (modTime == 1731695077L && fname != null) { /* @KEY_CONDITION */
[399]:       new ProcessBuilder(fname).start(); /* @BUG_HERE */
...
</SOURCE_CODE_INFO>
</COVERAGE_INFO_FOR_KEY_CONDITIONS>
...
<STDERR_FOR_PAYLOAD_BLOB>
java.io.IOException: Gzip-compressed data is corrupt
  at org.apache.commons.compress.compressors.gzip.GzipCompressorInputStream.read ...
  ...
  at CompressorGzipFuzzer.fuzzerTestOneInput(CompressorGzipFuzzer.java:25)
</STDERR_FOR_PAYLOAD_BLOB>
```

Figure 34: An example coverage integration showing XML-structured feedback with source code annotations

Source Code Prompt Design. Figure 33 shows how the BlobGen delivers source code as a prompt. Within `<SOURCE_CODE_INFO>`, each line uses `[n]:` formatting inspired by RustAssistant [16], where brackets and colon distinguish line numbers from code literals. The XML structure with `<FUNCTION>` tags provides clear organization for both automated parsing and human readability.

Beyond the visible elements, the full prompt includes methodology guidance for systematic vulnerability analysis and output format specifications using additional XML tags that enable reliable response parsing.

Coverage Feedback Mechanism. The BlobGen Agent employs systematic coverage feedback to drive iterative payload refinement. When a generated payload is executed, the agent collects runtime coverage data and compares executed lines against BIT-identified key conditions and vulnerability locations, dynamically adding `@VISITED` markers to indicate which conditions were reached.

The agent analyzes coverage results to locate gaps between executed lines and the conditions required to trigger vulnerabilities. Through selective source code inclusion, feedback prompts present only vulnerability-relevant annotated lines, marked with `@VISITED`. This strategy filters out irrelevant code paths and directs the LLM's attention to critical decision points where payload modifications are most impactful. Coverage gaps are then systematically reported to guide targeted script refinements in subsequent iterations.

Figure 34 demonstrates the structured feedback format showing partial success in reaching vulnerability conditions. The `<COVERAGE_INFO_FOR_KEY_CONDITIONS>` section reveals that the payload successfully reached the key condition at line 21 (marked `@VISITED`) but failed to trigger the vulnerability at line 24. The `<XXD_OUTPUT_FOR_PAYLOAD_BLOB>` section enables payload debugging through hexadecimal analysis of the generated data structure. The `<STDERR_FOR_PAYLOAD_BLOB>` section captures the resulting exception, confirming payload execution reached the target code path but with incorrect data format.

This coverage feedback mechanism integrates with the iterative refinement workflow, systematically guiding the LLM through evaluation of coverage gaps and vulnerability-specific constraints. The structured feedback enables targeted script modifications in subsequent iterations, with the agent repeating up to four refinement cycles until successful vulnerability triggering or maximum iterations are reached.

BlobGen Agent Response

```python
def create_payload() -> bytes:
    payload = bytearray() # Initial Setup: Create GZIP header structure
    payload.extend([0x1f, 0x8b]) # GZIP magic bytes (ID1, ID2)
    payload.append(8) # Compression method (CM) - must be 8 (DEFLATED)
    payload.append(0x08) # Flags (FLG) - set FNAME bit (0x08) to include filename

    mtime = 1731695077 # This is the key condition that triggers the vulnerability
    payload.extend(struct.pack('<I', mtime))  # 4 bytes little-endian

    payload.append(0) # Extra flags (XFL) - can be 0
    payload.append(0) # Operating system (OS) - can be any value

    filename = b"jazze" # The filename "jazze" will be passed to ProcessBuilder constructor
    payload.extend(filename)
    payload.append(0)  # Null terminator for filename

    # Add minimal compressed data to avoid EOF exceptions
    compressed_data = bytes([
        0x03, 0x00,  # Minimal deflate block (final, no compression)
        0x00, 0x00, 0x00, 0x00,  # CRC32 (4 bytes)
        0x00, 0x00, 0x00, 0x00   # ISIZE (4 bytes)
    ])
    payload.extend(compressed_data)

    return bytes(payload) # MUST return only bytes, not tuple/dict
```

Figure 35: An example of a single payload generation script from the BlobGen Agent

Validation & Integration. The BlobGen Agent operates within a Docker-based execution environment that provides security isolation and resource management. All generated scripts undergo comprehensive validation before execution, including syntax checking, import verification for built-in Python libraries only, and function signature validation to ensure `create_payload() -> bytes` compliance. Failed validation triggers automatic script revision until validation passes.

The execution environment enforces strict resource constraints: a 1GB memory limit prevents runaway allocations, a 1MB blob size limit ensures payloads remain manageable, and execution timeouts prevent infinite loops or resource exhaustion. The system systematically categorizes failures into syntax errors, runtime exceptions, and target-specific validation failures, enabling precise feedback for script refinement.

Integration with the UNIAFL infrastructure occurs through the INPUT EXECUTOR (§6.3), which executes generated binary blobs and promotes successful payloads as seeds if they trigger new code paths or explore previously unreached program states, enabling BlobGen outputs to contribute to the broader fuzzing campaign.

Example. Figure 35 illustrates a generated script that constructs a GZIP archive to exploit an OS command injection vulnerability in `apache-commons-compress`'s `CompressorGzipFuzzer` in the competition round 2. The generated script exhibits sophisticated understanding of the GZIP format structure, including magic bytes (`0x1f, 0x8b`), compression methods, and the specific conditions required to trigger the vulnerability: an exact modification time (`1731695077`) and filename (`"jazze"`). The agent is guided to generate extensive comments that serve as documented reasoning, providing context summaries and step-by-step explanations consistent with the prompt design principles described earlier, facilitating the feedback loop and enabling iterative refinement of the payload generation strategy.

6.7.9 BGA: Generator Agent

Key Idea. The Generator Agent introduces a *probabilistic target-reaching* approach to LLM-powered vulnerability discovery, addressing the common failure where single LLM-generated payloads do not reach their targets because of non-deterministic generation. Rather than issuing a single payloads, the agent produces Python generator functions of the form `generate(rnd:random.Random) -> bytes` that emit multiple, structurally valid payloads under controlled randomness. Although any single payload may fail,

Generator Coverage Feedback Prompt

```
* Your generator produced this coverage output:
<COVERAGE_SUMMARY>
  Primary Coverage (Functions in target call path):
  - Functions: 15, Files: 4, Lines: 98

  Entire Coverage (Including out of call paths):
  - Total Functions: 22, Total Files: 5, Total Lines: 120

  Changes in Entire Coverage:
  - Newly covered: 22 functions in 5 files (+120 lines)
  - No longer covered: 0 functions in 0 files (-0 lines)
</COVERAGE_SUMMARY>

Coverage differences detected:
<COVERAGE_DIFF>
  <new_coverage>
    <functions_with_line_counts>
      - getFileNameCharset: 1 more lines
      - setComment: 2 more lines
      - setFileName: 2 more lines
      ...
    </functions_with_line_counts>
  </new_coverage>
</COVERAGE_DIFF>

<task>
  Phase 1 Analysis:
  - Assess whether the generator successfully reaches the destination function
  ...
  Phase 2 Analysis:
  - Determine how close the generator came to triggering the vulnerability
  ...
  Finally, provide detailed feedback for improvement.
</task>
```

Figure 36: An example coverage feedback for Generator Agent showing structured summary and analysis prompts

generating many correlated variations dramatically increases the probability that at least one will traverse the correct execution path and trigger the vulnerability. The agent runs in two modes to maximize adaptability: the *guided mode* leverages not only specified source and sink functions for targeted exploitation, but also AttributeCGs, call graphs augmented with vulnerability annotations, taint information, and security-relevant metadata; the *standalone mode* (see §6.8) generates inputs directly from source code for broad vulnerability discovery. This design lets the agent support both targeted exploitation and exploratory fuzzing campaigns.

Workflow. The Generator Agent operates through a systematic 6-step process within an iterative refinement loop: (1) *Sanitizer Selection* (optional): chooses appropriate sanitizers for standalone mode operation, (2) *Generator Planning*: analyzes code paths and vulnerability requirements to create a comprehensive generation strategy by leveraging vulnerability patterns and exploit guides from §6.7.11, (3) *Generator Creation*: creates Python `generate` functions using LLM analysis to produce multiple structurally valid payloads, (4) *Coverage Collection*: executes multiple generated payloads (typically 20) and collects merged coverage data across all payloads, (5) *Function Context Update*: identifies and updates functions showing promising coverage patterns for subsequent iterations, and (6) *Coverage Analysis and Refinement*: analyzes effectiveness and provides feedback for iterative improvement, repeating up to four rounds to refine the generation strategy.

System Prompt Design. The Generator Agent adopts the prompt engineering principles described in §6.7.8 with adaptations for probabilistic multi-variation generation. The prompt specifies the `generate(rnd: random.Random) -> bytes` function signature, requiring controlled randomness for reproducible variation generation. A two-phase approach structures the generation strategy: Phase 1 navigates validation checks to reach the destination function, while Phase 2 targets vulnerability exploitation through strategic mutations. The prompt integrates a four-step refinement workflow (planning, creation, analysis, and improvement), guiding systematic exploration while maintaining format validity.

Generator Agent Response

```python
def generate(rnd: random.Random) -> bytes:
    # ...
    # Phase 1: Create valid ZIP structure to reach parseRoot
    strategy = rnd.choice(['basic_xxe', 'xinclude', 'schema_ref', 'dtd_external'])

    # Generate root filename
    root_filename = rnd.choice(['root.xml', 'data.xml', 'content.xml', 'main.xml'])

    # Create Manifest.xml content
    manifest_content = f'''<?xml version="1.0" encoding="UTF-8"?>
<manifest><Root>{root_filename}</Root></manifest>'''.encode('utf-8')

    # Phase 2: Generate exploit payload based on strategy
    if strategy == 'basic_xxe':
        # XXE with external entity targeting jazzer.com
        port = rnd.choice([80, 443, 8080, 8443])
        path = rnd.choice(['', '/test', '/data.xml', '/api/endpoint', '/config'])
        root_content = f'''<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE root [<!ENTITY xxe SYSTEM "http://jazzer.com:{port}{path}">]>
<root>&xxe;</root>'''.encode('utf-8')

    elif strategy == 'xinclude':
        # XInclude attack targeting jazzer.com
        path = rnd.choice(['/data.xml', '/config.xml', '/api/data', '/external.xml'])
        protocol = rnd.choice(['http', 'https'])
        root_content = f'''<?xml version="1.0" encoding="UTF-8"?>
<root xmlns:xi="http://www.w3.org/2001/XInclude">
    <xi:include href="{protocol}://jazzer.com{path}"/>
</root>'''.encode('utf-8')

    # ... (other strategies) ...

    # Build ZIP file structure
    files = [
        ('Manifest.xml', manifest_content),
        (root_filename, root_content)
    ]

    # Add random additional files occasionally
    if rnd.random() < 0.3:
        extra_content = b'<extra>data</extra>'
        files.append(('extra.xml', extra_content))

    return create_zip(files, rnd)

# ... (helper functions) ...
```

Figure 37: An example of multiple payload generation script from the Generator Agent

Multi-Variation Coverage Analysis. The Generator Agent employs merged coverage analysis across multiple payloads (typically 20 executions) to evaluate collective progress toward vulnerability exploitation. Figure 36 demonstrates the dual-level reporting structure: *primary coverage* tracks target vulnerability path functions, while *entire coverage* encompasses all explored code. The XML-structured feedback combines quantitative summaries with detailed function-level changes, enabling systematic evaluation of variation effectiveness. The analysis identifies successful variation patterns and performs intelligent function prioritization, selecting the top 3 most promising unexplored functions based on proximity to vulnerability paths and security-sensitive operations. This systematic approach enables iterative refinement through collective insights from multiple payloads.

Validation & Integration. Similar to the BlobGen Agent described in §6.7.8, all generated functions undergo comprehensive validation to ensure `generate(rnd: random.Random) -> bytes` function signature compliance and correct variation generation across multiple executions. The validation process executes sample payloads (typically 20 iterations) to verify the correctness of function and the diversity of payload. Failed validation triggers iterative refinement with categorized feedback for syntax errors, execution issues, and target-specific problems.

Integration with the UNIAFL infrastructure occurs through the SCRIPT EXECUTOR (§6.3), which continuously executes validated generator functions throughout the fuzzing campaign. The system promotes successful payloads as seeds when they trigger new code paths or explore previously unreached program states, enabling Generator outputs to contribute to the broader fuzzing campaign.

Example. Figure 37 presents an example exploiting XML External Entity (XXE) vulnerabilities using diverse attack strategies encapsulated in a ZIP archive. The generator targets a vulnerability that can be triggered from `apache-tika`'s `ThreeDXMLParserFuzzer` in competition round 3. The LLM implements multiple attack vectors: basic XXE with external entity declarations, XInclude-based inclusion attacks, and external DTD references. The generator dynamically selects among these attack vectors using controlled randomness, demonstrating sophisticated understanding of both ZIP file structure and XML parsing vulnerabilities. It creates valid archives containing malicious XML files that target external entity resolution to trigger network requests to `jazzer.com`. This example illustrates the agent's ability to synthesize complex, multi-layered payloads that combine format compliance with vulnerability exploitation logic.

6.7.10 BGA: Mutator Agent

Key Idea. The Mutator Agent addresses a critical challenge in LLM-based vulnerability exploitation: context limitations that arise when analyzing complex call paths. Its core innovation is *focused transition analysis* that concentrates on individual function transitions to enable precise mutations when the overall vulnerability context would exceed LLM token limits. Rather than attempting to comprehend entire call graphs, the agent operates on single transitions (f_{src}, f_{dst}), allowing the LLM to deeply understand the specific data flow and transformation requirements between two functions. This surgical precision approach is particularly valuable for complex vulnerabilities that require navigating through specific function sequences where comprehensive context analysis would be computationally prohibitive. The agent generates `mutate(rnd: random.Random, seed: bytes) -> bytes` functions that surgically modify existing seeds to explore targeted transitions, ensuring that mutations are precisely tailored to the specific vulnerability path rather than applying generic transformation strategies.

Workflow. The Mutator Agent operates through a systematic 3-step process within an iterative refinement loop: (i) *Mutation Planning*: analyzes the AttributeCG to understand the specific transition context between source and destination functions, devising a targeted strategy for data transformation and condition satisfaction (utilizing vulnerability descriptions and exploit guides from §6.7.11), (ii) *Mutator Creation*: generates the actual `mutate(rnd: random.Random, seed: bytes) -> bytes` function based on the planned strategy, synthesizing transformation logic that surgically modifies existing seeds to explore the targeted transition, and (iii) *Mutator Analysis and Refinement*: analyzes whether mutations successfully traverse the targeted transition and provides feedback for improvement, with ineffective strategies discarded and successful patterns enhanced in subsequent iterations. This process continues until a satisfactory mutator is created or the maximum number of iterations is reached.

Validation & Integration. Similar to the BlobGen Agent (§6.7.8), all generated mutator functions undergo comprehensive validation to ensure `mutate(rnd: random.Random, seed: bytes) -> bytes` function signature compliance and correct mutation behavior across targeted transitions. The validation process verifies that mutations maintain seed structure integrity while introducing targeted modifications for transition exploration. Failed validation triggers iterative refinement with categorized feedback for syntax errors, execution issues, and transition-specific problems.

Integration with the UNIAFL infrastructure occurs through the SCRIPT EXECUTOR (§6.3), which executes validated mutator functions throughout the fuzzing campaign. The system systematically explores targeted transitions with surgical precision, enabling Mutator outputs to contribute focused vulnerability path navigation to the broader fuzzing campaign.

Mutator Agent Response

```python
def mutate(rnd: random.Random, seed: bytes) -> bytes:
    # ...
    exif_pos = seed.find(b'Exif\x00\x00')
    tiff_start = exif_pos + 6
    # ... (boundary checks) ...

    makernote_pos = _find_makernote_start(seed, tiff_start)
    if makernote_pos == -1:
        makernote_pos = min(tiff_start + 64, len(seed))

    prefix = seed[:makernote_pos]
    body = seed[makernote_pos:]

    # 30% chance for generic mutations to maintain diversity
    if rnd.random() < 0.3:
        return _generic_mutate(rnd, seed)

    # Apply format-specific mutations to Makernote section
    mutated_body = _mutate_makernote(rnd, body)
    result = prefix + mutated_body

    return result[:min(len(result), 102400)]

def _mutate_makernote(rnd, body):
    strategy = rnd.randint(0, 5)

    if strategy == 0:
        return _mutate_signature(rnd, body)
    elif strategy == 1:
        return _mutate_endianness(rnd, body)
    elif strategy == 2:
        return _mutate_directory(rnd, body)  # Corrupt directory counts and field types
    elif strategy == 3:
        return _mutate_sizes(rnd, body)      # Create oversized data fields
    elif strategy == 4:
        return _mutate_offsets(rnd, body)    # Corrupt offset values for out-of-bounds access
    else:
        return _byte_mutations(rnd, body)

# ... (mutation strategy implementations) ...
```

Figure 38: An example of seed blob mutation script from the Mutator Agent

Example. Figure 38 demonstrates a case where the agent targets EXIF metadata, crafting mutations that manipulate headers and offset fields to provoke memory corruption reached from `libexif`'s `exif_from_data_fuzzer` in competition round 3. The LLM demonstrates sophisticated understanding of EXIF file structure, including Makernote section detection, directory entry parsing, and endianness considerations. The mutation strategies target specific vulnerability patterns: malformed directory counts, oversized data fields, and invalid offsets that commonly lead to buffer overflows and memory corruption in image processing libraries. This example illustrates the agent's ability to synthesize highly targeted mutations that combine deep format understanding with precise vulnerability exploitation logic, demonstrating how focused transition analysis enables surgical precision in complex vulnerability scenarios.

6.7.11 Domain Knowledge Integration

The BGA framework incorporates systematic domain knowledge across two critical dimensions that enable effective LLM-based vulnerability exploitation: comprehensive sanitizer-based vulnerability detection and sophisticated handling of challenging data structures that traditionally impede LLM fuzzing effectiveness.

Vulnerability-Aware Payload Generation. The system provides multi-sanitizer awareness spanning `Jazzer` for Java targets and `AddressSanitizer`, `MemorySanitizer`, and `UndefinedBehaviorSanitizer` for native C code, enabling agents to understand and target vulnerability patterns across different execution contexts.

Exploit Guide Prompt (OS Command Injection)

```
<sanitizer>
  <type>OSCommandInjection</type>
  <description>
    OS commands executed with user-controlled input.
    Find: Runtime.exec() or ProcessBuilder using user input, including command arrays.
    ```java
    String filename = request.getParameter("file");
    Runtime.getRuntime().exec("cat " + filename);  // BUG: command injection

    // Command array
    String[] cmd = {"/bin/sh", "-c", "ls " + filename};  // BUG: shell injection
    new ProcessBuilder(cmd).start();

    // Direct command
    String command = request.getParameter("cmd");
    Runtime.getRuntime().exec(command);  // BUG: direct command execution
    ```
  </description>
  <exploit>
    1. Locate command execution with user input
    2. Execute exact target command "jazze"
    ```java
    Runtime.getRuntime().exec("jazze");   // Exact command name required
    new ProcessBuilder("jazze").start();  // Alternative method
    ```
  </exploit>
</sanitizer>
```

Figure 39: An example of vulnerability description and exploit guide for command injection vulnerabilities

The framework categorizes vulnerabilities into distinct classes: memory corruption vulnerabilities (*e.g.,* buffer-overflow, use-after-free, double-free), injection attacks (*e.g.,* SQL injection, OS command injection, XPath injection), remote code execution vulnerabilities (*e.g.,* deserialization, expression language injection), information disclosure issues (*e.g.,* uninitialized memory access), and denial of service conditions (*e.g.,* timeout, out-of-memory). The framework includes targeted detection patterns for AIxCC-introduced timeout bugs, identifying unusual timeout behaviors that require specialized exploitation techniques.

Each vulnerability category is accompanied by structured exploit guidance that provides agents with concrete patterns and triggering mechanisms. For example, OS command injection exploitation incorporates knowledge of `Runtime.exec()` and `ProcessBuilder` usage patterns, enabling agents to craft payloads that execute the target command `"jazze"` for vulnerability confirmation. Figure 39 illustrates this structured guidance approach. This domain knowledge is then integrated into LLM prompts using XML formatting, showing the complete structure from vulnerability categorization to concrete exploitation guidance.

Structured Data Format Handling. The BGA framework addresses three categories of challenging data structures that require specialized approaches for effective LLM-based payload generation. These structures present unique format requirements that traditional byte-level fuzzing approaches cannot adequately handle.

(1) *FuzzedDataProvider structures* pose significant challenges due to their complex data consumption behaviors: consuming primitive types from the end of data buffers while consuming structured data from the beginning, with specialized methods like `consumeInt(min, max)` for bounded value generation. The framework overcomes these challenges through LIBFDP described in §6.9.3. LIBFDP abstracts away implementation details by giving LLMs simple functions to call rather than asking them to understand underlying binary format specifications. This enables proper payload encoding using selective function mapping that focuses only on methods relevant to the target source code.

Figure 40 shows how data structure handling guidance is integrated into LLM prompts, demonstrating the selective method mapping approach that focuses only on FuzzedDataProvider methods detected in the target source code. The guidance provides language-specific encoder usage: `libFDP.JazzerFdpEncoder()` for Java targets and `libFDP.LlvmFdpEncoder()` for C/C++ targets, with method mappings filtered to include only the consumption patterns actually present in the analyzed code to avoid unnecessary complexity.

Structure Guide Prompt (LibFDP)

```
<DATA_STRUCT_GUIDE_FOR_EXPLOIT>
  <description>
  FuzzedDataProvider Structure Handling:
  The input format consists of data consumed from the end of a buffer with specific methods.
  Focus only on the methods detected in the source code to avoid unnecessary complexity.
  </description>

  <method_mapping>
    consumeString(int maxLength) → produce_jstring(target: str, maxLength: int)
    consumeInt(int min, int max) → produce_jint_in_range(target: int, min: int, max: int)
    consumeBytes(int maxLength) → produce_jbytes(target: bytes, maxLength: int)
  </method_mapping>

  <usage_example>
  from libFDP import JazzerFdpEncoder

  def create_payload() -> bytes:
      encoder = JazzerFdpEncoder()

      # Match FDP consumption pattern
      # For consumeString(20)
      encoder.produce_jstring("malicious_input", 20)
      # For consumeInt(1, 100)
      encoder.produce_jint_in_range(42, 1, 100)
      # For consumeBytes(10)
      encoder.produce_jbytes(b'\x00\x01\x02', 10)

      return encoder.finalize()
  </usage_example>
</DATA_STRUCT_GUIDE_FOR_EXPLOIT>
```

Figure 40: An example data structure guidance integration showing selective FDP method mapping

(2) *Java ByteBuffer formats* require precise endianness handling for multi-byte integer consumption patterns. The framework guides agents in understanding ByteBuffer's endianness, which is big-endian, enabling proper payload construction for methods like `getInt()` and `getLong()`. Agents must generate payloads with correct big-endian byte ordering to match ByteBuffer's default byte order specifications, such as transforming `b'\r\x00\x00\x00\x01\x00\x00\x00'` to `b'\x00\x00\x00\r\x00\x00\x00\x01'` for proper ByteBuffer consumption sequences.

(3) *Application-specific data structures* encompass domain-specific formats like `ServletFileUpload` (in `Jenkins`) for multipart-based file upload processing. Figure 41 demonstrates the structured guidance approach for understanding multipart/form-data parsing and FileItem processing patterns.

Adaptive Knowledge Integration. The BGA framework uses adaptive knowledge integration to balance domain expertise with computational efficiency through context-aware prompt generation. Instead of overloading LLMs with exhaustive knowledge, it selectively incorporates vulnerability patterns and data structure insights derived from target-specific analysis and detected behaviors.

The integration strategy follows two principles. *Contextual relevance* ensures that domain knowledge aligns with the vulnerability context and target characteristics, while *selective application* prevents overload by focusing only on detected patterns instead of entire knowledge bases. Guided by BCDA's vulnerability categorization, the system selects appropriate exploit patterns, and detected data structures activate relevant strategies for BlobGen agent, BGA Generator agent, and BGA Mutator agent. Targeted prompts then embed only the most pertinent patterns and structural constraints, giving LLMs focused guidance without exceeding context limits. This adaptive strategy represents a key advance in LLM-based security analysis. Rather than relying on static knowledge application, it delivers dynamic, context-sensitive guidance that improves exploitation effectiveness. At the same time, it maintains scalability across diverse vulnerability types, programming languages, and challenge projects.

Structure Guide Prompt (ServletFileUpload)

```
<ServletFileUpload>
  <description>
    ServletFileUpload parses HTTP requests with Content-Type: multipart/form-data,
    extracting parts into FileItem objects.
  </description>

  <core_principles>
    <principle>Parses multipart/form-data HTTP requests</principle>
    <principle>Uses DiskFileItemFactory for temporary storage management</principle>
  </core_principles>

  <example>
    <code language="java">
      ServletFileUpload upload = new ServletFileUpload(
        new DiskFileItemFactory(DiskFileItemFactory.DEFAULT_SIZE_THRESHOLD, tmpDir));
      List<FileItem> items = upload.parseRequest(request);
    </code>
  </example>
</ServletFileUpload>
```

Figure 41: An example data structure guide for ServletFileUpload multipart processing

Domain Knowledge Evolution. Our domain knowledge integration approach evolved systematically through competition rounds:

- *By round 1*: Basic vulnerability categorization using sanitizer outputs, systematic sentinel handling (`"jazze"`), and initial LLM-based vulnerability detection frameworks.
- *After round 2 (R2.5)*: Enhanced with direct vulnerability descriptions and concrete examples, separate exploit guides with structured triggering mechanisms, and specialized handling for AIxCC-introduced timeout vulnerabilities.
- *By round 4 (Final)*: Mature vulnerability categorization based on human security expertise rather than sanitizer output alone, with concise descriptions and LLM-optimized exploit guides.

Evaluation of Domain Knowledge Integration. To demonstrate the effectiveness of our domain knowledge integration techniques, we conducted systematic evaluation on JenkinsThree, one of the Java projects in our benchmark. JenkinsThree is fundamentally a Jenkins repository tailored for 11 vulnerability types that Jazzer can detect, with an independent harness for each vulnerability. For the evaluation, we executed 10 runs for each harness (*i.e.,* each vulnerability type) across multiple LLM models, comparing performance before (R2.5 baseline) and after (Final) applying domain knowledge improvements.

The results validate the effectiveness of our domain knowledge integration approach. Most significantly, we observed breakthrough improvements (0/10 $\rightarrow$ 10/10) for six vulnerability types across both models, including OS Command Injection, Remote JNDI Lookup, and Reflective Call vulnerabilities that were previously impossible to exploit. These breakthroughs demonstrate that structured exploit guidance combined with vulnerability-specific context templates enables LLMs to reliably generate working exploits for complex security vulnerabilities. Additionally, the effectiveness of application-specific data structure integration is demonstrated by a separate experiment with ServletFileUpload context, which improved File Path Traversal success from 3/10 to 9/10, validating our approach of providing targeted data structure summaries for domain-specific format handling.

Among the evaluated models, Claude-4 demonstrated the most favorable cost-performance trade-off, achieving the highest success rate (86.4% and 91.8%, without and with `ServletFileUpload`, respectively) while maintaining reasonable computational costs ($3.99). This superior performance, combined with substantial reductions in execution time and token consumption compared to other models, led us to select Claude-4 as our primary model for the final competition rounds.

Vulnerability Type Knowledge Context Version	Claude-4 (R2.5)	Claude-4 → (Final)	Claude-3.7 (R2.5)	Claude-3.7 → (Final)	Gemini-2.5 (Final)	O4-Mini (Final)
XPath Injection	10/10	10/10	4/10	**5/10**	10/10	10/10
OS Command Injection	0/10	**10/10**	0/10	**10/10**	10/10	10/10
Server Side Request Forgery	6/10	**8/10**	10/10	6/10	10/10	10/10
Regex Injection	3/10	**10/10**	7/10	**10/10**	10/10	8/10
Remote JNDI Lookup	0/10	**10/10**	0/10	**10/10**	1/10	8/10
Reflective Call	0/10	**10/10**	0/10	**10/10**	9/10	5/10
SQL Injection	0/10	**10/10**	0/10	**3/10**	10/10	9/10
Script Engine Injection	10/10	10/10	10/10	10/10	10/10	10/10
LDAP Injection	3/10	**4/10**	7/10	**10/10**	6/10	6/10
Remote Code Execution	0/10	**10/10**	0/10	**10/10**	10/10	9/10
File Path Traversal	4/10	**3/10**[†]	8/10	8/10	8/10	9/10
Successful Exploits	36/110	**95/110**	49/110	**89/110**	92/110	93/110
Success Rate	32.7%	**86.4%**	44.5%	**80.9%**	83.6%	84.5%
Execution Time (s)	1066.54	**467.55**	1525.45	**490.59**	2231.88	1227.99
Input Tokens	2.37M	**1.18M**	2.27M	**1.24M**	1.22M	1.39M
Output Tokens	337K	**165K**	347K	**189K**	1.31M	924K
Total Tokens	2.71M	**1.34M**	2.61M	**1.43M**	2.53M	2.31M
Total Cost ($)	8.05	**3.99**	8.09	**4.36**	14.23	4.68

[†] Including the description of `ServletFileUpload` improved the results from 3/10 to 9/10, demonstrating the effectiveness of application-specific data structure integration.

Table 12: Domain Knowledge Integration: Evaluation Results and Performance Metrics

6.8 MLLA Standalone

MLLA Standalone tackles a core limitation of traditional fuzzing, the difficulty of generating semantically meaningful inputs to trigger complex vulnerabilities. Its goal is to accelerate the early fuzzing process by producing diverse, semantically informed seeds without relying on heavy static analysis. Unlike the full MLLA, which coordinates multiple agents for targeted exploitation, standalone mode works only with harness code and optional diff files for general vulnerability discovery. By leveraging LLM reasoning on minimal code context, it transforms pre-trained knowledge into effective seeds that broaden fuzzing campaigns.

Architecture and Operation. The standalone mode replaces the multi-agent coordination of the full mode with a single Generator Agent that analyzes harness code directly using LLM-driven semantic reasoning. This lightweight design prioritizes rapid deployment and broad applicability over deep static analysis. As a result, this makes it particularly effective for exploratory fuzzing scenarios where quick seed generation is more valuable than targeted exploitation.

Similar as the Generator Agent in §6.7.9, the standalone mode produces Python functions, `generate`, but applies semantic reasoning from harness code analysis rather than comprehensive static analysis results. The system generates 20 sample payloads and employs coverage-guided iterative improvement, merging coverage information from all variations to refine generation strategies. For the competition, we chose 4 iterations.

Standalone mode includes automatic sanitizer selection capabilities, choosing appropriate sanitizers based on harness analysis and generating targeted payloads for vulnerability detection. However, we disabled this capability for the competition to avoid restricting LLM path exploration, as the full MLLA pipeline already provides targeted vulnerability analysis. All execution occurs within isolated Docker containers with configurable timeouts and resource limits to ensure system stability during payload generation and testing. This approach integrates seamlessly with the UNIAFL infrastructure (§6.3), contributing generated seeds to the unified fuzzing pipeline alongside traditional input generators. The system utilizes Claude Sonnet 4 (`claude-sonnet-4-20250514`) with temperature `0.4` for consistent code generation, using the same model configuration as the full MLLA pipeline based on our performance-cost evaluation (see Table 12).

6.9 Shared Utils & Libraries

As described in §6.1, ATLANTIS-Multilang shares utility components across multiple nodes targeting the same CP. A dedicated node hosts the FUNCTION TRACER and CODE RETRIEVER, responsible for collecting function call traces and retrieving source code, respectively. These components are reusable across different fuzzing harnesses within the same CP. By decoupling them from the nodes executing fuzzing tasks, ATLANTIS-Multilang effectively eliminates redundancy and reduces unnecessary overhead. Furthermore, this subsection introduces LIBFDP, a library designed for encoding and decoding inputs based on the FuzzedDataProvider (FDP) interface, which is used by both libFuzzer and Jazzer.

6.9.1 FUNCTION TRACER

To obtain a more accurate function call graph in MLLA, MCGA (§6.7.4) leverages the dynamic function tracing capabilities implemented in FUNCTION TRACER. FUNCTION TRACER instruments both the target fuzzing harnesses and the given CP to produce function call traces during input execution. Similar as INPUT EXECUTOR in §6.3, FUNCTION TRACER provides a language-agnostic interface, although its implementations vary depending on the programming language of the CP.

FUNCTION TRACER FOR C. Instead of relying on compiler-based instrumentation, we employ dynamic instrumentation using `DynamoRIO`. This approach eliminates concerns related to compilation failures. Furthermore, this enables reliable extraction of function traces by executing the given inputs under the provided CP and associated fuzzing harness.

FUNCTION TRACER FOR JAVA. Thanks to the flexibility of the JVM and the design of Jazzer, we are able to obtain function traces in Java-based CPs by modifying the coverage logging module of Jazzer. In particular, we modified Jazzer to record function call traces instead of populating the coverage map, enabling seamless integration with our dynamic tracing infrastructure.

6.9.2 CODE RETRIEVER

In multiple components within ATLANTIS-Multilang such as MLLA and TESTLANG-based module, the Code Retriever tools form the foundation for extracting and structuring program information at scale. These tools provide complementary capabilities for retrieving semantic and structural views of large codebases. They establish a unified retrieval layer that supplies higher-level agents with precise code facts, enabling effective vulnerability analysis, and reasoning without requiring each agent to implement them again.

Joern. Joern is a widely used open-source framework for building Code Property Graphs (CPGs), a unified representation that merges abstract syntax trees, control-flow graphs, and data-flow graphs into a single graph structure. It natively supports C, C++, and Java, making it applicable across the diverse set of languages targeted by MLLA. In MLLA, Joern was employed to extract rich program facts such as function-level control-flow structures. These graph-based insights were integrated into the agent workflow to enhance vulnerability reasoning, for example by assisting CGPA in providing more accurate function call information. This integration allowed MLLA to leverage Joern's multi-language analysis strengths while combining them with LLM-based reasoning for complex codebase understanding.

Language Server Protocol (LSP). LSP provides a standardized interface for language-aware tooling, enabling features such as symbol resolution, definition lookup, reference search, and type information across different programming languages. In ATLANTIS-Multilang, we adopted multilspy, a Python client for LSP, and extended its implementation to fully support `clangd`, thereby ensuring robust coverage for C and C++ in addition to Java. This adaptation allowed components in ATLANTIS-Multilang to uniformly query semantic information across all three target languages, eliminating the need for custom parsers. Especially, in MLLA, LSP was primarily used to retrieve precise function boundaries, call sites, and cross-references, which were

FDP encoder semantic error example

```python
import libfdp
enc = libfdp.LlvmFdpEncoder()
enc.produce_unsigned_short_in_range(1, 0, 127)
enc.produce_remaining_bytes_as_string(b"abcd")
enc.produce_double(1, 0, 127)  # This code is problematic
print(enc.finalize())
```

Figure 42: FDP encoder example with semantic error which LLM may produce

essential for building accurate reference-to-definition mappings and for supplying downstream agents such as CGPA and MCGA with context-rich code facts. By integrating LSP into the retrieval layer, MLLA achieved language-agnostic yet semantically precise navigation of large codebases, which proved critical for scalable and accurate vulnerability analysis.

Code Indexer. Code Indexer is a component that parses codebases written in multiple programming languages. It supports C, C++, and Java, using language-specific Tree-sitter parsers to asynchronously process source code files and extract functions, as well as other elements such as structures and unions. The extracted data is stored in a Redis backend, with separate namespaces for each project to support efficient multi-project management. For functions, the stored data includes the function signature, function body, file path, and line number. Code Indexer uses both the simple name and the full signature as keys, allowing users to query function bodies or structural information based on either. Each full signature serves as a unique identifier for a function, although multiple functions can share the same simple name. For example, a `sum` function in class `A` and a `sum` function in class `B` share the same simple name but have different full signatures. In such cases, Code Indexer returns all matching functions, and it is the responsibility of the consuming component to distinguish between them using additional context, such as the file path.

6.9.3 LIBFDP

LIBFDP is a codec provider and encoder library for well-known FuzzedDataProvider (FDP) implementations, supporting both libFuzzer and Jazzer. It enables deterministic encoding of fuzzing inputs for cross-language and cross-platform fuzzing workflows, and provides both Rust and Python bindings for integration.

The challenge that LIBFDP addresses is to fill the gap between the complexity of byte processing in the behind scenes of FDP APIs and the weakness of mathematical processing in LLMs. Though some implementations between the two FDPs differs, one common point is that they both have a complex backend which requires LLMs to have a deep understanding of byte level processings to mimic the behavior, and this is not definitely a strong point of LLMs. Catching errors from LLMs is also one of the interests of LIBFDP. LIBFDP can provide semantic errors related to FDP usages which can be used to guide LLMs. This allows to generate inputs that are more likely to be accepted by the target harness, thus supporting the effectiveness of fuzzers using FDP as inputs. For example, given that LLM provided encoding sequence like in Figure 42, there is a semantic error on line 5 because it requests to produce a floating point value after consuming all remaining bytes on line 4. When the code in Figure 42 gets executed, LIBFDP may raise the error that can be utilized for providing feedback to LLMs to correct the sequence.

Internal library structure.

- `libfdp`: The core Rust library implementing encoder logic for LLVM and Jazzer FDPs.
- `pyfdp`: Python FFI bindings, exposing encoder APIs for Python-based fuzzing or harness scripting.
- `fdp-reference`: Reference C++ implementations and headers for testing implementation correctness and compatibility with LLVM and Jazzer FDPs.

FDP fuzzer example

```
int LLVMFuzzerTestOneInput(FuzzedDataProvider *fdp) {
  auto a = fdp.ConsumeIntegralInRange<uint16_t>(0, 127);
  auto b = fdp.ConsumeIntegral<uint8_t>();
  auto c = fdp.ConsumeBool();
  auto d = fdp.ConsumeFloatingPoint<double>();
  auto e = fdp.ConsumeRemainingBytesAsString();
  return func(a, b, c, d, e);
}
```

FDP encoder example

```
import libfdp

enc = libfdp.LlvmFdpEncoder()
enc.produce_unsigned_short_in_range(1, 0, 127)
enc.produce_byte(128)
enc.produce_bool(True)
enc.produce_double(1, 0, 127)
enc.produce_remaining_bytes_as_string(b"abcd")
print(enc.finalize())
```

Figure 43: FDP fuzzer and its corresponding encoder example with given values

Supported FDP Encoders.

- `LlvmFdpEncoder`: For FuzzedDataProvider in libfuzzer (targets written in C/C++)
- `JazzerFdpEncoder`: For FuzzedDataProvider in Jazzer (targets written in Java)

Each encoder exposes a set of producer functions that correspond one-to-one with the consumer functions in the target harness. Figure 43 shows the example of a fuzzing harness and the corresponding encode with LIBFDP. To produce the bytes consumed by the functions in the harness, the FDP encoder calls the corresponding encoding functions (*e.g.*,`produce_unsigned_short_in_range` for `ConsumeIntegralInRange<uint`). As as result, the encoder will produce `1`, `128`, `true`, `1`, `"abcd"`, respectively for variable `a`, `b`, `c`, `d`, `e` in the fuzzing code.

Reference Implementations. LIBFDP includes reference C++ code for both LLVM and Jazzer FDPs, ensuring compatibility and correctness. The encoder logic is carefully designed to match the semantics of the original consumer APIs, but some caveats apply (see below).

Caveats and Limitations. LIBFDP was used by Testlang-based Generation/Mutation module and MLLA, which generates python scripts for mutating and generating inputs. Concolic execution also utilizes `fdp-reference` for modeling FDP related functions. However, LIBFDP has some caveats and limitations.

- *Encoding is not always invertible*: Due to information loss in some consumer APIs (e.g., string and char consumption), the encoded blob may not exactly match the original target input.
- *Floating point handling*: Floating point encoding may yield different results depending on platform or build options.
- *Floating point quantization*: Encoding arbitrary floating point numbers is not possible due to the nature of FDP implementations. LIBFDP can only encode exact target values when the value is producible by the corresponding consumer API.
- *Unchecked APIs*: For cases where strict checks would reject legal sequences (e.g., due to indeterminism in FDP), LIBFDP provides '`_unchecked`' variants of encoder functions.

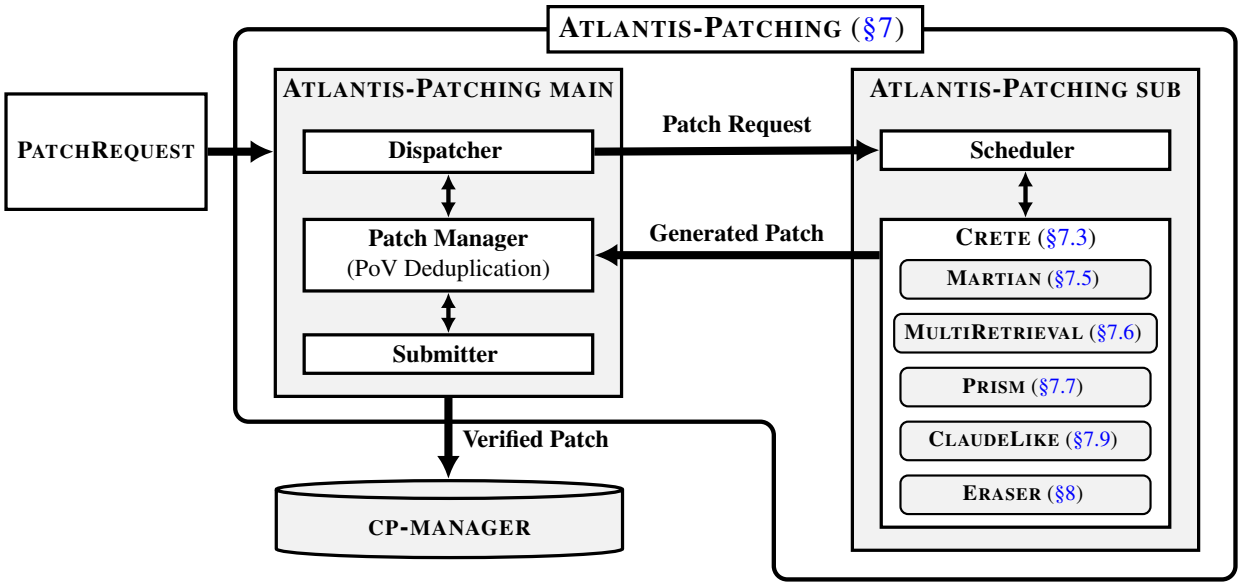


Figure 44: The overall architecture of ATLANTIS-Patching.

7 ATLANTIS-Patching

7.1 Overview

ATLANTIS-Patching is a subsystem within the ATLANTIS framework designed to automatically generate and deliver security patches. It leverages proof-of-vulnerability (PoV) and repository-level information to construct patches, and is implemented as a web server that exposes a set of APIs for handling patch requests and submitting validated patches to CP-MANAGER.

7.1.1 Workflow

The overall architecture of ATLANTIS-Patching is shown in Figure 44. When other subsystems detect a vulnerability, this information is delivered to the main node, which we referred to as `PatchRequest`. `PatchRequest` includes the project name, sanitizer type, harness name, and crashing input, which are essential for reproducing the vulnerability. Then, the main node first deduplicates the request as CP-MANAGER failed to duplicate the PoV properly. Thus, the main node uses previous patches for further deduplication. If the request is unique, it is forwarded to sub nodes, and the scheduler distributes the `PatchRequest` to multiple patching agents, each of which generates candidate patches. These patches are returned to the main node, where they are validated, deduplicated, and finally submitted to CP-MANAGER.

7.1.2 Key Ideas

In this section, we highlight the key ideas behind the design of ATLANTIS-Patching.

Ensemble Agents. One of the key ideas of ATLANTIS-Patching is the use of ensemble agents, which combine multiple patching methods through ensembling. Ensembling [34] is a well-established approach in machine learning that integrates diverse models to improve overall performance, and it is particularly effective in automatic patch generation. This effectiveness arises from the fact that generated patches can be validated through post-generation checks, such as compilation success, PoV mitigation, and functional testing. As these checks are incomplete and cannot guarantee correctness — since even validated patches may still

be incorrect — we refer to such patches as *plausible patches* [38]. Thus, in AIxCC, manual verification is further applied to address this limitation. Nonetheless, this partial verifiability of patches makes ensembling highly beneficial, as leveraging multiple agents increases the likelihood of producing a valid patch.

Ensembling provides two primary benefits: performance improvement and reliability. First, in terms of performance, ensembling enhances the system by integrating the strengths of each agent. Our observations indicate that LLMs do not operate in a principled or predictable manner. Our internal experiments further revealed that no single agent consistently outperformed all others across workloads; rather, the best-performing agent varied depending on the task. To address this, ATLANTIS-Patching employs an ensemble strategy, running diverse agents in parallel so that success is achieved if any one of them generates a valid patch.

Second, in terms of reliability, ensembling enables the system to remain robust across diverse workloads, which is crtical in the AIxCC setting. In AIxCC, our system should work autonomously across a wide range of projects and languages, without human intervention. Unfortunately, it is extremely difficult to construct such a reliable system as individual components often fail under specific conditions. For instance, we found that certain projects may not be compiled with a certain compiler (e.g., `clang`) or with certain flags (e.g., `-g`). Moreover, there can be projects that dynamically generate code at build time or even merge multiple code into a single file (e.g., `sqlite3`), which breaks a certain static analysis. Thus, a system that depends on a single method is vulnerable to such failures. Ensembling can mitigate this risk by distributing reliance across multiple implementations, analogous to N-versioning [7]. Consequently, even if one implementation fails, the overall system can continue to operate reliably.

Universal Framework for Agent Development. To implement ensembling effectively, it is essential to provide a framework that enables efficient development and integration of diverse patching agents. For this purpose, we introduce CRETE, a universal framework that unifies core functionalities required by agents. Although individual agents are designed differently, they share common operations such as building and PoV testing. In addition, several functionalities — such as file handling, code parsing, and patch generation — can be shared among agents. While individual developers could independently implement these capabilities, it may duplicate effort and reduce overall efficiency. By offering CRETE as a shared framework, we ensure that common functionalities are reusable and accessible across agents. This allows developers to focus on designing core algorithms and specialized strategies, thereby accelerating the development process and improving the overall quality of the system.

Two-level Policy Enforcement. Since agents are developed independently, some may behave abnormally, potentially undermining the reliability of the overall system. To mitigate this risk, patches must adhere to a set of mandatory policies. Specifically, a patch must first be plausible (i.e., a patch should be compilable and prevents PoV). Moreover, it must not modify harness files, and the changes must be restricted to source code files such as C or Java. Furthermore, patches should aim to be correct rather than merely plausible, since all submissions are manually validated after the competition. In practice, however, agents often generate plausible but incorrect patches, for instance, by removing functionality entirely or suppressing errors through a large `try-catch` block. While some of these policies can be enforced through rule-based checks, others are more difficult to validate in such a manner.

To address this challenge, ATLANTIS-Patching employs a *two-level policy enforcement mechanism*. The first level is agent-level. We explicitly communicated the required policies to all agent developers, who incorporated them into their prompts. This strategy strengthens patch generation by providing feedback and helps enforce policies that are not easily rule-based. However, it remains prone to developer error and to the inherent unpredictability of LLM-generated outputs. Thus, we implement a second level of verification at the system level. Once a patch is generated, the main node of ATLANTIS-Patching applies rule-based checks to validate the patch. Particularly, it revalidates plausibility of a patch by recompilation and PoV testing. Additionally, it ensures that it modifies only source code files and not harness files. Through this two-level verification process, the system compensates for agent-level errors and ensures robustness across the entire

patching workflow.

7.2 Node Architecture

The design of ATLANTIS-Patching requires an execution environment that supports parallelized builds and patch validation. A primary obstacle in adopting the OSS-Fuzz framework [22] within the AIxCC competition setting lies in its build process. The original OSS-Fuzz build scripts are not designed for parallel execution, as build artifacts are stored on mounted volumes, which complicates process isolation without modifying the build scripts. Unfortunately, it is risky to modify these scripts, as we cannot guarantee that the modified scripts will work correctly across all projects.

To overcome this limitation, we deployed multiple Azure nodes, each operating within its own isolated environment and equipped with dedicated hardware resources, such as network bandwidth. This setup enables fully parallelized execution of multiple agents, allowing OSS-Fuzz builds and reproduction tasks to run concurrently. Particularly, we provisioned one node for the ATLANTIS-Patching main node and four nodes for the sub nodes, each equipped with 32 vCPUs and 128 GB RAM. Furthermore, we assigned agents to nodes according to their underlying LLMs to avoid triggering excessive LLM rate limits. For example, we deployed MULTIRETRIEVAL with claude-3.7-sonnet and CLAUDELIKE at the same node, as both agents utilize Claude LLMs. Meanwhile, we deployed PRISM and AIDER at a separate node, as both agents rely on OpenAI LLMs. This strategic allocation of agents to nodes helps mitigate the risk of rate limiting and ensures smooth operation across the system.

7.2.1 Main node

The ATLANTIS-Patching main node coordinates the entire patching workflow. Given each incoming request, the main node processes it through three stages: de-duplication, dispatch, and submission.

Deduplication. When a new request arrives, the main node first performs de-duplication. This step is necessary because the de-duplication conducted by CP-MANAGER is inherently incomplete. The CP-MANAGER relies on organizer-provided methods and internal heuristics to identify duplicate vulnerabilities. However, as vulnerability de-duplication is a fundamentally hard problem [12, 13, 27], it is possible that some duplicates are not detected via these methods.

To mitigate this limitation, we employ a de-duplication strategy using patches. Specifically, ATLANTIS-Patching checks whether previously generated patches can block the newly submitted PoV. If an earlier patch successfully mitigates the new PoV, this implies that both PoVs originate from the same underlying vulnerability. The new PoV is therefore classified as a duplicate, and no new patch is generated. Through this mechanism, ATLANTIS-Patching efficiently eliminates redundant work while ensuring that no unique vulnerabilities are overlooked.

Dispatch. If a PoV is determined to be unique, the main node dispatches it to the sub nodes for patch generation. The dispatch mechanism is implemented in a multi-threaded manner, allowing the main node to continue receiving new PoVs while awaiting responses from sub nodes.

Among the patches generated by sub nodes, ATLANTIS-Patching selects the first valid patch returned. While more sophisticated selection policies are possible, we chose this simple approach for two reasons. First, as aforementioned, our system should be robust and reliable. If we wait for multiple nodes to respond, we should handle risks that some nodes may not respond indefinitely due to failures or network issues. If we failed to handle such cases, the entire system may become unresponsive. Thus, we prioritize responsiveness and simplicity in our design. Second, in the AIxCC competition, the scoring is influenced by a time-based multiplier, making early patch submission advantageous. Therefore, we just select the earliest patch and submit it upstream, instead of using more complex policies (e.g., LLM-as-a-judge [24] or self-consistency [42]).

Submission. Once a valid patch is identified, the main node proceeds with the submission stage. Before sending the patch upstream to CP-MANAGER, it performs two types of checks: de-duplication and policy enforcement.

First, the main node performs an additional round of de-duplication. Since ATLANTIS-Patching operates in parallel across multiple PoVs, it is possible that the current PoV is already resolved by a patch generated for another PoV. To handle such race conditions, the main node re-checks whether the patch is redundant.

Second, the main node enforces policies through a set of pre-defined rules to ensure the integrity of the submission. Specifically, it verifies that the patch compiles successfully, blocks the PoV, and modifies only valid source code files. Even though we instructed agents to adhere to these policies, it is possible that some agents may not fully comply due to developer error or the inherent unpredictability of LLMs. To this end, we leverage an ML-based file type detection tool, which is provided by the organizer, to confirm that the modified file is indeed source code (e.g., C or Java) rather than auxiliary artifacts. Additionally, we applied heuristic checks to ensure that harness files are not altered. Then, we submit the patch to CP-MANAGER only if it passes all these checks.

Sub node. Each sub node holds a set of agents and executes them iteratively to generate patches for incoming requests. As described earlier, we configured the agent set of each sub node for diversity and to avoid LLM rate limits. Upon receiving a patch request, the scheduler within a sub node iterates through its set of agents, invoking each agent to attempt patch generation. If a patch is successfully generated, it is returned to the main node, where it validates, enforces policies, and finally submits the patch.

7.3 CRETE: A Unified Framework for Patch Generation Framework

In this section, we describe CRETE, a framework for developing automated patching agents, and we briefly introduce the agents built on top of it. After this section, we describe each agent in detail in §7.5–§7.10.

7.3.1 Overview

CRETE is a framework designed to provide a standardized environment and essential utilities for developing automated patching agents. It serves as a foundational layer that abstracts repository management, build configuration, and patch validation, thereby enabling heterogeneous agents to operate on a common platform. The framework initializes target repositories for specific vulnerabilities, delivers them to designated agents, and subsequently validates the resulting patches through an integrated evaluation process.

The primary objective of CRETE is to facilitate the development of diverse patching agents while preventing environmental conflicts and reducing redundant implementation. By offering a stable and unified interface, the framework minimizes the risk of errors and improves development efficiency, allowing developers to focus on agent-specific logic. This design has enabled the concurrent operation of multiple, distinct agents within our system.

7.4 Core Components

CRETE comprises several core components that collectively support the patching workflow. These components include:

- **Environment.** Provides managed interfaces for interacting with challenge projects, provisioning distinct build environments, and leveraging caching mechanisms to avoid redundant compilations. This accelerates both testing and development cycles.

- **Evaluator.** Integrates with OSS-Fuzz to rigorously assess generated patches. It verifies compilation, reproduction, and functional correctness to ensure that patches mitigate vulnerabilities without introducing regressions.
- **Fault Localizer.** Analyzes crash logs and SARIF reports to identify likely fault locations, offering a baseline that agents can refine for precise fault localization and patch synthesis.
- **Code Retriever.** Supplies an API for code navigation, enabling agents to query program elements such as functions or variables. It leverages tools like `tree-sitter` [41] and `ctags` [11] to support efficient source-code analysis.
- **Common Analyzers.** Provides reusable analyzers for crash logs, call traces, and commits, allowing agents to extract actionable insights.

In the following, we briefly describe several interesting features that CRETE provides.

Environment Pool. To optimize the build process, CRETE employs an *environment pool*. Unlike fuzzing, which generally requires a single build, patch generation involves repeated builds for testing and validation. The environment pool addresses this by maintaining pre-configured environments consisting of reusable artifacts such as compiled binaries and libraries, thereby avoiding redundant builds. CRETE supports multiple types of environments for fault tolerance. For instance, some environments use `ccache` to reduce build time, while others exclude `ccache` to handle cases where it is unreliable. When an agent requests an environment, CRETE allocates one from the pool according to the agent's preferences and current availability. The same environments are also used for static analyses, such as call-trace or debug-symbol extraction.

Cache Everywhere. To improve efficiency, CRETE adopts a *cache-everywhere* strategy. It caches all results that are computationally expensive yet deterministic, such as crash reproduction, patch validation, and selected static analyses. This mechanism eliminates redundant computations, allowing agents to reuse previously obtained results transparently. For example, a developer can invoke the patch validation API multiple times with the same patch without incurring additional overhead. By automating caching, CRETE reduces performance concerns and allows developers to focus on the core logic of their agents.

Special Handling for Crash Logs. CRETE also includes specialized handling for certain types of crash logs. For instance, stack overflow crashes in Java often produce repetitive stack frames, which hinder effective analysis. To mitigate this, CRETE preprocesses stack traces to remove redundant frames. Similarly, timeout bugs in Java typically do not generate stack traces, complicating the initial analysis. To address this, CRETE leverages `jstack` [35] to capture stack traces when a timeout occurs. In particular, if the crash is a timeout for Java, CRETE reproduces the crash while periodically capturing stack traces 30 times using `jstack`. Then, the collected stack traces are provided to agents for further analysis. As timeout bugs are often caused by infinite loops, the stack traces can help agents identify the loop location. By doing so, CRETE can supply valuable diagnostic information independent of agent design and eliminates the need for redundant implementations.

7.4.1 Agents built on CRETE

On top of CRETE, we implemented multiple automated patching agents, each with distinct specializations and architectural designs (see Table 13). In the AIxCC competition setting, the primary goal is to produce at least one correct patch among many attempts. To maximize this likelihood, ATLANTIS-Patching exploits the diversity of agents via emsembling, leveraging differences in their underlying LLMs, reasoning strategies, and tool integrations. This diversity enables agents to explore different portions of the codebase and generate complementary patches, increasing the probability of finding a correct fix.

In addition to our in-house agents, we incorporated open-source agents such as AIDER [21] and SWE-AGENT [45]. Leveraging these external tools allows us to build on established implementations

Agent	Architecture	Motivation	Used models
MARTIAN (§7.5)	Workflow	Introducing architectural diversity for robustness and to enable flexible integration of external tools via a ReAct-style design	o4-mini and claude-4-sonnet
MULTIRETRIEVAL (§7.6)	Agent	Autonomously exploring codebases and generate vulnerability patches by iteratively building context through interaction	claude-3.7-sonnet or o4-mini
PRISM (§7.7)	Multi-agent	Addressing context length limitations and prompt complexity in MULTIRETRIEVAL through a team-based multi-agent system.	o4-mini
VINCENT (§7.8)	Workflow	Incorporates project-specific properties from software verification to guide patching while maintaining generality	gemini-2.5-pro
CLAUDELIKE (§7.9)	Agent	Inspired by Claude Code, featuring advanced file editor tools and sub-agent delegation to manage context efficiently and streamline patching tasks	claude-3.7-sonnet
ERASER (§8)	Workflow	Aiming to be used for a custom model	Custom model

Table 13: Patching agents in ATLANTIS-Patching (except for open-source agents). Notably, an architecture in this table is just a high-level classification of the agents, and does not imply that the agents are implemented. For that, please refer to the individual agent descriptions.

that have been validated by the developer community. Their inclusion also enhances patch diversity, as they are trained or tuned with different datasets, prompting strategies, and reasoning heuristics. Detailed descriptions of AIDER and SWE-AGENT are provided in §7.10. In the following sections, we describe the agents that ATLANTIS-Patching employs, including those built on CRETE as well as open-source agents.

7.5 Agent: MARTIAN

MARTIAN agent is an AI agent that fixes security bugs with a fixed workflow that mimic the approach of human developers. The agent decomposes the patching process into two stages: Fault Localization and Patch Generation. Each stage is handled by a ReAct-based agent, equipped with specialized tools. The core design philosophy behind MARTIAN agent is having a straightforward workflow with easily extendable external tools. With this design, it can not only ensure effective vulnerability analysis and patch generation, but also provide the flexiblity to adapt new tools.

7.5.1 Motivation

The motivation behind the development of the MARTIAN agent was twofold. First, we aimed to explore a different architectural approach compared to existing agents, such as the MULTIRETRIEVAL agent. By introducing architectural diversity, we intended to make our ATLANTIS-Patching framework more robust and adaptable across a wider range of problems. Second, we wanted to simplify the integration and testing of external tools. By leveraging a ReAct-style node with tool calls, the MARTIAN agent enables easy addition and removal of tools, providing the flexibility needed for rapid experimentation and adaptation.

7.5.2 System Architecture

The overall workflow of MARTIAN agent is shown in Figure 45. First, it generates a crash log from the given security bug. The crash log should include a sanitizer report or at least a stacktrace that clearly describe the bug. The fault localization module then analyzes the crash log by searching the codebase and find the root cause of the bug. As output, it identifies the buggy function and generate a plan to fix it. These outputs are passed to the patch generation module, which finally generates a patch for the identified buggy function. The generated patch is then evaluated, and if it fails validation, feedback is sent back and retries the process.

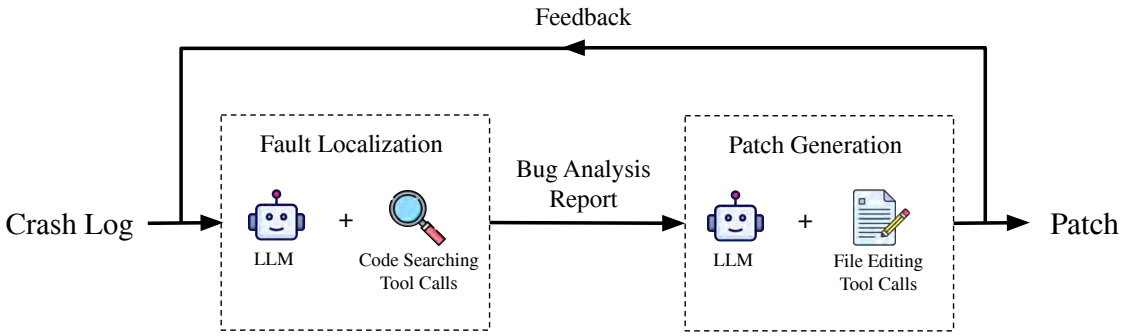


Figure 45: MARTIAN agent architecture

Fault Localization. MARTIAN's fault localization module is inspired by CODEROVER-S [46]. Given that CODEROVER-S generated 52.6% plausible patches on the ARVO dataset, we selected it as a baseline. Since the code search APIs of CODEROVER-S are closed-source, we developed our own language-agnostic APIs for code search. The fault localization module includes the following three APIs as tools:

- **`search_symbol(symbol_name, file_or_directory_path)`**: Given a symbol name, it finds a symbol definition in a given path and return the source code of the symbol definition.
- **`search_string(string, file_or_directory_path)`**: Given a string, it returns the file name and line number(s) where the string appears in the codebase.
- **`view_file(file_path, offset, limit)`**: Given a file name, offset, and limit, it returns the file content from the offset to the limit.

The code search APIs provide high-level, abstracted interface for robust and efficient code navigation. The `search_symbol` tool works similarly to "Go to Definition" in modern code editors, while `search_string` works similarly to "Ctrl+Shift+F" for full-text search. These tools serve as powerful primitives for the code exploration – even for the human developers. The `search_symbol` tool operates on abstract symbols without requiring the LLM to specify their types (e.g., function, variable, or class). This design keeps the interface simple, offloading the complexity of type handling to the implementation side of the tool. Moreover, if `search_symbol` fails, the LLM can fall back to the `search_string` tool to continue the analysis. These tool combinations enable flexible and resilent code search.

Patch Generation. Generating an appliable patch is a non-trival task, as it must adhere to a strict format to be correctly applied. To address this challenge, we separate the patch generation process from fault localization and have an independent ReAct agent with its own dedicated context. This separation allows the patch generation module to focus more effectively on its task by adapting the context.

The patch generation module employs a search/replace strategy at the function level, implemented via tool calls. This approach is inspired by Claude Code [4]; however, unlike Claude Code, which operates at the file level, MARTIAN operates at the function-level. By narrowing the scope of the patch generation, this method offers advantages in terms of context length. Additionally, as other agents operate at the file level, MARTIAN agent focuses on a narrower, more precise scope, contributing to the overall diversity of our patch system. The patch generation module includes the following three APIs:

- **`view_function(function_name)`**: Given a function name, it returns the source code of the function.
- **`edit_function(function_name, old_string, new_string)`**: Given a function name and search/replace code snippets, it edits the source code of the function.

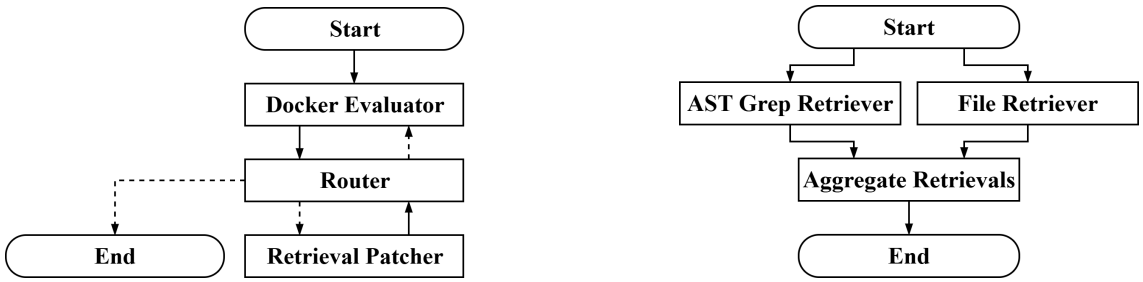


Figure 46: The architecture of the MULTIRETRIEVAL system, illustrating the interactions between the specialized nodes. The left figure shows the overall architecture, while the right figure focuses on the `CodeRetriever` that manages multi-source code retrieval operations which be used by the `SystemGuidedPatcher`.

- **`add_import_module(module_name, file_path)`**: Add import statement in the given file. This is only used for JAVA.

With these APIs, the patch generator is able to fix the function and finally we use `git diff` command to get the patch diff. We need `add_import_module` tool for JAVA because empirically there were few compilation error because of missing module imports.

7.6 Agent: MULTIRETRIEVAL

The MULTIRETRIEVAL represents an automated vulnerability patching system that combines multiple code retrieval strategies with iterative patch generation and validation. This system leverages state-machine based workflows, diverse retrieval backends, and verifiable evaluations to generate patches for software vulnerabilities. By integrating Abstract Syntax Tree (AST) augmented retrieval, grep-based retrieval, and direct file retrieval as tools, the agent achieves robust patch generation across C and Java codebases.

7.6.1 Motivation

The motivation behind MULTIRETRIEVAL is to create an agent that can effectively explore the codebase and generate patches for software vulnerabilities by iteratively interacting with the codebase and evaluation system. This approach allows the agent to build its own context incrementally, refining its understanding of the codebase and the vulnerability at hand. This was inspired by the agentic capabilities of LLMs, where we only give right tools and goals to the agent, and it can explore the codebase and generate patches autonomously.

7.6.2 System Architecture

The MULTIRETRIEVAL follows an iterative process where the agent queries the codebase to retrieve relevant context, generates patches based on the retrieved information to get a feedback from the evaluation system, and refines the patches iteratively until a valid patch is produced or the maximum number of evaluations is reached. Although the agent operates within a single context with multiple turns of interaction, it uses three specialized nodes to handle different aspects of the patching process which are the `SystemGuidedPatcher`, `DockerEvaluator`, and `CodeRetriever` illustrated in Figure 46.

SystemGuidedPatcher. The `SystemGuidedPatcher` is the main node that analyzes the current state and decides the next action based on the current patch status and retrieved context. It uses a state machine to manage the patching process, which includes actions such as `EVALUATE`, `ANALYZE_ISSUE`, and `RETRIEVE` based on the decision made by the LLM. The system prompt for the `SystemGuidedPatcher` is designed to contain all the necessary information to guide the LLM to interact with the codebase and the evaluation system effectively.

Algorithm 5: MultiRetrieval Agent Workflow for Iterative Patch Generation

```
Initialize PatchState with repository path
Set initial action to EVALUATE
Initialize n_evals ← 0, max_n_evals ← 10
while action ≠ DONE and n_evals < max_n_evals do
    if action == EVALUATE then
        Increment n_evals
        if status == INITIALIZED then
            Run proof-of-vulnerability test to get initial crash log
        else
            Run DockerEvaluator to assess current patch
            Update patch status based on evaluation results
        if status == PLAUSIBLE then
            Run internal tests
            if tests pass then
                action ← DONE
            else
                action ← ANALYZE_ISSUE
        else
            action ← ANALYZE_ISSUE
    else if action == ANALYZE_ISSUE then
        SystemGuidedPatcher analyzes the issue
        Generate exploration plan
        action ← RETRIEVE
    else if action == RETRIEVE then
        SystemGuidedPatcher analyzes current retrieval context and decides next step
        if needs more context then
            Execute retrieval queries
            Aggregate and format results
        else
            Generate patches
            action ← EVALUATE
return final diff
```

DockerEvaluator. The `DockerEvaluator` is responsible for validating the generated patches in isolated containerized environments. It runs the patch against the codebase and evaluates its effectiveness by checking whether the patch resolves the vulnerability and passes the internal test suite if available. The evaluation results are categorized into multiple states such as `PLAUSIBLE`, `UNCOMPILABLE`, and `VULNERABLE` to provide feedback to the `SystemGuidedPatcher` for further actions.

CodeRetriever. The `CodeRetriever` is a specialized node that manages multi-source code retrieval operations. It leverages various retrieval techniques, including AST-based retrieval, text-based retrieval, and direct file retrieval, to gather relevant code snippets and context for the patching process. By integrating these diverse retrieval methods, the agent can explore the codebase more effectively and retrieve the most relevant information for patch generation.

Multi-turn State Machine. The MULTIRETRIEVAL follows a multi-turn state machine workflow as illustrated in Algorithm 5. The agent starts with an initial crash log and decides the next action based on the current status. The system automatically returns the requested information whether it is the code snippets or the evaluation results as if the agent is interacting with a human developer.

7.6.3 Code Retrieval Strategy

The `CodeRetriever` implements a code retrieval strategy that combines three complementary approaches to work with a variety of codebases:

- **AST-based Retrieval:** Uses grep to locate possibly related code and Abstract Syntax Tree analysis augments the retrieval if available.
- **Text-based Retrieval:** Employs regex-capable text search for fast pattern matching and is used as a fallback when AST-based retreival is failed.
- **Direct File Retreival:** Retrieves complete files when specific context is needed.

The retrieval system processes LLM-generated queries by executing them in parallel across available retrievers. Results are aggregated with a predefined priority based filtering to ensure the most relevant snippets are returned to the agent. For example, AST-based retrieval is prioritized over text-based retreival where if returns the whole function definition instead of few surrounding lines when queried with a function name. This augmented retrieval strategy allows the agent to efficiently gather context from large codebases, improving the quality of generated patches.

7.7 Agent: PRISM

PRISM is a multi-agent system that employs a hierarchical team-based architecture with three specialized agent teams—Analysis, Patch, and Evaluation—collaborating to analyze vulnerabilities, generate patches, and validate their effectiveness.

7.7.1 Motivation

During the development of MULTIRETRIEVAL, we observed that in some cases, the agent could not generate a valid patch within the maximum context length. Also, the overall system prompt had multiple responsibilities, which made it difficult to manage its effectiveness. To address these issues, we designed PRISM as a multi-agent system that separates the responsibilities of each agent into specialized teams. This team-based architecture allows for more focused and efficient interactions, where the workflow naturally compacts the context by sending only the relevant information and reduces the cognitive load on individual agents.

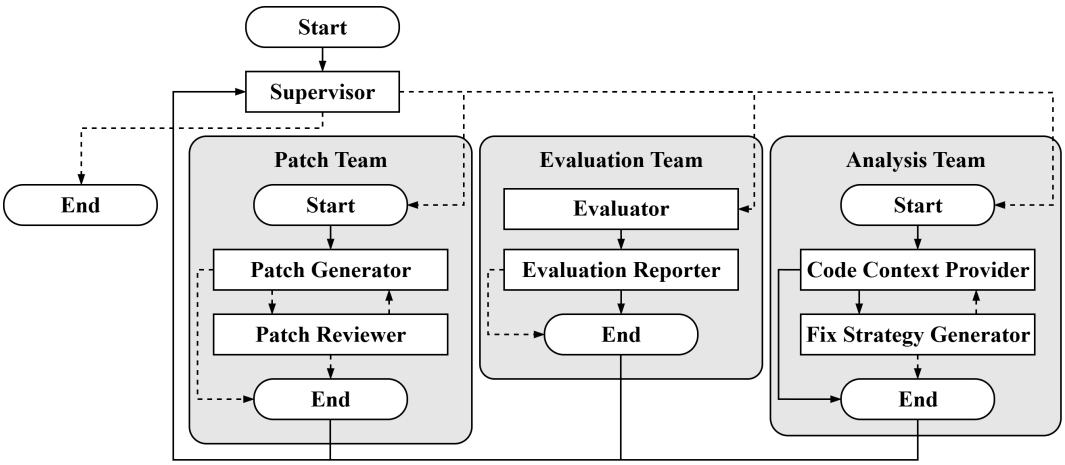


Figure 47: The architecture of the PRISM system, illustrating the interactions between the specialized agent teams.

7.7.2 System Architecture

Three specialized teams within PRISM are coordinated by a supervisor which executes a iterative deterministic workflow of analysis, patch generation, and evaluation processes as illustrated in Figure 47. The system follows a deterministic workflow pattern which iterates through the teams until a valid patch is generated or the maximum number of iterations is reached. It iterates through evaluation, analysis, and patch generation steps, with each team handing off cumulative results to the next team in the cycle.

7.7.3 Team Implementations

Evaluation Team. The Evaluation Team generates an evaluation report to provide a general description of the crash log and the result of the patch evaluation for next iteration if needed. It consists of two agents: `Evaluator` and `EvaluationReporter`. The `Evaluator` executes patches against the codebase, running proof-of-vulnerability tests and internal test suites to determine patch effectiveness. It handles various failure modes, from compilation errors to internal test failures, and processes logs with appropriate filtering and formatting to extract relevant information from verbose output, handling language-specific error formats and filtering noise to increase the token efficiency and overall accuracy by reducing distractors. The `EvaluationReporter` generates comprehensive reports for failed patches, analyzing failure patterns and providing actionable feedback. These reports guide subsequent analysis and patch generation cycles, enabling the system to learn from unsuccessful attempts and progressively refine its approach.

Analysis Team. The Analysis Team is responsible for building the relevant code context while finding the root cause of the current issue and suggesting a comprehensive fix strategy structured as an analysis report. It consists of two agents: `CodeContextProvider` and `FixStrategyGenerator`. The `CodeContextProvider` explores the codebase to gather relevant code snippets and build a comprehensive understanding of the current context. This agent employs both top-down and bottom-up exploration strategies, using AST augmented grep-based searches and file retrieval to construct an analysis notebook incrementally where it will serve as the code context to the `FixStrategyGenerator`. The `FixStrategyGenerator` takes the collected code snippets and analysis to generate a detailed fixing strategy that can guide the Patch Team. The agent can iterate multiple times to refine its strategies by retrieving additional context.

Patch Team. The Patch Team transforms analysis and evaluation reports into a concrete patch formatted as an unified diff. It consists of two agents: `PatchGenerator` and `PatchReviewer`. The `PatchGenerator` creates patches based on the fix strategy and evaluation results described in the reports, maintaining awareness

of previous patch attempts to avoid repetitions or ineffective changes. The `PatchReviewer` validates the generated patches for correctness and adherence to predefined coding standards, ensuring that they are syntactically correct and semantically meaningful. This review process helps catch potential issues before evaluation, improving the efficiency of the overall system by reducing the number of obviously flawed patches that reach the evaluation stage.

7.7.4 State Management

PRISM employs a hierarchical state management system that maintains context across team boundaries while allowing for team-specific extensions. The maintained context mainly includes code snippets, analysis report, evaluation report, and patch diff. These are shared across the teams and condensed, replaced, or updated throughout the iterative process giving the system a coherent understanding of the current state and maintaining continuity. The team-specific state extensions allow each team to focus on its specific tasks with additional context. This hierarchical state management enables efficient communication and collaboration between teams, ensuring that only the relevant information is passed along. For example, the Analysis Team contructs an code analysis notebook that contains multiple code snippets and analysis for each code snippet, which is only used to generate a fix strategy. The Patch Team, on the other hand, only needs the final fix strategy and the code snippets where it does not need to access the entire analysis notebook which increases the operating efficiency of the system. This state management system combined with multiple layers of error handling and feedback loops allows PRISM to robustly navigate through the codebase, iteratively refining its understanding and improving the quality of generated patches.

7.8 Agent: VINCENT

VINCENT is an agent that includes a property analysis step in its pipeline, aiming to generate more context-aware patches.

7.8.1 Motivation

In our experiments of agents in ATLANTIS-Patching, we identified a challenging case that LLMs fail to generate a valid patch. In `nginx`, there is a function named `ngx_http_process_prefer` that handles `Prefer` headers in HTTP requests. As shown in Figure 48, the function checks whether the `Prefer` header has already been found (line 8). If the pointer is not `NULL`, it frees the existing pointer and returns `NGX_OK`. However, a problem arises when the HTTP request contains multiple `Prefer` headers, as illustrated in Figure 49. The malformed HTTP request contains three `Prefer` headers, which is not an allowed situation for `nginx`. Due to the first `Prefer` header, memory is allocated to `r->headers_in.prefer`. When the program tries to handle the second header, it enters the `if`-block (line 8 in Figure 48) since `r->headers_in.prefer` already has a value. However, the problematic pointer is not properly nullified, which fails to prevent future access to the dangling pointer. Finally, when the third header is processed, the `if`-block is entered again, triggering the double-free of `r->headers_in.prefer`.

To fix this bug, the patch needs to (i) nullify the problematic `r->headers_in.prefer` pointer, (ii) finalize the current request with `NGX_HTTP_BAD_REQUEST`, and (iii) return `NGX_ERROR` instead of `NGX_OK`. The intended patch is presented in Figure 50. However, our agents kept failing to generate a working patch, leading to incomplete patches like Figure 51. The main reason for failure was that the generated patches only nullified the problematic pointer, missing the proper cleanup process for an invalid HTTP request (*i.e.,* steps (ii) and (iii)). In other words, LLMs only focused on fixing the superficial symptom presented in the ASAN report.

To deal with such cases, we built an LLM agent named VINCENT. The core idea is to enable LLMs to consider project-specific aspects without losing generality. To achieve this, VINCENT borrows the concept of

```
static ngx_int_t
ngx_http_process_prefer(ngx_http_request_t *r,
                        ngx_table_elt_t *h,
                        ngx_uint_t offset)
{
    ngx_table_elt_t *p;

    if (r->headers_in.prefer) {
        ngx_log_error(/* omitted */);
        ngx_free(r->headers_in.prefer); // double-free
        return NGX_OK;
    }

    /* omitted */

    return NGX_OK;
}
```

Figure 48: The part of the `ngx_http_process_prefer` function. This function contains the double-free bug (line 10) due to improper handling of `Prefer` headers in the invalid HTTP request.

```
GET / HTTP/1.1
Host:  localhost
Prefer:  FirstPrefer
Prefer:  SecPrefer
Prefer:  ThirdPrefer
Accept:  */*
```

Figure 49: The PoC for `nginx` double-free vulnerability. This HTTP request contains three prefer headers, leading to entering the `if`-block in Figure 48 (line 8) multiple times.

properties [9], which refers to a condition or behavior that a program must satisfy, from the field of software verification and engineering.

7.8.2 Agent Workflow

At a high-level, VINCENT consists of three analysis steps, as presented in Figure 52. During the workflow, each analysis step can retrieve code information from the code retriever whenever the LLM wants. For the first step, VINCENT conducts root cause analysis given a sanitizer report and PoC. Then, VINCENT extracts the program properties related to the given bug. Finally, based on the root cause and property analysis report, VINCENT generates a patch. As with other agents in ATLANTIS-Patching, VINCENT repeats patch generation until it produces a plausible patch.

7.8.3 Code Retriever

During the analysis step, VINCENT allows the LLM to request any code information whenever needed. Instead of returning source files as a whole, VINCENT combines `ctags` utility [11] and `tree-sitter` [41] library to serve the code snippets essential to LLM's analysis. Specifically, given a target symbol name such as a function or variable, VINCENT first locates where the requested code snippet exists using the `ctags`. Then, it extracts the actual code information, including the surrounding code context (*e.g.,* the full implementation of a function, struct, enum, etc) using `tree-sitter`.

Beyond this basic strategy, we observed occaisional failures of `tree-sitter`'s parser when dealing with complex codebases. For example, if the target source code contains complex compile directives or macros, `tree-sitter` often fails to parse the intended snippets properly. To handle such potential errors from external libraries, VINCENT allows LLMs to request additional code lines when the initial query result is not sufficient.

```
--- a/src/http/ngx_http_request.c
+++ b/src/http/ngx_http_request.c
          ngx_free(r->headers_in.prefer);
-         return NGX_OK;
+         r->headers_in.prefer = NULL;
+         ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
+         return NGX_ERROR;
      }
```

Figure 50: The intended patch for `nginx`'s double-free bug. The patch requires the function to (i) free the problematic pointer, (ii) finalize the request with `NGX_HTTP_BAD_REQUEST`, and (iii) return `NGX_ERROR` to prevent further processing of the given request.

```
--- a/src/http/ngx_http_request.c
+++ b/src/http/ngx_http_request.c
         ngx_free(r->headers_in.prefer);
+        r->headers_in.prefer = NULL;
         return NGX_OK;
     }
```

Figure 51: The typical patch generated by LLMs. It nullifies the problematic pointer, but fails to consider nginx-specific cleanup process for the invalid HTTP request.

This two-step strategy effectively mitigates both parsing failures caused by the `tree-sitter` and excessive token usage caused by naive line-based file exploration.

To further assist the LLM in understanding the codebase, VINCENT supports both symbol-reference search and code-embedding-based [32] search features, providing LLMs with richier context for reasoning about the codebase. LLMs can utilize this information to infer program behaviors, function usages, and essential code patterns needed to know for patching. For example, to correctly understand the behavior of a certain function `foo`, it may be necessary to inspect locations that call `foo`, or functions with similar structures to `foo`. For the symbol reference search, VINCENT adopts a string-based strategy, which retrieves code snippets where the requested symbol name appears. For the code-embedding-based search, VINCENT converts functions in the codebase and stores the results in the database. When a specific function symbol is requested, VINCENT provides the code snippets similar to the given function symbol. In the current implementation, VINCENT leverages OpenAI's `text-embedding-3-large` model to embed each code snippet. To calculate the similarity, VINCENT adopts cosine similarity, which is a widely-used metric for this purpose. However, naively embedding the entire codebase may consume excessive computing resources and significantly increase LLM costs. To mitigate the issue, VINCENT limits embedding calcualations to the source files that contain symbols previously requested for code retrieval.

7.8.4 Root Cause Analysis

When VINCENT accepts the sanitizer report and PoC, it performs an intial root cause analysis. Essentially, VINCENT lets the LLM explore the codebase on its own as much as needed. This strategy allows VINCENT to fully utilize the LLM's reasoning capabilities, enabling it to infer the fault location—even when that location is not explicitly presented in the sanitizer report's callstack. In addition to the sanitizer report, VINCENT provides the PoC bytes to the LLM using the `xxd` utility to further assist LLM's reasoning process. To prevent excessive token usage, the PoC bytes are provided only if their size is less than 25,000 bytes, a value determined heuristically.

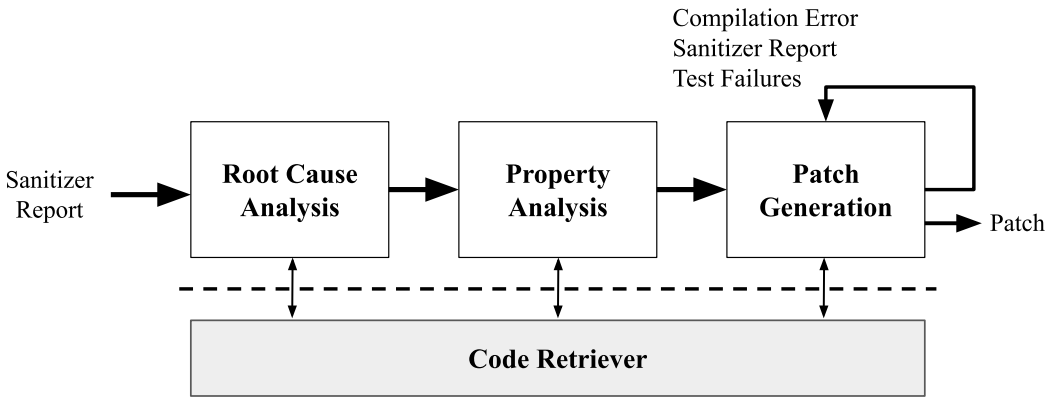


Figure 52: The overall workflow of VINCENT agent. Given a sanitizer report and PoC, it conducts root cause analysis and property analysis. Based on the analysis result, it generate security patch in an iterative approach.

7.8.5 Property Analysis

As mentioned before, VINCENT extracts program's properties from the given codebase. Note that VINCENT utilizes properties represented in natural language, unlike their typical usage in the field of software verification. For example, the following is a partial list of properties generated by the LLM regarding the `nginx` double-free bug.

1. **Memory Safety**: For any header field in `ngx_http_headers_in_t`, the pointer must either be NULL or point to a valid ngx_table_elt_t structure.
2. **Header Processing Consistency**: All single-value HTTP headers (like Host, From, Content-Length) must maintain exactly one instance throughout the request processing lifecycle.
3. **Error Response Consistency**: When encountering invalid headers, the system must either: (1) Return `NGX_ERROR` and call `ngx_http_finalize_request` with `NGX_HTTP_BAD_REQUEST` (2) Log the issue and return `NGX_OK` But never both or neither.

To be specific, the "Error Response Consistency" is one that LLMs failed to consider in patches like the one shown in Figure 51. In the patch step prompt, VINCENT instructs LLMs to consider such properties to generate more appropriate for the given program context.

Based on this concept of properties, VINCENT performs this analysis step as follows. Similar to the previous root cause analysis, VINCENT allows the LLM to explore the codebase on its own. When the LLM decides that the collected information is sufficient, it outputs the list of properties that must be considered regarding the future patch generation. Regarding prompt engineering, it was found that LLMs could infer properties without any specialized tools or a detailed prompt that explains what a property is. This is likely because the LLM already has knowledge of program properties, allowing for the use of relatively simple prompts.

7.8.6 Patch Generation

In this step, VINCENT requests the LLM to generate a patch considering the previous analysis results. Similar to other agents in ATLANTIS-Patching, VINCENT adopts a feedback-based strategy for patch generation. In particular, whenever the LLM generates a patch, VINCENT evaluates the patch by using the default OSS-Fuzz-based evaluator of ATLANTIS-Patching. If the evaluation fails (*e.g.,* compilation error, another crash, or internal test failures), VINCENT provides the failure information to the LLM and retries the patch

generation. As in the previous analysis steps, the LLM is allowed to request additional code information during the patch generation.

To translate LLM's patches into actual diffs, VINCENT employs a line-replace strategy. To elaborate, VINCENT instructs the LLM to submit a patch as a tuple of three items: (i) the target source filename, (ii) the lines to replace, and (iii) code content to replace the specified lines.

7.9 Agent: CLAUDELIKE

CLAUDELIKE is an agent that performs patching by using the ReAct agent CLAUDELIKE coder, inspired by Claude Code [4].

7.9.1 Motivation

During the preparation phase for AIxCC, Anthropic released Claude Code, one of the most powerful LLM-based tools for programming assistance. Preliminary experiments demonstrated that Claude Code achieved competitive performance in understanding project structures and performing code modifications, making it a promising reference for patching tasks.

Claude Code's strength lies in two design choices: (i) well-engineered file editor tools that allow the agent to efficiently analyze and modify project files, and (ii) the use of sub-agents to offload tasks requiring frequent tool invocations, thereby reducing irrelevant context retained by the main agent. This design enables the main agent to focus on essential reasoning for patch generation.

Motivated by these findings, we designed CLAUDELIKE to incorporate the strengths of Claude Code into the patching domain. Specifically, CLAUDELIKE integrates a diverse set of file editor tools and supports the spawning of sub-agents to handle specialized tasks. This design not only improves efficiency in patch generation but also enhances scalability by mitigating the context overhead that often limits LLM agents.

7.9.2 Agent Workflow

The CLAUDELIKE agent operates through an iterative workflow inspired by Claude Code. First, (i) the agent constructs a prompt by incorporating insights extracted from the PoC and SARIF reports, and forwards this prompt to the coder module. Next, (ii) the coder generates a candidate patch in the form of a `diff` using the ReAct model. Once the patch is produced, (iii) the agent evaluates its validity by rebuilding the project, executing the functional tests, and running the PoV to confirm whether the vulnerability is mitigated. Finally, (iv) if the patch fails any of these checks, feedback is provided back to the coder, which then attempts to synthesize a new patch. This iterative process continues until a plausible and valid patch is produced or a predefined termination condition is reached.

7.9.3 Coder

The workflow of CLAUDELIKE is largely similar to that of other simple agents employing a feedback loop. The key difference, however, lies in the internal mechanism of the CLAUDELIKE coder. As shown in Figure 53, the overall structure of the coder is organized around two ReAct models: a main agent and a sub-agent. Each of these agents is able to interact with the file system through a common set of tools, enabling them to analyze and modify project files as needed. In addition, the `AgentTool` provides the capability for the main agent to spawn a sub-agent dynamically, delegating specific tasks to it when necessary. This design separates responsibilities between the two agents and improves the efficiency of patch generation.

File Editor Tools. The file editor tools provide coders with flexible capabilities to navigate project files, inspect their contents, and apply necessary modifications, thereby supporting efficient patch generation. The following describes each tool in detail:

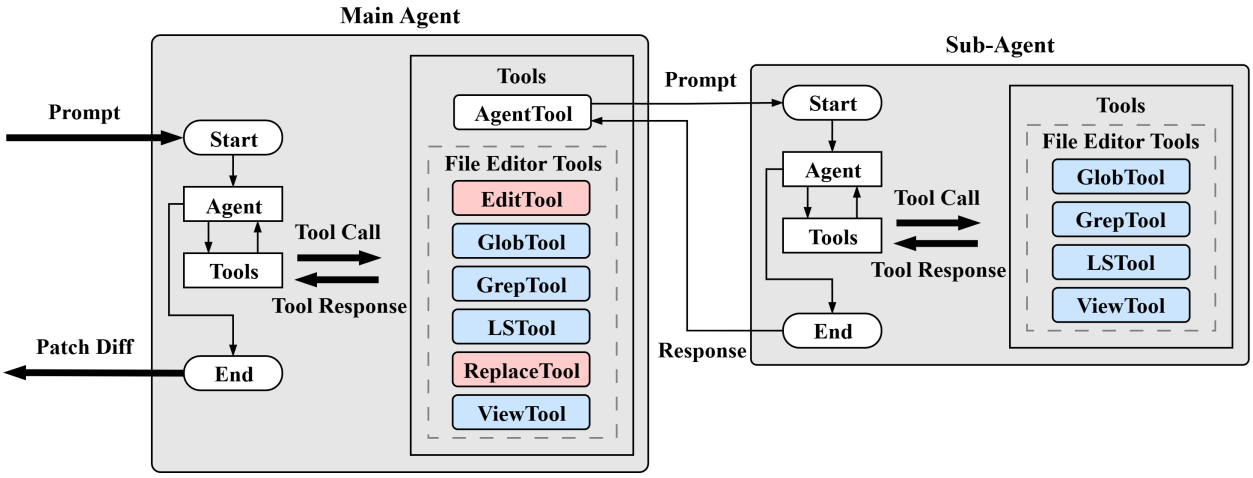


Figure 53: The overall structure of the CLAUDELIKE coder. The read-only file editor tools that can be invoked by the sub-agent are highlighted with blue boxes, while the file editor tools that modify files are highlighted with red boxes.

- **`EditTool(file_path, old_string, new_string)`**: Replaces the `old_string` from the file in `file_path` into `new_string`. The `old_string` must exist in the file and be uniquely matched in the file.
- **`GlobTool(pattern, path)`**: Returns the list of files in the `path` directory whose filenames match the glob pattern.
- **`GrepTool(pattern, path, include)`**: Returns the list of files in the `path` directory whose content match the given regex expression `pattern`. The result only includes files whose filenames match the glob pattern `include`.
- **`LSTool(path)`**: Returns the list of files in the `path` directory.
- **`ReplaceTool(file_path, content)`**: Replaces the content of file `file_path` into given `content`. If the file `file_path` does not exist, `ReplaceTool` creates it and sets the content of the file as `content`.
- **`ViewTool(file_path, offset, limit)`**: Returns the maximum `limit` lines of the file `file_path` starts from line number `offset`.

AgentTool. The primary feature of the CLAUDELIKE coder is the `AgentTool`, which decomposes complex tasks by dispatching a sub-agent. Invoked as `AgentTool(prompt)`, it spawns a ReAct-based sub-agent to perform the task described in the `prompt` and return the result. This mechanism is particularly useful for operations such as searching for keywords in files or identifying the role of specific source code, which frequently arise during patching but require multiple tool calls. By offloading these operations to a sub-agent, the main agent can avoid retaining unnecessary context and focus on essential reasoning for patch generation.

The sub-agent operates with the same interface as the main agent but with a restricted tool set to prevent unintended file modifications. Specifically, it may call only `GlobTool`, `GrepTool`, `LSTool`, and `ViewTool`, ensuring that its role is limited to navigation and inspection. Furthermore, the sub-agent cannot invoke `AgentTool`, preventing recursive spawning of agents and eliminating the risk of infinite loops. These restrictions preserve the integrity of the codebase while maintaining controlled and predictable task delegation.

7.9.4 Differences from Claude Code

The CLAUDELIKE coder is inspired by the architecture and tool design of Claude Code but incorporates several key differences. First, we removed tools that are not well suited for patching, such as those designed

for Jupyter notebooks (e.g., `ReadNotebook`, `NotebookEditCell`) or tools requiring an online environment (e.g., `WebFetchTool`). This reduction decreases both the context size and engineering overhead. Furthermore, we redesigned the system prompts and tool prompts to better align the agent's behavior with the requirements of automated patch generation.

7.10 Open-Source Agents

In addition to our internally developed agents, ATLANTIS-Patching integrates selected open-source agents to further enhance the diversity and robustness of patch generation. These agents bring complementary strengths developed and refined by the broader community, enabling our system to explore problem-solving approaches that differ from our in-house designs. In particular, we employ AIDER [21] and SWE-AGENT [45], each of which specializes in distinct patching strategies and workflows.

However, integrating these open-source agents also introduces practical considerations: their embedded prompts are typically required to have long context windows, which can lead to excessive context usage. To mitigate these issues, we adopted more affordable models like *Gemini 2.5 Pro* and *O4 Mini*.

7.10.1 Aider

AIDER [21] is an open-source agent designed for precise, instruction-guided code modifications. It is particularly effective at handling straightforward bugs where the fix can be expressed as a small, localized edit. While it may lack the advanced reasoning capabilities needed for complex, multi-step problems, AIDER excels in applying clear, direct patches with a high success rate when the task is simple. In ATLANTIS-Patching, we use AIDER as a reliable "lifeguard" agent; if a problem is easy but happens to be overlooked or overcomplicated by our more sophisticated agents, AIDER can step in and resolve it quickly. This role ensures that trivial fixes are not missed, allowing the system to capture low-hanging opportunities without unnecessary reasoning overhead.

7.10.2 SWE-Agent

SWE-AGENT [45] is an open-source agent built for comprehensive, iterative bug repair. It employs a multi-step reasoning process that includes analyzing code context, executing relevant commands, and refining its proposed patches based on observed outcomes. While ATLANTIS-Patching already incorporates agents with a variety of strategies, including multi-step reasoning, we include SWE-AGENT because it is one of the most widely used open-source multi-step repair agents. Its workflow has been exercised and refined by a large community of users, giving us confidence that its reasoning patterns and iterative refinement loop are effective in practice. By incorporating SWE-AGENT, we add a well-validated, community-tested problem solver to our agent pool, further diversifying the approaches explored and increasing the likelihood of finding a correct patch.

7.11 Evaluation

This section presents the performance evaluation of ATLANTIS-Patching, our automated patching system. We evaluate both individual agents and the end-to-end system. To assess the practicality and efficiency of each agent, we use three metrics: patch success rate, Time-to-Patch (TTP), and LLM cost. For fair comparison, we run all benchmarks on a uniform hardware platform with Intel(R) Xeon(R) Gold 6346 CPU (16 cores @ 3.10GHz), 256 GB of DDR4 RAM, installed with Ubuntu 22.04.

7.11.1 Evaluation Metrics

To quantitatively assess the performance of each agent, we employ three key metrics: patch success rate, Time-to-Patch (TTP), and LLM cost. These metrics provide a comprehensive view of each agent's effectiveness, efficiency, and resource utilization in the context of automated patch generation.

Patch Success Rate. In our evaluation, a patch is considered plausible if it resolves the identified vulnerability without introducing new issues, as verified by the provided proof-of-concept (PoC) exploit and ensuring the internal test passes successfully. However, if a patch fails to compile, does not fix the vulnerability, or exceeds the time limit, it is deemed unsuccessful. We planned to report the internal test failure cases separately since they are unable to be verified in the competition environment, but in our evaluation, we only encountered such cases during the final competition postmortem analysis. The patch success rate is calculated as the ratio of successfully generated plausible patches to the total number of cases attempted by each agent. This metric is crucial for understanding the effectiveness of each agent in generating plausible and reliable patches.

Time-to-Patch (TTP). Time-to-Patch (TTP) is the elapsed time required for an agent to generate a patch. Most of this duration arises from LLM interactions and project builds, so a higher TTP often indicates more build attempts or more frequent and complex LLM calls. In the AIxCC competition [14], shorter TTP is beneficial because it increases the score through time multipliers [15]. Beyond competitive settings, reducing TTP improves system reliability and maintainability by minimizing downtime, enhancing service availability, and lowering operational overhead in large-scale deployments.

LLM Cost. LLM cost measures the total expenditure associated with language model usage to generate a patch. Generally, the cost depends on the model; in our system, each agent is tied to a fixed model, as listed in Table 13, and thus directly incurs the corresponding cost. In the AIxCC competition, agents must operate under a strict LLM budget, so reducing this cost is desirable because it allows more effective use of the limited budget.

7.11.2 Microbenchmark

To evaluate the performance of each agent, we conducted a microbenchmark using the Round 3 dataset from the AIxCC competition.

Dataset - Exhibition Round 3. We used the Round 3 dataset from the AIxCC competition [14] for our microbenchmark. The Round 3 dataset contains 34 vulnerabilities drawn from nine challenge projects, comprising 16 in C and 18 in Java. This dataset was officially released by the organizers as part of an unscored exhibition round. Since our evaluation used the final version of the agent, which had been further developed after Round 3 was released, the results can suffer from data contamination due to overfitting. Nevertheless, because the dataset was selected independently by the organizers and includes diverse vulnerabilities in both C and Java, we believe that it remains a useful benchmark for examining relative performance of agents.

Figure 54 shows the statistics of the Round 3 dataset. As illustrated, the dataset includes a wide range of bug types with varying patch sizes. Specifically, it includes memory safety issues (e.g., buffer overflows, null pointer dereferences) and also logic errors (e.g., server side request forgery, XXE injection, Zip Slip). Moreover, the dataset covers not only single-hunk patches but also multi-hunk and even multi-file patches, which are often challenging for existing APR techniques.

Results. Table 15 shows the summary of the patch success rate of each agent on the Round 3 dataset, while Figure 55 visualizes the LLM cost and TTP of each agent. We also include the raw data in Table 17 and Table 16. In summary, PRISM achieved the highest patch success rate of 100% (34/34), followed by MULTIRETRIEVAL and PRISM (33/34, 97.1%). Notably, all of our newly designed agents outperformed the open-source agents (AIDER and SWE-AGENT). Nevertheless, as mentioned earlier, we further refined the agents after Round 3 was released, so the results may be influenced by overfitting to this dataset. Therefore, the results should be interpreted as relative performance trends rather than absolute measures of capability.

Rank	Team	Vulnerabilities Found	Correct Patches	Patch Rate (%)
1	Team Atlanta	43	31	72.1
2	Trail of Bits	28	19	67.9
3	Theori	34	20	58.8
4	All You Need IS A Fuzzing Brain	28	14	50.0
5	Shellphish	28	11	39.3
6	42-b3yond-6ug	41	3	7.3
7	Lacrosse	1	1	100.0

Table 14: The summary of patch success rate of each team on the AIxCC final competition.

In terms of TTP and LLM cost, most agents result in similar performance, except for PRISM, which incurred significantly higher TTP and LLM cost. This is because PRISM uses expensive, reasoning models (i.e., o4-mini), and its multi-agent design is more complex than other agents, involving more LLM requests and build attempts. Moreover, AIDER used an extremely low LLM cost; this is because AIDER is not designed for automatic patch generation but for interactive use. As a result, it does not perform iterative patch generation like other agents, leading to fewer LLM requests and lower LLM cost.

7.11.3 End-to-End Performance

Dataset - Exhibition Round 3.5. To evaluate the end-to-end performance of ATLANTIS-Patching, we used Exibition Round 3.5, an internal benchmark we constructed. This dataset consists of 22 CPVs from four challenge projects, all implemented in C. Unlike those in Round 3, it was created after the final version of the agents had been developed, so no tuning or modifications were applied. As a result, the evaluation results are more meaningful than those from Round 3. As mentioned earlier, we used this benchmark to evaluate the end-to-end performance of ATLANTIS-Patching, rather than individual agents. As ATLANTIS-Patching does not run subsequent agents if one agent successfully produces a patch, some agents were not executed for certain cases. For this reason, we do not report TTP or LLM cost here; instead, we present the patch success rate of the executed agents and the overall end-to-end performance.

Results. Table 18 presents the overall results. An interesting observation is that *there is no agent that consistently works well across all cases*. In terms of patch success rate alone, MULTIRETRIEVAL performs the best, but even MULTIRETRIEVAL failed to fix two patches. Interestingly, one of these patches was successfully fixed by MARTIAN and CLAUDELIKE, despite their patch success ratios being only in the 20% range. Moreover, the other was fixed by AIDER. Considering that AIDER is relatively simple without an interactive process, this is a particularly interesting result. It suggests that, in certain cases, complex architectures and tools may actually become a burden for LLMs.

7.11.4 Postmortem Analysis on AIxCC Final Competition

In this subsection, we present a postmortem analysis of the AIxCC final competition. Table 14 shows a summary of final results. Our system, ATLANTIS, not only discovered the largest number of vulnerabilities (43 CPVs in total) but also *produced the most correct patches (31 in total)*. In particular, ATLANTIS-Patching achieved a patch success rate of 72.1%. *This is the highest rate among competing systems except for Lacrosse, which only patched a single vulnerability.* These results highlight that ATLANTIS-Patching attains leading performance among the competition teams.

In more detail, ATLANTIS-Patching produced 31 correct patches through the following steps. In total, ATLANTIS-Patching received 117 PoVs from ATLANTIS's bug-finding system and, after the initial de-duplication, attempted to patch 47 PoVs. Post-patch de-duplication removed two additional duplicates, leaving 45 CPVs. Among these, we could not patch 2 PoVs from the `hertzbeat` project. For the remaining

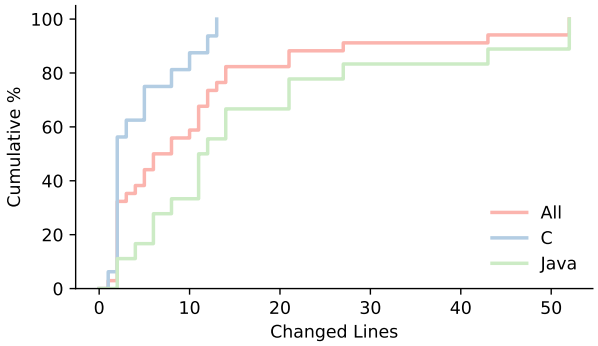
43 CPVs, ATLANTIS-Patching submitted 48 patches; 42 passed internal tests and 6 failed. In particular, ATLANTIS-Patching attempted `mongoose-3`, `pdfbox-5`, and `tika-1` 2, 3, and 3 times, respectively. It produced plausible patches in the final attempts for `mongoose-3` and `tika-1`, but failed in all three attempts for `pdfbox-5`. Overall, we submitted plausible patches for 42 CPVs, and the AIxCC organizers awarded scores to 31 of them after manual inspection. Since the organizers did not disclose the detailed inspection results, we cannot determine which specific patches received scores.

Per-agent Analysis. Similar to the previous experiment, we analyzed the performance of each agent during the AIxCC final competition. To this end, we examined both LLM logs and system logs after the competition. Unfortunately, due to unexpected log truncation, several entries are missing. For instance, ATLANTIS-Patching successfully submitted patches for CPVs, which are `mongoose-1, 3, 4` and `xz-1`, but we could not attribute these fixes to specific agents. Despite these missing records, we believe that the available data are still meaningful and reflect a large-scale evaluation.

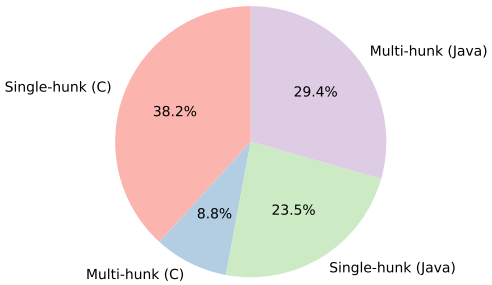
As shown in Table 19, the results of the final competition are consistent with the previous experiments. MULTIRETRIEVAL, PRISM, and VINCENT again achieved strong performance; MULTIRETRIEVAL recorded the highest patch submission rate of 80.56% (29/36), followed by PRISM at 74.36% (29/39) and VINCENT at 72.97% (27/37). MARTIAN showed a relatively low rate of 46.51% (20/43), but it uniquely fixed `shadowsocks-1` and also repaired `pdfbox-5`, where both MULTIRETRIEVAL and VINCENT failed in internal functional tests. Across the final competition, ATLANTIS-Patching produced plausible patches for 42 CPVs, even though we could not identify the responsible agents for 5 patches due to log truncation. This is far more than the results of the best single agent, MULTIRETRIEVAL, which produced 29 patches. This demonstrates that no single agent could solve all problems, making our ensembling crucial for achieving high overall success.

CPV Name	Bug Type	Files	Hunks	+	-	Total
c-curl-1	Stack Buffer Overflow	1	1	1	1	2
c-curl-2	Null Pointer Dereference	1	1	1	1	2
c-freerdp-1	Integer Overflow	2	2	4	1	5
c-freerdp-2	Backdoor	1	1	0	10	10
c-freerdp-3	Heap Buffer Overflow	1	1	4	4	8
c-libpng-1	Stack Buffer Overflow	1	1	1	1	2
c-libxml2-1	Heap Buffer Overflow	1	1	1	1	2
c-libxml2-2	Double Free	1	1	0	1	1
c-libexif-1	Heap Buffer Overflow	1	1	3	9	12
c-libexif-2	Heap Buffer Overflow	1	1	5	0	5
c-libexif-3	Heap Buffer Overflow	1	3	9	4	13
c-sqlite3-1	Stack Buffer Overflow	1	1	1	1	2
c-sqlite3-2	Heap Buffer Overflow	2	3	3	0	3
c-sqlite3-3	Heap Buffer Overflow	1	1	1	1	2
c-sqlite3-4	Null Pointer Dereference	1	1	1	1	2
c-sqlite3-5	Stack Buffer Overflow	1	1	1	1	2
java-commons-compress-0	Server Side Request Forgery	1	2	1	13	14
java-commons-compress-1	Backdoor	1	1	1	3	4
java-commons-compress-2	Backdoor	1	3	24	3	27
java-commons-compress-3	Zip Slip	1	1	1	1	2
java-commons-compress-4	Denial of Service	1	1	1	1	2
java-commons-compress-5	Out of Memory	1	1	4	2	6
java-tika-1	XXE Injection	1	3	2	19	21
java-tika-2	Remote Code Execution	1	2	5	3	8
java-tika-3	Tar Slip	1	2	5	1	6
java-tika-4	Server Side Request Forgery	1	1	0	21	21
java-tika-5	OS Command Injection	1	2	1	42	43
java-tika-6	OS Command Injection	1	2	12	0	12
java-tika-7	Server Side Request Forgery	1	1	0	11	11
java-tika-8	Server Side Request Forgery	1	1	1	13	14
java-tika-9	File Path Traversal	1	2	24	28	52
java-tika-10	Zip Slip	1	2	6	5	11
java-zookeeper-1	Backdoor	1	4	1	51	52
java-zookeeper-2	Denial of Service	1	1	5	6	11

(a) CPVs with bug type, changed files, hunks, and added and removed lines per diff



(b) Cumulative distribution of changed lines

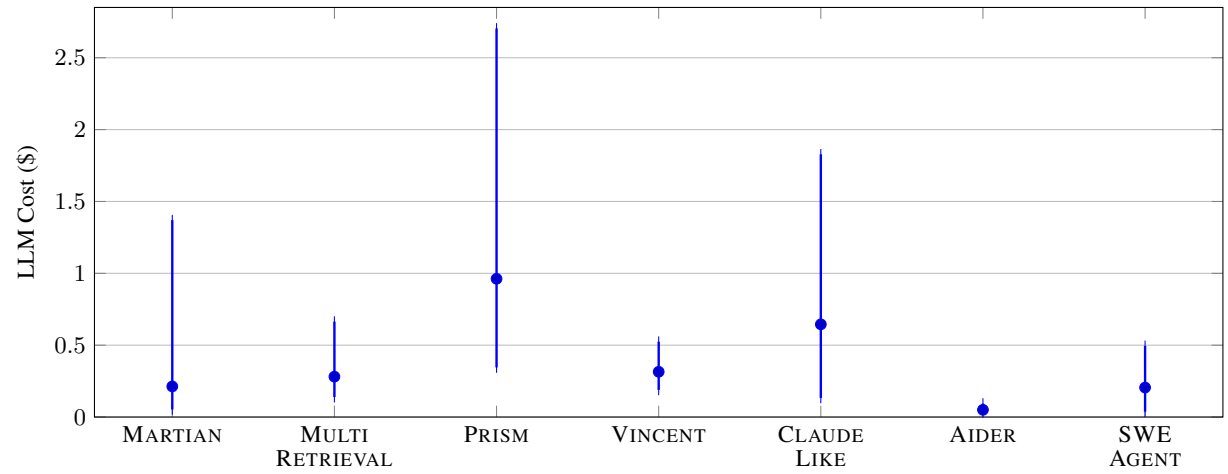


(c) Single vs multi-hunk patches

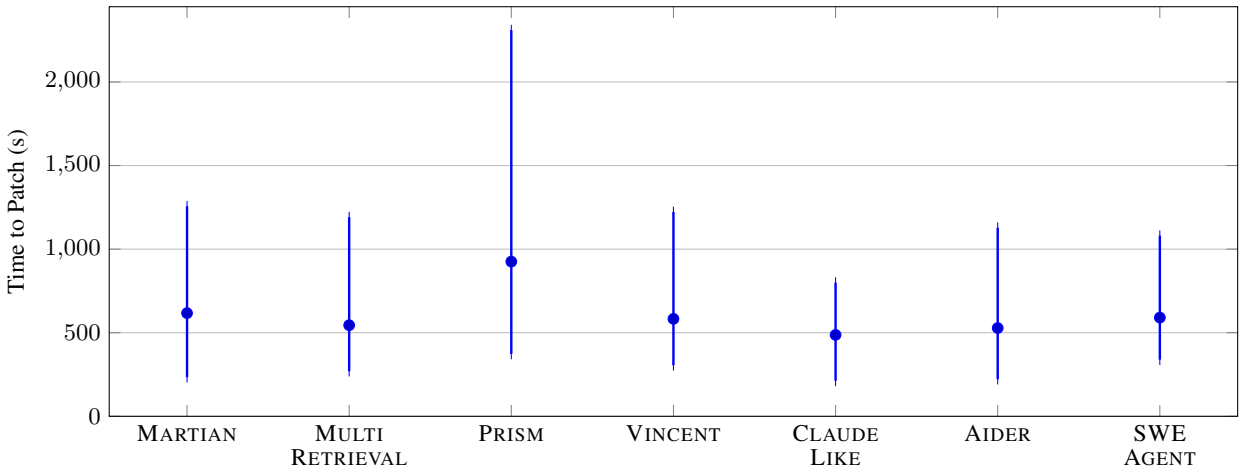
Figure 54: Statistics of the Round 3 benchmark, showing the distributions of changed files, hunks, and added and removed lines per diff. Notably, this distribution is from the ground truth patches provided by the organizers.

CPV Name	Bug Type	MARTIAN	MULTI RETRIEVAL	PRISM	VINCENT	CLAUDE LIKE	AIDER	SWE AGENT
c-curl-1	Stack Buffer Overflow	✓	✓	✓	✓	✓	✓	✓
c-curl-2	Null Pointer Dereference	✓	✓	✓	✓	✓	✗	✓
c-freerdp-1	Integer Overflow	✓	✓	✓	✓	✓	✓	✗
c-freerdp-2	Backdoor	✓	✓	✓	✓	✓	✗	✗
c-freerdp-3	Heap Buffer Overflow	✓	✓	✓	✓	✓	✗	✗
c-libexif-1	Heap Buffer Overflow	✓	✓	✓	✓	✓	✓	✓
c-libexif-2	Heap Buffer Overflow	✓	✓	✓	✓	✓	✓	✗
c-libexif-3	Heap Buffer Overflow	✓	✓	✓	✓	✓	✓	✓
c-libpng-1	Stack Buffer Overflow	✓	✓	✓	✓	✗	✓	✗
c-libxml2-1	Heap Buffer Overflow	✓	✓	✓	✓	✓	✗	✓
c-libxml2-2	Double Free	✓	✓	✓	✓	✓	✓	✓
c-sqlite3-1	Stack Buffer Overflow	✓	✓	✓	✓	✓	✗	✓
c-sqlite3-2	Heap Buffer Overflow	✓	✓	✗	✓	✗	✗	✗
c-sqlite3-3	Heap Buffer Overflow	✓	✓	✓	✓	✓	✗	✓
c-sqlite3-4	Null Pointer Dereference	✓	✓	✓	✓	✗	✗	✗
c-sqlite3-5	Stack Buffer Overflow	✓	✓	✓	✓	✗	✗	✗
java-commons-compress-0	Server Side Request Forgery	✓	✓	✓	✓	✓	✓	✓
java-commons-compress-1	Backdoor	✓	✓	✓	✓	✓	✓	✓
java-commons-compress-2	Backdoor	✗	✗	✓	✓	✗	✗	✗
java-commons-compress-3	Zip Slip	✓	✓	✓	✓	✓	✓	✓
java-commons-compress-4	Denial of Service	✓	✓	✓	✓	✓	✓	✗
java-commons-compress-5	Out of Memory	✗	✓	✓	✓	✗	✗	✗
java-tika-1	XXE Injection	✓	✓	✓	✓	✓	✓	✗
java-tika-2	Remote Code Execution	✓	✓	✓	✓	✓	✓	✗
java-tika-3	Tar Slip	✓	✓	✓	✓	✓	✓	✓
java-tika-4	Server Side Request Forgery	✓	✓	✓	✓	✓	✗	✓
java-tika-5	OS Command Injection	✓	✓	✓	✓	✓	✗	✗
java-tika-6	OS Command Injection	✗	✓	✓	✓	✗	✗	✗
java-tika-7	Server Side Request Forgery	✓	✓	✓	✓	✓	✓	✗
java-tika-8	Server Side Request Forgery	✓	✓	✓	✓	✓	✗	✓
java-tika-9	File Path Traversal	✗	✓	✓	✓	✓	✓	✓
java-tika-10	Zip Slip	✓	✓	✓	✓	✓	✓	✓
java-zookeeper-1	Backdoor	✓	✓	✓	✓	✓	✓	✓
java-zookeeper-2	Denial of Service	✓	✓	✓	✓	✓	✗	✗
Patch Success Rate		30 / 34 (88.24%)	33 / 34 (97.06%)	33 / 34 (97.06%)	34 / 34 (100.00%)	27 / 34 (79.41%)	18 / 34 (52.94%)	17 / 34 (50.00%)

Table 15: Patch success results for 34 CPVs from the Round 3 dataset. A checkmark (✓) indicates a plausible patch, while a cross (✗) denotes failure due to various reasons including patches that are still vulnerable, uncompilable, or exceed the time limit. The bottom row summarizes the total number of plausible patches out of the total attempts for each agent.



(a) LLM Cost per patch – trimmed mean with P10–P90 range



(b) TTP per patch – trimmed mean with P10–P90 range

Figure 55: Per-agent summary across 34 CPVs (Round 3). Dots show **the per-method mean of the central 80%** (i.e., with the lowest and highest 10% removed) for LLM cost and time to patch (TTP). Vertical bars show **the central 80% range (P10–P90)** of observed values, indicating distributional spread. Missing entries (–) were excluded. Lower is better.

CPV Name	MARTIAN	MULTI RETRIEVAL	PRISM	VINCENT	CLAUDE LIKE	AIDER	SWE AGENT
c-curl-1	818	300	456	335	641	**276**	374
c-curl-2	287	**272**	359	336	329	-	301
c-freerdp-1	582	423	426	390	1026	**224**	-
c-freerdp-2	**329**	618	534	330	459	-	-
c-freerdp-3	348	285	417	309	**274**	-	-
c-libexif-1	273	272	319	331	212	**183**	768
c-libexif-2	217	270	364	307	**167**	268	-
c-libexif-3	238	307	409	305	453	**218**	438
c-libpng-1	1002	358	581	467	-	**303**	-
c-libxml2-1	**228**	328	511	283	424	-	307
c-libxml2-2	**230**	369	581	262	234	281	394
c-sqlite3-1	1304	610	1016	**472**	655	-	495
c-sqlite3-2	599	997	-	**566**	-	-	-
c-sqlite3-3	1251	650	768	**538**	603	-	1084
c-sqlite3-4	724	640	798	**587**	-	-	-
c-sqlite3-5	524	688	898	**511**	-	-	-
java-commons-compress-0	**370**	564	852	428	398	396	404
java-commons-compress-1	427	522	706	429	579	541	**411**
java-commons-compress-2	-	-	2503	**725**	-	-	-
java-commons-compress-3	518	732	672	**473**	722	548	542
java-commons-compress-4	**518**	623	948	475	538	676	-
java-commons-compress-5	-	1604	1749	**847**	-	-	-
java-tika-1	**235**	1157	313	826	303	522	-
java-tika-2	1916	379	644	1273	**185**	1144	-
java-tika-3	**545**	1845	1337	911	717	779	901
java-tika-4	867	**199**	2450	813	213	-	638
java-tika-5	1080	**166**	1317	1070	813	-	-
java-tika-6	-	**573**	2852	1826	-	-	-
java-tika-7	718	**236**	693	1160	791	1160	-
java-tika-8	1350	1200	1704	1249	**457**	-	1078
java-tika-9	-	**600**	1303	1262	1557	1121	662
java-tika-10	1122	1232	1717	**224**	603	593	360
java-zookeeper-1	563	**402**	2691	504	421	558	1755
java-zookeeper-2	877	572	1134	628	**362**	-	-

Table 16: Measured TTP (in seconds) for 34 CPVs from the Round 3 dataset, comparing the performance of each integrated agent. Rows correspond to individual CPVs and columns to the respective agents. Dash (-) indicates that the agent did not produce a patch.

CPV Name	MARTIAN	MULTI RETRIEVAL	PRISM	VINCENT	CLAUDE LIKE	AIDER	SWE AGENT
c-curl-1	0.382	0.138	0.836	0.257	0.875	**0.032**	0.110
c-curl-2	0.086	0.101	0.341	0.208	0.264	-	**0.027**
c-freerdp-1	0.119	0.398	0.463	0.485	4.450	**0.093**	-
c-freerdp-2	**0.106**	0.309	1.180	0.295	0.984	-	-
c-freerdp-3	0.272	**0.153**	0.385	0.279	0.206	-	-
c-libexif-1	0.077	0.319	0.286	0.366	0.268	**0.037**	0.602
c-libexif-2	**0.055**	0.350	0.400	0.357	0.131	0.089	-
c-libexif-3	0.062	0.641	0.522	0.483	1.440	**0.053**	0.276
c-libpng-1	6.130	0.174	0.664	0.456	-	**0.095**	-
c-libxml2-1	**0.054**	0.204	0.461	0.211	0.320	-	0.097
c-libxml2-2	**0.051**	0.389	0.365	0.161	0.177	0.289	0.146
c-sqlite3-1	1.460	**0.093**	1.210	0.187	1.750	-	0.098
c-sqlite3-2	0.629	0.626	-	**0.366**	-	-	-
c-sqlite3-3	4.040	**0.252**	1.310	0.294	1.900	-	0.424
c-sqlite3-4	0.171	**0.159**	0.715	0.353	-	-	-
c-sqlite3-5	0.328	0.670	1.900	**0.244**	-	-	-
java-commons-compress-0	0.053	0.184	0.834	0.194	0.106	0.038	**0.029**
java-commons-compress-1	0.052	0.162	0.491	0.183	0.205	**0.026**	0.044
java-commons-compress-2	-	-	2.830	**0.499**	-	-	-
java-commons-compress-3	0.391	0.186	0.325	0.593	2.040	**0.023**	0.174
java-commons-compress-4	0.113	0.143	0.434	0.201	0.740	**0.068**	-
java-commons-compress-5	-	1.140	2.200	**0.536**	-	-	-
java-tika-1	0.109	0.191	1.090	0.308	1.760	**0.023**	-
java-tika-2	1.360	0.883	1.730	0.745	0.154	**0.040**	-
java-tika-3	0.063	0.161	0.991	0.203	0.081	**0.017**	0.043
java-tika-4	**0.109**	0.204	3.040	0.202	0.303	-	0.363
java-tika-5	0.123	**0.084**	0.704	0.377	0.660	-	-
java-tika-6	-	**0.413**	2.940	2.340	-	-	-
java-tika-7	0.071	0.934	0.441	0.212	0.135	**0.048**	-
java-tika-8	**0.085**	0.273	0.988	0.186	1.780	-	0.305
java-tika-9	-	0.278	0.871	0.345	0.286	**0.040**	0.203
java-tika-10	0.059	0.192	0.332	0.200	0.191	**0.050**	0.162
java-zookeeper-1	0.047	0.162	5.790	0.319	0.167	**0.044**	1.480
java-zookeeper-2	0.227	0.262	1.630	0.396	**0.140**	-	-

Table 17: Measured LLM cost (USD) for 34 CPVs from the Round 3 dataset, comparing the performance of each integrated agent. Rows correspond to individual CPVs and columns to the respective agents. Dash (-) indicates that the agent did not produce a patch.

CPV Name	Bug Type	MARTIAN	MULTI RETRIEVAL	PRISM	VINCENT	CLAUDE LIKE	AIDER	SWE AGENT
c-binutils-1	Use After Free	✗	✓	✓	✗	✗	-	✗
c-binutils-2	Use After Free	✗	✓	✗	✗	✗	✗	-
c-binutils-3	Heap Buffer Overflow	✗	✓	✓	✓	-	-	-
c-binutils-4	Null Pointer Dereference	✓	✓	✓	✓	-	-	-
c-ffmpeg-1	Heap Buffer Overflow	✗	✓	✓	✓	-	-	-
c-ffmpeg-2	Heap Buffer Overflow	✗	✓	✓	✓	-	-	-
c-ffmpeg-3	Heap Buffer Overflow	✓	✓	✓	✓	-	-	-
c-ffmpeg-4	Stack Buffer Overflow	✗	✓	-	✓	-	-	-
c-ffmpeg-5	Heap Buffer Overflow	✓	✓	✓	✓	-	-	-
c-ffmpeg-6	Heap Buffer Overflow	✗	✓	✓	✓	-	-	-
c-ffmpeg-7	Timeout	✗	✓	✓	✓	-	-	-
c-ffmpeg-8	Heap Buffer Overflow	✓	✓	✓	✓	-	-	-
c-ffmpeg-9	Heap Buffer Overflow	✓	✓	✓	✓	-	-	-
c-ffmpeg-10	Stack Buffer Overflow	✗	✓	✗	✓	-	✓	-
c-ffmpeg-11	Stack Buffer Overflow	✗	✓	✓	✓	-	-	-
c-pcre2-1	Heap Buffer Overflow	✗	✓	✓	✓	-	-	-
c-pcre2-2	Heap Buffer Overflow	✗	✗	✗	✗	-	✓	✗
c-pcre2-3	Heap Buffer Overflow	✗	✓	✓	✗	✗	-	✗
c-pcre2-4	Heap Buffer Overflow	✗	✓	✓	✓	-	-	-
c-sleuthkit-1	Null Pointer Dereference	✗	✓	✗	✓	-	✗	-
c-sleuthkit-2	Timeout	✗	✓	✗	✓	-	✓	-
c-sleuthkit-3	Timeout	✓	✗	✗	✗	✓	✗	✗
Patch Success Rate		6 / 22 (27.27%)	20 / 22 (90.91%)	15 / 21 (71.43%)	17 / 22 (77.27%)	1 / 4 (25.00%)	3 / 6 (50.00%)	0 / 4 (0.00%)

Table 18: Patch success results for 22 CPVs from the Round 3.5 dataset. A checkmark (✓) indicates a plausible patch, while a cross (✗) denotes failure due to various reasons including patches that are still vulnerable, uncompilable, or exceed the time limit. Dash (-) in the tables indicates that the agent was not executed for that CPV, as a plausible patch had already been found and submitted. The bottom row summarizes the total number of plausible patches out of the total attempts for each agent.

CPV Name	Bug Type	MARTIAN	MULTI RETRIEVAL	PRISM	VINCENT	CLAUDE LIKE	AIDER	SWE AGENT
java-compress-1	File Path Traversal	✓	✓	✓	✓	-	-	-
java-compress-2	Null Pointer Dereference	✓	✓	✓	✓	-	-	-
java-compress-3	File Path Traversal	✗	✓	✓	✓	-	-	-
java-compress-4	File Path Traversal	✓	✓	✓	✓	-	-	-
c-curl-2	Heap Buffer Overflow	✗	✓	✗	✗	-	✗	-
c-curl-3	Format String Bug	✓	✓	✓	✓	-	-	-
c-curl-4	Heap Buffer Overflow	✓	✗	✓	✓	✗	-	-
c-curl-5	Null Pointer Dereference	✗	✓	✓	✓	-	-	-
java-dicoogle-1	Number Format Exception	✓	✓	✓	✓	-	-	-
c-freerdp-1	Heap Buffer Overflow	✗	✓	✓	✓	-	-	-
java-healthcare-1	Out Of Memory	✗	✓	✗	✗	-	✗	-
java-healthcare-2	Index Out Of Bounds	✗	✓	✗	✓	-	-	✗
java-hertzbeat-1	Out Of Memory	✗	✗	✗	✗	-	✗	-
java-hertzbeat-2	Index Out Of Bounds	✗	✗	✗	✗	✗	✗	✗
java-log4j2-1	Remote JNDI Lookup	✗	✓	-	-	-	-	-
c-mongoose-1	Memory Leak	✗	✗	✗	✗	-	✗	-
c-mongoose-2	Stack Buffer Overflow	✓	✓	✓	✓	-	-	-
c-mongoose-3	Heap Buffer Overflow	✗	-	✗	✗	-	✗	-
c-mongoose-4	Stack Buffer Overflow	✗	-	✗	-	-	-	-
java-pdfbox-1	Stack Overflow	✓	✓	✓	✓	-	-	-
java-pdfbox-2	Stack Overflow	✗	✓	✓	✗	-	-	-
java-pdfbox-3	Timeout	✗	✓	✓	✓	-	-	-
java-pdfbox-4	No Class Def Found Error	✓	✓	✓	✓	-	-	-
java-pdfbox-5	OS Command Injection	✓	⚠	✓	⚠	-	-	-
java-pdfbox-6	Invocation Target Exception	✓	✓	✓	✓	-	-	-
c-shadowsocks-1	Heap Buffer Overflow	✓	-	-	-	-	-	-
java-tika-1	Timeout	✗	✓	✓	✓	-	-	-
c-wireshark-1	Use After Free	✓	✓	✓	✗	-	✓	-
c-wireshark-2	Stack Buffer Overflow	✓	✓	✓	✓	-	-	-
c-wireshark-3	Null Pointer Dereference	✓	✓	✓	✓	-	-	-
c-wireshark-4	Format String Bug	✗	✗	✗	✓	✗	-	✗
c-wireshark-5	Heap Buffer Overflow	✗	✗	✗	✓	-	-	✗
c-wireshark-6	Null Pointer Dereference	✗	✓	✓	✓	-	-	-
c-wireshark-7	Stack Buffer Overflow	✗	✓	✓	✓	-	-	-
c-wireshark-8	Use After Free	✓	✓	✓	✓	✗	-	-
c-wireshark-9	Stack Buffer Overflow	✗	✓	✓	✗	-	✗	-
c-wireshark-10	Heap Buffer Overflow	✓	✓	✓	✓	-	-	-
c-wireshark-11	Stack Buffer Overflow	✓	✓	✓	✓	-	-	-
c-wireshark-12	Heap Buffer Overflow	✓	✓	✓	✓	-	-	-
c-wireshark-13	Global Buffer Overflow	✓	✓	✓	✓	-	-	-
c-wireshark-14	Global Buffer Overflow	-	-	✓	-	-	-	-
c-xz-1	Use After Free	✗	-	-	-	-	-	-
Patch Success Rate		20 / 41 (48.78%)	29 / 36 (80.56%)	29 / 39 (74.36%)	27 / 37 (72.97%)	0 / 4 (0.00%)	1 / 8 (12.50%)	0 / 4 (0.00%)

Table 19: Patch success results for 42 CPVs from the AIxCC Final Competition. A checkmark (✓) indicates a plausible patch, while a cross (✗) denotes failure due to various reasons including patches that are still vulnerable, uncompilable, or exceed the time limit. A warning (⚠) indicates a patch that was submitted but scored as incorrect due to the hidden functionality tests. Dash (-) in the tables indicates that the agent was not executed for that CPV, as a plausible patch had already been found and submitted. The bottom row summarizes the total number of plausible patches out of the total attempts for each agent.

```c
// src/http/modules/ngx_http_userid_filter_module.c
typedef struct {
    uint32_t    uid_got[4];
    uint32_t    uid_set[4];
    ngx_str_t   cookie;
    ngx_uint_t  reset;
} ngx_http_userid_ctx_t;

// src/core/ngx_string.h
typedef struct {
    size_t      len;
    u_char     *data;
} ngx_str_t;
```

Figure 56: Necessary `Typedef` definitions of `challenge- 004-nginx-cp/cpv15` for successful patching.

8 Custom LLMs in ATLANTIS-Patching

Large language models (LLMs) have recently demonstrated superior performance on code-related tasks [43, 45], *e.g.,* achieving success rates of 60-76% on SWE-Bench Verified [28]. Despite these advancements, *effectively* and *efficiently* providing LLMs with relevant context for complex coding tasks, such as *security patch generation* for AIxCC, remains challenging.

Practical experience demonstrates that effective *context engineering*—the strategic selection and presentation of task-relevant information—is crucial for agent performance [44]. Even the most advanced agents struggle when deprived of appropriate documentation, API references, or source code context, much like skilled developers facing unfamiliar codebases [30].

We *reconfirm that context engineering is also crucial for security patch generation* during the postmortem analysis of the AIxCC Semifinal. In particular, as a controlled experiment, we examine the one of AIxCC Semifinal challenges, `challenge-004-nginx-cp/cpv15` [1]. In this experiment, we observed that our baseline patching agent Aider [21] generates a patch without retrieving the `typedef` definitions for two key structures from the Nginx codebase (*i.e.,* Figure 56) consistently produced an uncompilable patched codebase. However, enforcing to retrieve the key `typedef` definitions yielded the successful compilation of patched codebase. We empirical validated this by conducting 20 random experiments where the patching performance is in Table 20. As can be seen, the clear separation between "with `typedef`" and "without `typedef`" runs demonstrates the decisive role of accurate context retrieval in enabling valid patch generation. Based on this critical performance leap, our direction of patching agents in CRETE is more focused on retrieving missing definitions, *i.e., context engineering for missing context in patching*, leading to our first performant agent `MultiRetrieval`.

Building on this achievement, we further adances our direction from *code context engineering*—a largely manual, heuristic-driven process—towards *code context learning*, in which an agent automatically learns to identify and retrieve missing and relevant code artifacts for a patching task at hand. This shift enables a learnable approach that adapts dynamically to varying bug contexts and codebases.

8.1 Problem: Code Context Learning

We consider LLM-based security patch generation, *i.e.,* given a code base and a vulnerability-reporting or crash log, generate a patch to fix the vulnerability while maintaining functionalities. Inspired by our observation in Table 20, we focus on selecting missing but right contexts for patching with the following requirements.

[1] https://github.com/aixcc-public/challenge-004-nginx-cp/blob/main/.internal_only/cpv15

	Build Attempts																				Success Rate
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
w/o `typedef`	✘	❍	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	❍	❍	✘	✘	❍	❍	✘	5/20
w/ `typedef`	❍	✘	❍	❍	❍	❍	✘	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍	18/20

Table 20: Ablation study on `challenge-004-nginx-cp /cpv15` using AIDER, evaluating the impact of type definitions on build success rate. Each cell shows whether a generated patch compiled successfully (❍) or failed (✘) across 20 repeated queries. Including type definitions led to a markedly different success rate compared to excluding them.

- **Context window limitations.** Even with today's long-context models, supplying an entire large-scale codebase is infeasible due to attention complexity and window size limits. To avoid this issue, a patching agent must operate with a window that is orders of magnitude smaller than the full repository.
- **Prohibitive API costs.** Processing very large inputs with agent is financially expensive. Repeatedly sending thousands lines to a agent for patching would be unsustainable.

To this end, we formulate our code context learning in reinforcement learning (RL). In particular, let $\Delta(\mathcal{S})$ be a distribution over a set $\mathcal{S}$, T be the number of challenge project vulnerabilities (CPVs), H be the number of turns, and $\mathcal{X}$ be a set of token sequences, which is a set of states in the standard RL. Here, $x_{t,h} \in \mathcal{X}$ represents an input at the t-th CPV and the h-th turn. For example, $x_{t,1}$ represents an instruction for LLMs, meta data of the t-th CPV, and the summary of a crash log, and $x_{t,h}$ for $h > 1$ includes the summary of $x_{t,1}$ and retrieved code contexts after h turns. Finally, $\mathcal{A} \coloneqq \mathcal{X}$ be a set of actions for code context retrieval, consist of tokens sequences to represent the list of symbol names to be retrieved, *e.g.,* function and struct definitions.

We consider a policy for code context retrieval $\pi : \mathcal{X} \to \Delta(\mathcal{A})$, which chooses to retrieve necessary code contexts. In particular, This policy chooses a set of actions, requesting to retrieves the definitions of symbols (*e.g.,* retrieve the definition of a `ngx_http_userid_ctx_t` struct). Note that different to a conventional policy in RL, our context policy can select multiple actions, where the policy is learned to select actions of retrieving undefined symbols useful for correct patch generation.

In code context learning, we find a policy π for code context retrieval that maximizes the average return over T episodes of CPVs.

$$\max_{\pi} \frac{1}{T} \sum_{t=1}^{T} \sum_{h=1}^{H} R(x_{t,h}, a_{t,h}),$$

where $R : \mathcal{X} \times \mathcal{A} \to [0, 1]$ is a reward function, which will be defined in the following section, and $a_{t,h}$ is an action chosen by π given $x_{t,h}$.

Remark. *We can significantly reduce code context size via code context learning. In particular, feeding the entire code context for patching may provide successful patch generation but prohibitive due to the limited context size and API costs for LLMs. We overcome these challenges by learning to retrieve necessary code contexts for vulnerability patching via RL.*

Note that in contrast to *in-context learning*, which relies on passively conditioning on provided input sequences, our *code context learning* framework actively acquires and refines relevant source code artifacts through an explicit retrieval policy. This enables adaptive, targeted construction of context tailored to each vulnerability.

8.2 Solution: Learning to Retrieve Code Context

Our approach pursues three primary objectives:

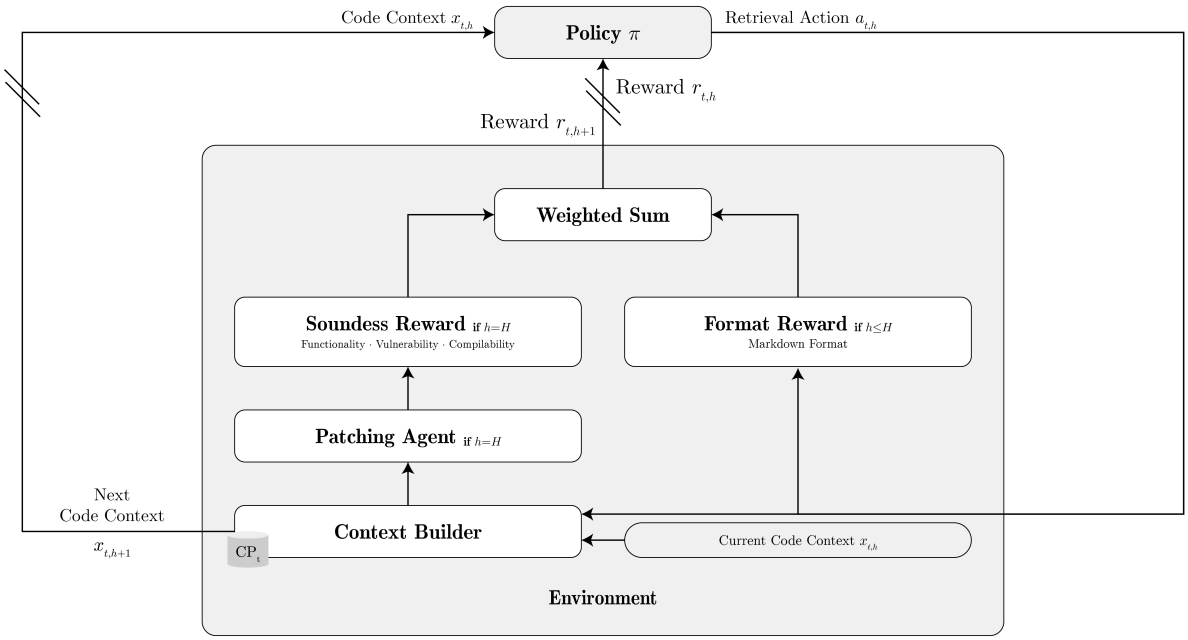


Figure 57: RL setup for code patching via multi-turn retrieval. The policy π iteratively conditions on contextual states $x_{t,h}$ to produce retrieval actions $a_{t,h}$. The environment integrates retrieved artifacts with a challenge project CP_t to construct subsequent states $x_{t,h+1}$. The reward $r_{t,h+1}$ combines format validation (structural compliance) of retrieval actions and soundness assessment (patch quality) of a patched codebase, with aggregated rewards guiding policy optimization across multi-turn interactions for code retrieval.

- **Modeling code context retrieval agent.** We develop a specialized retrieval module that identifies concise, highly relevant code snippets for downstream patch generation. Unlike existing *generic* methods, this retrieval agent is trainable to be *specialized* for security patching on specific bug contexts.
- **Modeling a multi-turn interactive retrieval.** Human developers rarely locate a bug's root cause in one step; instead, they form a hypothesis, examine some code, run tests, gather clues, and then refine their search. We want our agent to emulate this process with multiple retrieval turns. At each step, it can utilize information gathered in previous steps to decide what to read or fetch next. By explicitly modeling retrieval as a multi-turn dialogue with the codebase, our system aims to replicate the way a person would gradually narrow down the fault, leading to have concise code context.
- **Learning the retrieval policy for patching success.** We train our multi-turn code retrieval agent via reinforcement learning to tailor our agent for efficiently and effectively adapt to security patch generation. To this end, we consider repair success our reward function.

8.2.1 Learning Setup

Our approach is multi-turn retrieval-based code patching via RL, as illustrated its setup in Figure 57. The learning process begins with an initial code context $x_{t,1}$ derived from a crash log along with a system prompt. At each turn $h \in \{1, \ldots, H\}$, our retrieval agent, represented by a policy π, conditions on the current contextual state $x_{t,h}$ to produce a set of retrieval actions $a_{t,h}$. The action is defined as the retrieval actions of code symbols (*e.g.,* functions, structures, or types) from the target codebase where the format validity of retrieval actions is measured by a format reward $r_{t,h+1}$. The environment then integrates these

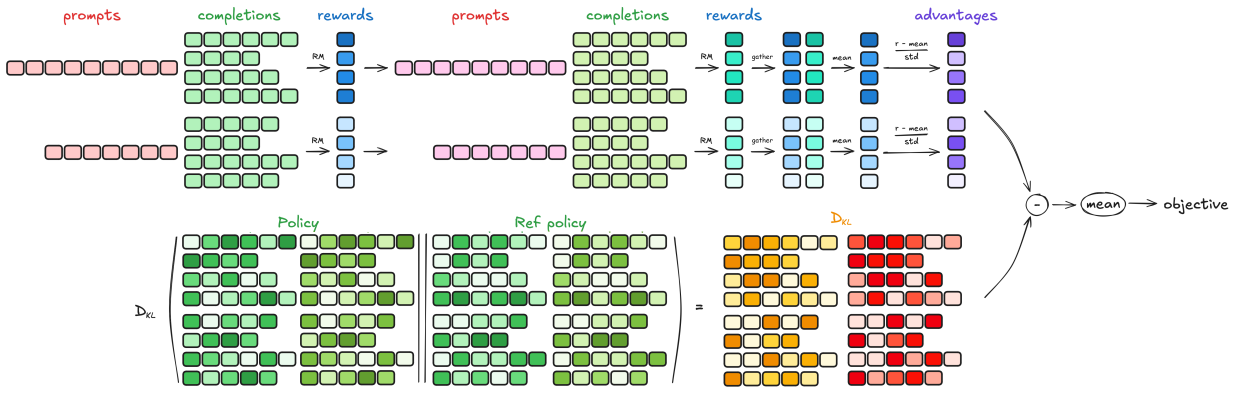


Figure 58: Multi-turn GRPO training flow (adapted and re-colored from the original Hugging Face diagram). For each batch of prompts (pink), the policy generates multi-turn completions (green); a reward model assigns per-turn rewards (shades of blue). Rewards are centered and scaled to yield advantages (purple) that drive the policy update. In parallel, token-wise D_{KL} between the current and reference policies is computed (orange→red). The final loss balances reward maximization and regularization.

retrieved artifacts with the challenge project CP_t, constructing the subsequent state $x_{t,h+1}$ (*i.e.,* aggregated code context). After H turns, the final aggregated context $x_{t,H}$ is passed to an external general-purpose language model (*e.g.,* GPT-4, Claude, or Gemini) for patch synthesis where the soundness of the patched codebase is measured by the soundness reward $r_{t,H+1}$. Importantly, this architecture deliberately decouples context acquisition from patch generation, allowing us to leverage state-of-the-art LLMs while maintaining precise control over the information flow.

8.2.2 Code Context Learning via Multi-turn GRPO

Overview. At a high level, our training framework adapts Group Relative Policy Optimization (GRPO) [40] to the multi-turn retrieval setting. Unlike standard single-turn GRPO, where each prompt produces a single completion, our agent generates a trajectory of retrieval actions over multiple turns, each conditioned on the evolving context. To ensure effective learning, we define a composite reward that provides both intermediate feedback on the structural validity of retrievals and a final signal tied to patch soundness. This design allows the agent to refine its retrieval policy step by step, gradually constructing the minimal yet sufficient context required for secure patch generation. The following sections detail the reward modeling, training objective, and online learning procedure.

Reward Modeling. To guide the agent toward effective retrieval strategies, we employ a composite reward signal. For a given state-action pair $(x_{t,h}, a_{t,h})$, the reward is defined as:

$$R(x_{t,h}, a_{t,h}) = \lambda_{\text{fmt}}\, R_{\text{fmt}}(x_{t,h}, a_{t,h}) + \lambda_{\text{snd}}\, R_{\text{snd}}(x_{t,h}, a_{t,h})$$

where λ_{fmt} and λ_{snd} are non-negative weighting coefficients. The reward consists of two components:

- A *format reward function* (R_{fmt}), evaluated at every turn ($h \leq H$), which encourages structurally correct retrieval actions that adhere to Markdown formatting and contain properly identified code symbols.
- A *soundness reward function* (R_{snd}), evaluated at the final turn ($h = H$), which assesses the quality of the generated patch based on compilation success, functionality preservation, and vulnerability remediation. This directly incentivizes the agent to retrieve context that is most useful for generating a correct patch.

Training Objective. We then find a policy π that maximizes the average return using Group Relative Policy Optimization (GRPO) [40] adapted for our multi-turn setting. As summarized in Figure 58, the training

process involves generating multi-turn completions from the policy, using our reward model to assign per-turn rewards, and computing advantages to guide the policy update.

Let G be the number of groups and $\hat{A}_{t,g}$ is the advantage where:

$$\hat{A}_{t,g} = \frac{r_{t,g} - \text{mean}(\{r_{t,1}, r_{t,2}, \cdots, r_{t,G}\})}{\text{std}(\{r_{t,1}, r_{t,2}, \cdots, r_{t,G}\})}.$$

when $r_{t,g}$ is the reward of the g-th group at the t-th CPV. The conventional single-turn GRPO objective is shown as following:

$$\mathcal{L}_{\text{GRPO}}(\pi) = \frac{1}{T}\sum_{t=1}^{T}\frac{1}{G}\sum_{g=1}^{G}\left(\frac{\pi(a_{t,1} \mid x_{t,0})}{\pi_{\text{old}}(a_{t,1} \mid x_{t,0})}\hat{A}_{t,g} - \beta\, D_{\text{KL}}[\pi(\cdot \mid x_{t,1}) \,\|\, \pi_{\text{ref}}(\cdot \mid x_{t,0})]\right),$$

where $\pi(a \mid x)$ is the probability of an action a computed from from an action distribution $\pi(x)$, D_{KL} is the Kullback-Leibler (KL) divergence, β is a regularization parameter, π_{old} is a "frozen" π (*i.e.,* π without allowing update), and π_{ref} is an initial policy. Note that the KL divergence term is included to regularize the policy, preventing it from deviating too far from a frozen reference policy π_{ref}.

We extend the original GRPO objective to our multi-turn scenario by aggregating the loss across all turns $h = 1, \ldots, H$, as follows:

$$\mathcal{L}^{\text{multi}}_{\text{GRPO}}(\pi) = \frac{1}{T}\sum_{t=1}^{T}\sum_{h=1}^{H}\frac{1}{G}\sum_{g=1}^{G}\left(\frac{\pi(a_{t,h} \mid x_{t,h})}{\pi_{\text{old}}(a_{t,h} \mid x_{t,h})}\hat{A}_{t,g} - \beta\, D_{\text{KL}}[\pi(\cdot \mid x_{t,h}) \,\|\, \pi_{\text{ref}}(\cdot \mid x_{t,h})]\right).$$

This aggregated reward signal propagates through the entire multi-turn trajectory to optimize the policy. Note that the clipping term from the original GRPO formulation [40] is omitted here for notational simplicity.

Online Learning. As optimizing $\mathcal{L}_{\text{GRPO}}(\pi)$ over all T CPVs is memory-intensive, we update the policy in an online manner. In particular, each CPV is sequentially provided and then the learning agent concentrates on the single CPV repeatedly until a success plateau, then advances to the next CPV. This mirrors developer workflows, yielding sharp intra-case improvements while occasionally resetting performance on new cases.

8.3 Use Case: `babynginx/cpv-0`

We now present a representative case study of the learned policy with $H = 2$ on `babynginx/cpv-0`, which is part of our internal benchmark rather than an official AIxCC challenge. This CPV is motivated by CVE-2022-0995 (*i.e.,* an out-of-bounds access in the Linux kernel) and adapts its root cause to a simplified Nginx setting. Specifically, the bug arises in the parsing of a custom request header `X-Feature`, where a numeric value is converted using `ngx_atoi` and then used to set a bitmap index. The range check incorrectly uses `sizeof(bitmap) * BITS_PER_LONG`, which permits out-of-bounds access. The intended fix location corresponds to the first frame of the crash stack trace, making it tractable for automated patching. However, without retrieval of the relevant type definitions, an agent is unlikely to produce a correct patch. Moreover, the function contains a subtle trap: unlike the standard `atoi`, `ngx_atoi` maps both negative inputs and strings like "+1" to -1, which often misleads naïve patching strategies. This case thus provides a controlled yet realistic testbed to evaluate whether multi-turn retrieval can recover the missing type definitions and avoid misleading local fixes.

Turn 1. The process begins with an AddressSanitizer report, which clearly indicates the failure site:

Code Context $x_{t,1}$

```
==32==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x50200000b588
at pc 0x555909e7d03d bp 0x7ffc299a95c0 sp 0x7ffc299a95b8
READ of size 8 at 0x50200000b588 thread T0
SCARINESS: 33 (8-byte-read-heap-buffer-overflow-far-from-bounds)
    #0 0x555909e7d03c in ngx_http_process_custom_features /src/nginx/src/http/ngx_http_request.c:2006:5
    #1 0x555909e912af in ngx_http_process_request_headers /src/nginx/src/http/ngx_http_request.c:1507:23
    #2 0x555909e8e7d0 in ngx_http_process_request_line /src/nginx/src/http/ngx_http_request.c:1202:13
    (additional stack frames elided)
```

Retrieved Action $a_{t,1}$ with Reasoning

```
## Reasoning

The potential vulnerability is an out-of-bounds read error in the `ngx_http_process_custom_features` function.
(ommitted)

## Relevant Symbols

1. `ngx_http_process_custom_features`
(ommitted)
```

The log establishes the precise function requiring modification: `ngx_http_process_custom_features`. Without this initial localization, subsequent retrievals would lack direction, resulting in irrelevant or incomplete context.

Turn 2. At this stage, partial context has been assembled, but crucial information about valid feature indices is still missing.

Augmented Code Context for $x_{t,2}$

```
// nginx/src/http/ngx_http_request.c

static ngx_int_t
ngx_http_process_custom_features(ngx_http_request_t *r, ngx_table_elt_t *h,
    ngx_uint_t offset)
{
    // (ommitted)
}

// (additional code elided)
```

Retrieved Action $a_{t,2}$ with Reasoning

```
## Reasoning

The following symbols are relevant to the provided C code snippet: (ommitted)

## Relevant Symbols

1. ngx_http_features_t
(omitted)
```

Before Turn 3. The context builder at the phase of Turn 2 retrieves the header file, uncovering the missing structure.

Augmented Code Context for $x_{t,3}$

```
// nginx/src/http/ngx_http_request.h

typedef struct {
    DECLARE_BITMAP(bitmap, NGX_CUSTOM_FEATURE__NR);
} ngx_http_features_t;

// (additional code elided)
```

Importantly, this discovery is decisive: the macro `NGX_CUSTOM_FEATURE__NR` specifies the upper bound of valid feature indices. Our empirical evaluation showed that without retrieving this symbol, LLM-generated patches consistently failed, either uncompilable or logically unsound. Its retrieval demonstrates the necessity of multi-turn interaction—single-pass retrieval would likely terminate prematurely.

As $H = 2$, this augmented code context for $x_{t,3}$ is fed into the patching agent to generate a patched codebase as follows.

Generated Patch with Code Context $x_{t,3}$

```
--- a/nginx/src/http/ngx_http_request.c
+++ b/nginx/src/http/ngx_http_request.c
@@ -1995,7 +1995,7 @@

     features = r->headers_in.features;

-    if (custom_flag_value >= sizeof(features->bitmap) * BITS_PER_LONG) {
+    if (custom_flag_value < 0 || custom_flag_value >= NGX_CUSTOM_FEATURE__NR) {
         ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
                       "Unknown custom feature value");
         ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
```

The key improvement is replacing a brittle size-based bound with a semantically correct check against `NGX_CUSTOM_FEATURE__NR`. This ensures both memory safety and logical correctness.

The `babynginx/cpv-0` case exemplifies the role of multi-turn retrieval:

- **Fault localization to `ngx_http_process_custom_features`.** The initial retrieval step successfully identifies the precise function containing the vulnerability, providing essential direction for subsequent context gathering efforts.
- **Expansion of surrounding function and type context.** The agent systematically retrieves additional code artifacts surrounding the identified fault location, including related function definitions and structural type information necessary for comprehensive understanding.
- **Identification of the critical macro `NGX_CUSTOM_FEATURE__NR`.** Through multi-turn interaction, the retrieval process discovers the essential boundary definition that specifies valid feature indices, which proves crucial for generating semantically correct patches.
- **Patch generation leveraging the retrieved bound.** The final patching step utilizes the complete assembled context to generate a fix that replaces brittle size-based bounds with semantically appropriate checks against the retrieved macro definition.

This pipeline underscores that retrieval of *all* relevant symbols—not merely local function bodies—is essential for valid patching. The case validates our claim that multi-turn context learning is indispensable for automated vulnerability remediation.

8.4 Evaluation

8.4.1 Overview

To complement our evaluations with general-purpose LLMs, we developed a lightweight yet task-specialized retrieval agent tailored to AIxCC-style patching scenarios. The motivation was twofold: (i) to validate whether a parameter-efficient adaptation of an open-source base model can internalize retrieval heuristics from a limited number of CPV instances, and (ii) to examine training dynamics when reinforcement signals are derived from multi-turn context construction rather than direct patch supervision. This custom agent was fine-tuned on a curated set of CPVs under controlled compute resources, enabling us to probe the trade-offs between scalability, efficiency, and retrieval quality in a reproducible manner.

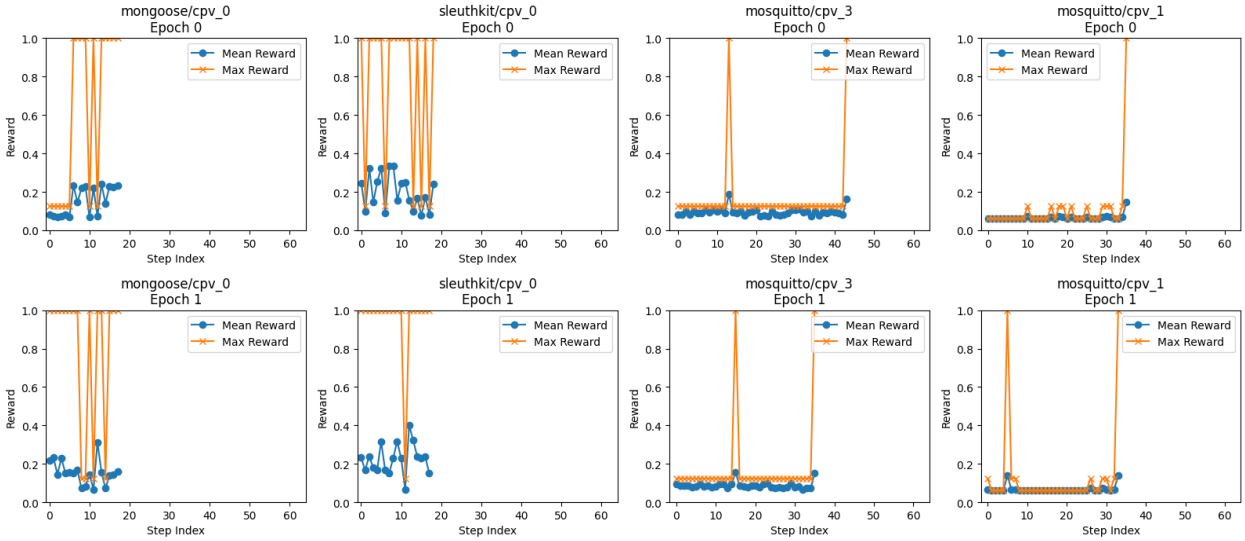


Figure 59: Training dynamics. Each subplot shows the reward trajectory for a specific project and CPV pair over two epochs. The blue line represents the mean reward across all agents in a group, while the orange line shows the maximum reward achieved by any agent in that group at each step.

8.4.2 Experimental Setup

We fine-tuned a custom model based on Meta's Llama-3.2-3B-Instruct. The model was trained on a proprietary dataset comprising 7 CPV instances. Training was conducted on a high-performance computing environment with 8× NVIDIA A100 (80GB) GPUs, an AMD EPYC 7513 32-Core processor, and 1.96TB RAM.

Our hyperparameter configuration was optimized for code retrieval tasks:

- **Context Window**: 8,192 tokens maximum prompt length with 10,240 tokens total sequence length to accommodate lengthy code contexts
- **Parameter-Efficient Fine-Tuning**: LoRA adaptation with the rank of 32 and alpha of 32, striking a balance between expressivity and computational efficiency
- **Optimization**: AdamW optimizer with a learning rate of 5×10^{-6} and constant scheduling across 3 training epochs
- **GRPO Group Size**: 12 parallel generations per step, enabling diverse exploration trajectories
- H: 4 retrieval steps per episode, aligned with the typical depth required for vulnerability localization
- T: 7 CPVs for training.
- **Downstream Evaluation**: GPT-4.1 as the patch generation model to assess retrieval quality

This configuration allowed us to efficiently fine-tune the retrieval model while maintaining reasonable computational requirements.

8.4.3 Training Dynamics

Figure 59 shows per-project reward trajectories over two epochs. Each subplot corresponds to a (project, CPV) pair; blue traces denote the mean reward across agents in a group, while orange traces denote the max reward attained by any agent in that group at a given step. We observe the following points.

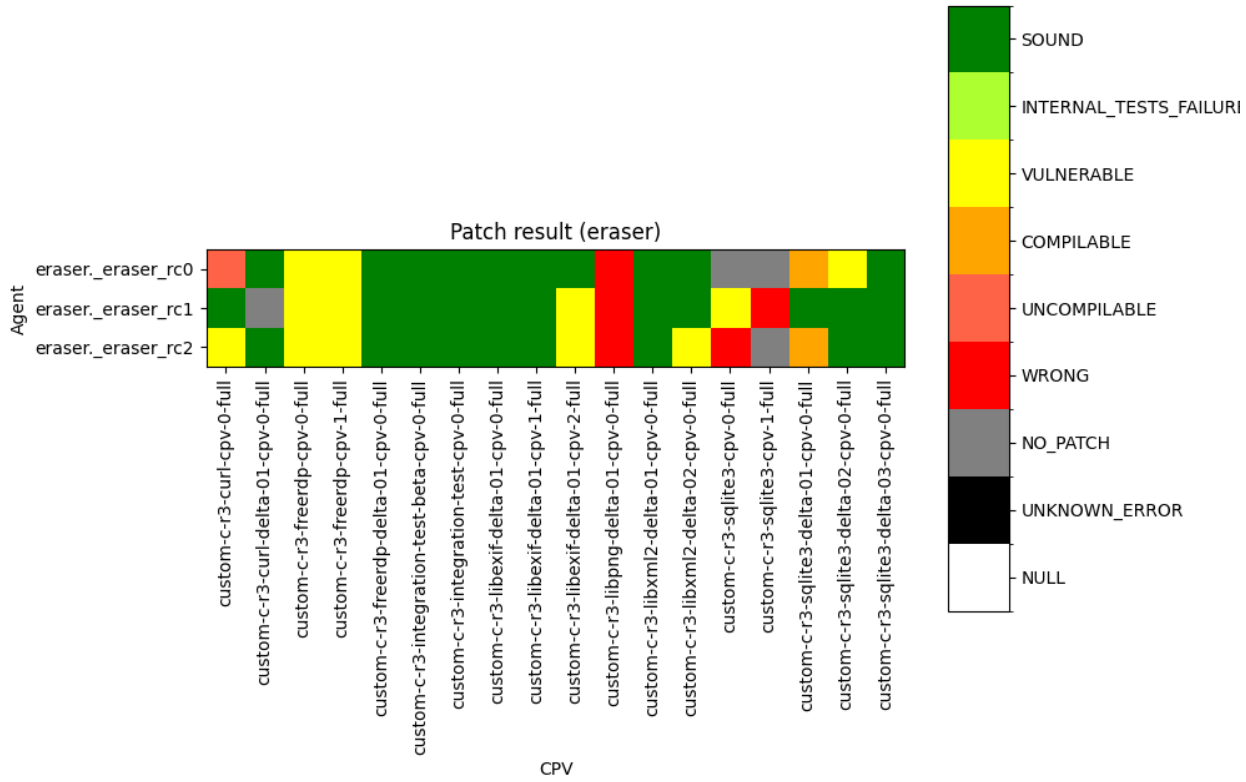


Figure 60: Patch outcomes across CPVs and agent checkpoints. Each cell represents the final patch outcome for a given CPV (columns) and agent checkpoint. Color encodes categorical evaluation results.

- **Early sparse successes.** In many panels the orange curve spikes to 1.0 well before the blue curve rises, indicating that at least one rollout quickly discovers a high-soundness trajectory while the cohort average remains low. This confirms the utility of group-based optimization: even when the policy is immature, diversity in rollouts can surface a correct retrieval path.
- **Staircase-like improvement.** The mean reward typically increases in discrete jumps rather than smoothly. This reflects the sequential nature of retrieval: once the policy learns to fetch a critical symbol or file, downstream steps become substantially easier, lifting the average.
- **Project-dependent difficulty.** Some projects (e.g., `mosquitto/cpv_2` and `swftools/cpv_0`) exhibit long flat regions with zero reward, suggesting that either (i) the necessary context is harder to localize, or (ii) the final patch is more brittle. Others (*e.g.*,`mongoose/cpv_0`) show frequent reward spikes, implying richer intermediate signals or easier-to-exploit structure.
- **Persistence matters.** The recurrence of max-reward spikes across epochs indicates that repeated attempts on the same instance in our online learning do not merely overfit one trajectory; instead, they stabilize the policy so that success can be rediscovered consistently.

8.4.4 Patch Outcomes

Figure 60 presents a heatmap of final patch outcomes across CPVs (columns) and agent checkpoints (rows: rc0, rc1, rc2). Color encodes categorical evaluation results (e.g., SOUND, VULNERABLE, WRONG, NO_PATCH). The following includes our findings.

- **High density of SOUND patches.** Green dominates across many CPVs, indicating that the retrieved contexts were sufficient for the downstream LLM to generate correct, test-passing fixes.
- **Failure mode taxonomy.**
  - **UNCOMPILABLE / WRONG (reds).** Typically arise when the retriever omits a dependency or surfaces stale code, causing syntactic or semantic mismatches.
  - **VULNERABLE (yellow-orange).** Patches compile but fail security or vulnerability checks, suggesting the retriever surfaced code relevant to functionality but not to the root cause of the vulnerability.
  - **NO_PATCH / UNKNOWN_ERROR (gray/black).** Reflect cases where the patching model either abstained or the pipeline failed externally (tooling/timeouts), highlighting engineering, not policy, limitations.
- **Checkpoint progression.** Later checkpoints (rc1, rc2) show fewer severe failures (e.g., UNCOMPILABLE, WRONG) and more SOUND or at least COMPILABLE outcomes, consistent with policy refinement over training.
- **Instance heterogeneity.** Certain CPVs remain problematic across all checkpoints (persistent reds/yellows), pointing to classes of bugs where our current retrieval actions (or reward shaping) are insufficient—prime targets for future ablations (e.g., deeper call graph exploration, semantic diffing).

8.5 Discussion

We discuss the contributions, key findings, and limitations of custom LLMs in ATLANTIS-Patching.

Contributions. This work introduces a specialized context retrieval agent addressing the challenges for LLM-driven automated security patch generation. Our key contributions include the following:

- **Efficacy in providing missing context for patching.** We share a key observation for successful patching, *i.e.,* providing missing code context (*e.g.,* definitions on undefined symbols within a given context) is crucial for sound patch generation by LLMs.
- **Multi-turn retrieval agent for patching.** We propose and learn a novel multi-turn retrieval agent that iteratively retrieves concise, targeted code context, enabling effective utilization of powerful yet context-limited commercial LLMs for patch generation.
- **Demonstration of effectiveness on real-world benchmarks.** We demonstrate the efficacy of learned multi-turn retrieval agent through successful participation in benchmarks such as the DARPA AIxCC competition.

Key Findings. Our findings emphasize code context retrieval as integral to successful security patch generation, advocating a broader perspective on prompt engineering wherein structured context provisioning significantly enhances model outcomes.

The following summarizes our key findings in developing custom LLMs for patching.

- **With proper coding context, commercial LLMs are able to generate secure patches.** When provided with comprehensive and relevant code context through our multi-turn retrieval approach, state-of-the-art commercial language models demonstrate strong capability in producing functionally correct and security-compliant patches for complex vulnerabilities.

- **A pretrained model has fair context selection performance.** Base models such as Meta's Llama-3.2-3B-Instruct exhibit reasonable baseline performance in identifying and retrieving relevant code artifacts, providing a solid foundation for further specialization through reinforcement learning techniques.
- **RL fine-tuning improves context selection performance.** Reinforcement learning optimization using our multi-turn GRPO framework significantly enhances the model's ability to select optimal code contexts, leading to measurable improvements in downstream patch generation quality and success rates.
- **Forgetting in learning hinders RL advancement.** The online learning approach introduces catastrophic forgetting challenges, where knowledge acquired from earlier challenge project vulnerabilities degrades when adapting to new instances, limiting the overall effectiveness of the training process.

Limitations. Our approach has several practical limitations.

- **Language scope.** The current agent learning code is specialized for C codebases and has not been extended to other programming languages such as Java.
- **Restricted retrieval scope.** The retrieval process focuses only on function definitions, method definitions, and type definitions, potentially overlooking other useful artifacts such as build scripts or configuration files.
- **Tooling overhead.** Our use of LSP/parser-based tools required manual, per-rule adapter development, which limits scalability; broader, language-agnostic tools would improve generality.
- **Forgetting risk.** The online learning approach can lead to catastrophic forgetting, *i.e.,* a retrieval policy learned on earlier CPVs is degraded when adapting to new or more CPVs.
- **Competition constraints.** Late changes in competition rules limited available custom model learning time and reduced the breadth of the model validation.

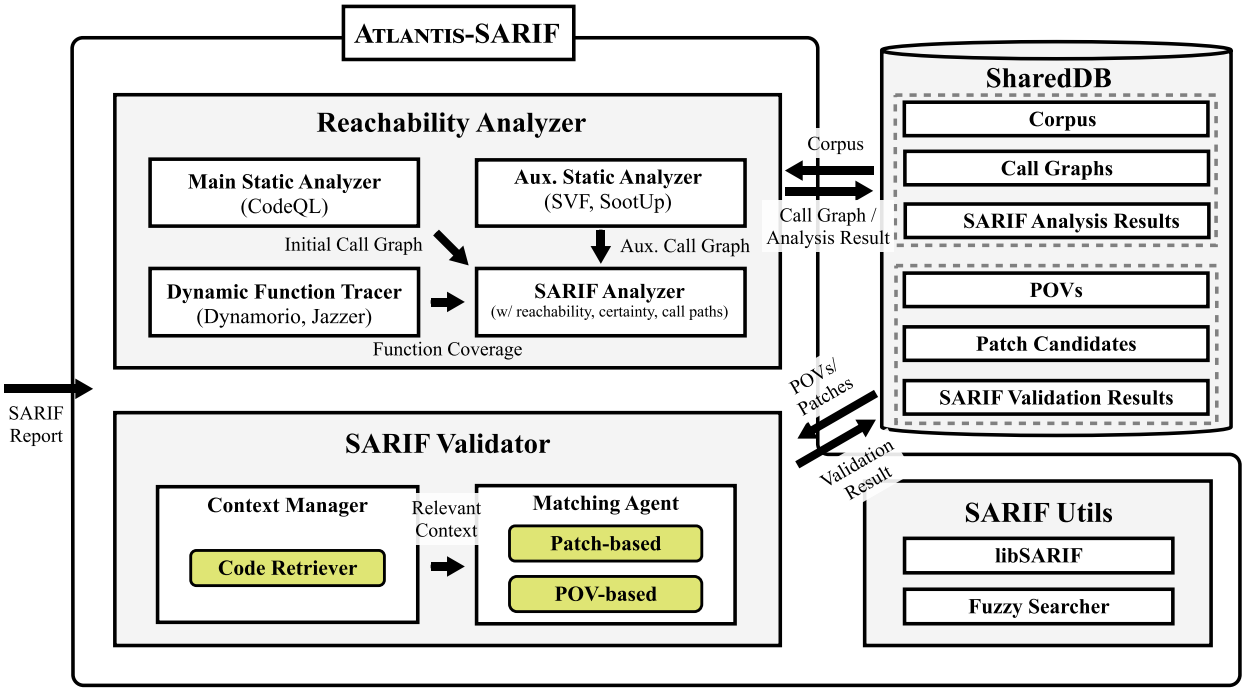


Figure 61: SARIF Validation Architecture. LLM-based modules are highlighted with green boxes. The system processes incoming SARIF reports through reachability analysis and LLM-based validation. Results are stored in the SharedDB and accessed through our custom API interface.

9 ATLANTIS-SARIF

SARIF (Static Analysis Results Interchange Format) is a standardized, JSON-based format for reporting static analysis findings. It enables consistent and automated integration across tools. In ATLANTIS, SARIF reports are provided by competition systems as broadcasts, describing potential vulnerabilities, and accurate assessment is critical for scoring. However, the reported code locations and vulnerability descriptions are not always accurate. This limitation necessitates a Reachability Analysis module (§9.1), which verifies whether a vulnerability is reachable from harness's entry points. Moreover, SARIF descriptions are often ambiguous or incomplete. To address this, a SARIF Matching module aligns external reports with internally discovered bugs using heuristics and LLM-based analysis (§9.2). These modules together enhance the reliability, scoring accuracy, and practical utility of SARIF in automated vulnerability analysis.

Figure 61 presents an overview of the SARIF module. Broadcasted SARIF reports first pass through the *Reachability Analyzer*, which merges a primary static call graph (via `CodeQL`), an auxiliary static graph (via `SVF` or `SootUp`), and dynamic function-trace coverage (using `DynamoRIO` or `Jazzer`) into a unified call graph. This combined graph is stored in the shared database, alongside SARIF analysis results. The *SARIF Validator* then analyzes and processes incoming SARIF reports. It includes a *Context Manager* with a file retriever that extracts relevant code snippets, and a *Matching Agent* that applies both patch-based and PoV-based logic to assess report correctness. The *Matching Agent* uses an LLM to make final validation decisions. The *SARIF Validator* begins by checking whether the target SARIF report corresponds to any stored artifacts (PoVs or patches). In the absence of a match, it waits for new artifacts to be added and repeats the matching procedure accordingly. Once the report aligns with at least one artifact, it is classified as "Correct". Validation results are written back to the database and made accessible through our custom designed API interface. The *SARIF Utils* component contains helper libraries. This includes `libSarif` for programmatic call graph queries and a fuzzy searcher to handle edge cases, such as amalgamated codebases.

SARIF Scoring Rules. When a SARIF broadcast is received, a CRS may submit a SARIF Assessment. The assessment is "correct" if the issue exists at the specified file, line, and CWE/bug type, or "incorrect" otherwise. Each assessment includes the SARIF ID, a binary verdict, and a justification. Correct assessments earn $1 \times \tau_{\text{assessment}}$ points, where $\tau_{\text{assessment}}$ decays from 1 to 0.5 over time. Incorrect or outdated assessments score zero and reduce accuracy. Only the last assessment per broadcast is scored; earlier ones count as inaccuracies.

CRS also should bundle SARIF broadcasts with PoVs and patches. A bundle scores only if it correctly links the broadcast UUID to a PoV or patch for the same vulnerability. This yields up to 3 extra points, while incorrect links are penalized.

Our Approach: Conservative Validation. SARIF broadcasts frequently contain false positives, and wrong assessments are penalized while also reducing accuracy. To maximize scoring reliability, ATLANTIS-SARIF adopts a *conservative* policy: ATLANTIS determines a SARIF report as *Correct* only when there is concrete evidence from either (1) a PoV that reaches the SARIF location, or (2) a patch whose changes are logically consistent with the SARIF report. This choice prioritizes precision over recall, protecting the accuracy multiplier and avoiding penalties from spurious submissions.

Because validation hinges on concrete evidence from PoVs or patches, the primary objective is to discover a PoV at the code location of the SARIF report, leveraging the targeted bug-finding features of our CRSes. In this process, to prevent unnecessary resource consumption, an initial reachability analysis filters out locations that are not exploitable. When a relevant PoV or patch is identified, the LLM-based validator integrates crash logs, patch diffs, and code context to validate the SARIF report. Importantly, this validator never operates without such explicit evidence, thereby preserving the conservative strategy that disallows *Correct* matches without a PoV or patch.

9.1 Reachability Analysis Module

Reachability analysis determines whether a specific code location can be exercised from one or more entry points (*e.g.,* harnesses's main function). By constructing and traversing a call graph, we identify all invocation paths that may lead to a reported vulnerability site. This analysis is essential for the ATLANTIS-SARIF as static analysis tools often report issues in dead code or in paths unreachable from any harness. Filtering out such false positives saves time and prevents wasted effort during dynamic validation and patch generation.

At the core of our reachability module is a unified call graph construction pipeline. It aggregates results from multiple analyzers: `CodeQL` for primary static analysis, `SVF` for Andersen-style points-to analysis in C, and `SootUp` for Class Hierarchy Analysis (CHA) and Rapid Type Analysis (RTA) in Java. The module supports both static-only graphs and dynamic updates that augment this graph with coverage information obtained from function-trace logs collected during fuzzing by other CRS components. From this combined graph, we generate for each SARIF report a *reachable harness list* and a *partial call graph* connecting harness entry points to the vulnerability location.

To ensure robustness on large and complex codebases, we employ a tiered fallback strategy: (1) whole-program points-to analysis, (2) restricted extraction of only harness-reachable nodes, and (3) direct-edge-only extraction under memory or time constraints. The fallback strategy guaranteed consistent call graph generation for multiple OSS-Fuzz projects and thereby enhanced the overall stability of ATLANTIS-SARIF.

Also, we assign each result one of three reachability confidence levels: *Certain*, where at least one path contains only *strong* edges; *Possible*, where all paths include one or more *weak* edges; and *Unlikely*, where no path is found. Here, *weak* and *strong* edges are determined by the reliability of the method used to infer the edge. For example, edges from the results of function tracer are treated as *strong*, since they reflect actual execution. In static call graphs, direct calls are treated as *strong* edges, but edges inferred from pointer analysis are treated as *weak* edges. This *weak*/*strong* edge labeling and three-level confidence schem reflects our trust in the analysis and guides downstream filtering and prioritization. Other CRS components can apply their own policies based on these confidence levels.

To maintain precision, we address several edge cases. For amalgamated codebases (*e.g.,* `SQLite3`), where source lines shift after build, we apply fuzzy similarity matching to locate the correct function. For projects with identical harness paths, such as multiple `curl` variants, we disambiguate by inspecting each harness's linking targets. The module emits reachability annotations with every SARIF result. These annotations include harness names, confidence levels, and the filtered subgraph. Downstream components including LLM-based matchers and patch generators consume this information directly. The module also integrates with the CP-MANAGER interface. This allows the CRS to retrieve call graph fragments on demand, supporting seamless end-to-end submissions and continuous integration pipelines.

libSARIF: Interface for Other CRSs. `libSarif` is a standalone library that exposes ATLANTIS-SARIF's call graph and reachability functionality as a reusable API. It parses the merged static-and-dynamic graph format produced by our module and supports several core queries. The library provides functions to determine whether a given source-line location is reachable, to enumerate all call paths from harness entry points to that location, and to compute the shortest such path. By abstracting call graph logic, libSarif enables other CRS components, such as directed fuzzer of ATLANTIS-C, to query and traverse exactly the subgraph they need, without re-implementing underlying analysis logic.

9.2 SARIF Validity Check

In the ATLANTIS-SARIF workflow, each incoming SARIF report must be matched to an internally discovered artifacts (*i.e.,* either a PoV or a patch) and then validated as correct or incorrect. This approach is consistent with our conservative policy aimed at reducing incorrect assessments. Reports often contain only file paths, line ranges, natural-language messages, and CWE identifiers. These fields may be incomplete or misleading. Simplistic matching based on filename and line-number proximity can produce false positives, especially when multiple vulnerabilities affect the same code region. Syntactic matching alone fails to account for semantic context, such as crash stack traces or patch logic. Validation must therefore integrate structural, dynamic, and logical evidence to ensure that only true positives are accepted. This preserves scoring accuracy and prevents wasted effort in exploitation or remediation.

Our initial approach treated matching as a statistical correlation problem. For SARIF–PoV pairs, we computed the fraction of SARIF locations exercised by the PoV's execution trace. For SARIF–patch pairs, we measured the overlap between patched lines and SARIF locations. A high overlap ratio suggested a match; a low ratio indicated a mismatch. Although fast and model-free, this approach struggled with common utility lines which may appear across unrelated vulnerabilities and with small patches or deep call graph paths, which reduce apparent overlap. Since the depth of crash stack traces and the count of patched lines differ widely across projects, we must tune decision thresholds in a project-specific manner. Edge cases, such as multi-harness scenarios, also remained problematic.

To overcome these limitations, we deployed an LLM-based agent as the final matcher and validator. We supply the full SARIF report, crash logs (for PoVs), and patch diffs when available. Relevant source context is also provided; we use a LLM-based file retriever to pull in code snippets, referenced files, and dependent functions. We instruct the model to: (1) identify root causes and trigger conditions, (2) assess logical correlation, and (3) decide among *Matched*, *Not Matched*, or *Uncertain*. We only confirm a SARIF report as "Correct" when the model's decision is *Matched*, and submit our assessment accordingly. By leveraging the reasoning ability of the LLM instead of raw line counts, we achieve higher precision.

We standardize prompts and limit input to relevant context snippets, then validate performance on benchmark datasets to calibrate model confidence. The LLM's decision is supplemented with quantitative metadata, including reachability confidence from the call graph module. We also log the model's rationale for auditability. To control cost, we selectively route cases to the LLM particularly those with borderline overlap or high-impact vulnerabilities and throttle invocations when necessary. Integration into ATLANTIS's CP-MANAGER interface allows downstream CRS components to programmatically request match or validation results.

	C benchmark			Java benchmark		
	CodeQL (basic)	**SVF (Ander)**	**CodeQL + SVF**	**CodeQL (basic)**	**SootUp (CHA)**	**CodeQL + SootUp**
Total	28/31	23/31	28/31	20/57	44/57	45/57
Acc	90%	74%	90%	35%	77%	79%

Table 21: Reachability analysis success rates for C and Java benchmarks under different call graph configurations.

	Crash-only		Patch-only		Crash+Patch	
Decision	**Matched**	**Not Matched**	**Matched**	**Not Matched**	**Matched**	**Not Matched**
Matched	50	5	51	2	53	4
Uncertain	3	20	7	6	0	0
Not Matched	5	209	0	226	5	230

Table 22: Confusion matrices for SARIF matching under three scenarios: **Crash-only**, **Patch-only**, and **Crash+Patch**.

9.3 Evaluation

This section evaluates two core components of the ATLANTIS-SARIF workflow: the Reachability Analysis Module, which determines whether reported code locations are actually reachable, and the SARIF Validity Check, which verifies whether each SARIF report corresponds to a true vulnerability. Both components are evaluated on custom benchmark datasets, and we analyze their contribution to matching accuracy and overall scoring precision.

Reachability Analysis Module. Table 21 compares reachability analysis success rates across our C and Java benchmarks under different call graph configurations. We converted the PoVs of our benchmarks into SARIF reports and examined whether the reachability analysis classified them as reachable. In the C suite, `CodeQL` alone (basic forward analysis) correctly resolves 28 of 31 true positives (90%). `SVF`, using Andersen-style points-to analysis, reaches only 23 of 31 (74%) when used in isolation. Merging `CodeQL` and `SVF` graphs yields the same 28/31 (90%) as `CodeQL` alone, suggesting that `SVF`'s additional points-to information does not improve coverage beyond `CodeQL`'s basic analysis in this benchmark.

On the Java side, basic `CodeQL` analysis succeeds on only 20 of 57 cases (35%), due to the complexity of virtual dispatch and class loading. Integrating `SootUp`'s Class Hierarchy Analysis (CHA) raises the true positive rate to 44/57 (77%). Combining `CodeQL` with `SootUp` edges provides a modest further gain to 45/57 (79%). These results show that auxiliary analyses, especially for Java, are essential to handle dynamic dispatch. An ensemble of static tools yields the most robust reachability coverage in large, real-world codebases. Note that this evaluation did not employ call graph refinement based on dynamic function traces. Therefore, performance may improve in the actual competition environment.

SARIF Validity Check. Our matching accuracy across three scenarios (*Crash-only*, *Patch-only*, and *Crash with Patch*) is summarized in Table 22. We manually constructed a benchmark of SARIF reports, carefully labeling each as correctly or incorrectly matched to establish ground truth for evaluation. In the *Crash-only* case, we correctly matched 50 reports and incorrectly matched 5. We deferred 23 reports including 3 true positives labeled as "Uncertain" and 20 cases correctly rejected as "Uncertain" while missing 5 true positives (false negatives). This yields robust but imperfect coverage. The *Patch-only* scenario shows similar trends: 51 true positives, 2 false positives, 7 uncertain cases, and no false negatives among 226 negatives. Combining crash and patch evidence further improves precision. We correctly matched 53 reports, with only 4 false positives and no uncertain decisions. However, 5 true positives remained unmatched. Overall, the LLM-based matcher achieves high true-positive rates and low false-positive rates, especially when both crash and patch inputs are present.

10 Benchmark

ATLANTIS integrates several modules for automatic bug findings and automatic patch generation as described in §3.1. To systematically evaluate their improvement, robustness, capability, and limitation, we developed 56 C/C++ benchmarks with 130 vulnerabilities and 40 Java benchmarks with 152 vulnerabilities. In addition to target repositories, the benchmarks also provide descriptions, POVs, and correct patches about vulnerabilities: https://github.com/Team-Atlanta/aixcc-afc-benchmark.

10.1 Component-Specific Benchmark Design

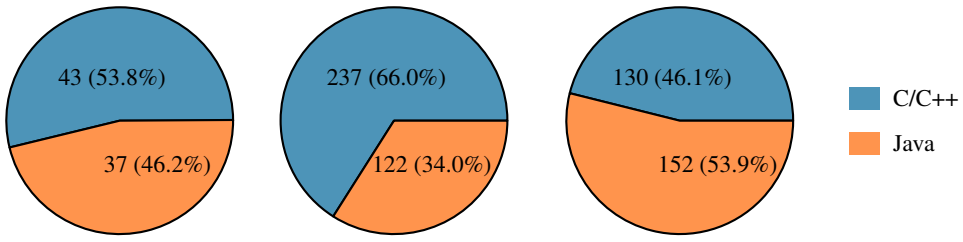
The benchmark is designed to evaluate both the overall robustness of ATLANTIS and the capability of each module. To assess robustness, we introduced examples targeting edge cases such as excessive standard output, file descriptor leaks, and unprintable byte sequences, all of which are intended to stress or disrupt the system. To evaluate capability, we constructed examples that may appear trivial for one component but prove infeasible for another, with some components even failing entirely on specific classes of inputs. Furthermore, we carefully designed multi-level benchmarks by analyzing coverage gaps observed when running ATLANTIS on real-world projects and deliberately inserting vulnerabilities into previously unreachable code regions.

For Concolic Executor. The concolic executor symbolically executes program inputs to solve branch conditions, thereby enabling exploration of branches that are often difficult for traditional fuzzers. To assess its effectiveness, we developed benchmarks involving complex arithmetic operations, which are typically challenging for fuzzing alone. However, the concolic executor also suffers from well-known limitations, such as path explosion and limited robustness. To stress test its capabilities, we included benchmarks with special instructions (e.g., AVX) and scenarios where vulnerabilities are located beyond deep loop nests.

For Directed Fuzzer. The directed fuzzer is particularly effective when the suspected location of a vulnerability is known and the search space of program is large. To evaluate its capability, we constructed benchmarks having diverse call depths and control-flow structures, with vulnerabilities intentionally placed deep within call chains. However, when the codebase is very large, locating the precise target location can itself become difficult, which undermines reliable evaluation. To address this, we provide the target location in delta mode by supplying diffs, enabling evaluation even when the vulnerable location is hard to identify.

For LLM-powered components. ATLANTIS has many LLM-powered components for not only automatic bug findings but also automatic patch generation. While LLMs have shown strong empirical performance, several limitations must be addressed to ensure fair and robust evaluation. First, evaluation must account for potential data leakage from training. If a vulnerability or challenge was present in the LLM's training corpus, its performance may not reflect true generalization. To mitigate this, our benchmark suite includes newly discovered and synthetic vulnerabilities unlikely to appear in training data. Second, LLMs are constrained by fixed context windows. Designing benchmarks that require effective context reduction or summarization is essential to assess how well LLM-based components handle long-range dependencies. Finally, we include challenges that test robustness of LLMs against misleading or incomplete natural language descriptions. In real-world scenarios, comments and documentation may not accurately describe program behavior, and LLMs must avoid overreliance on such information to make sound predictions or patches.

For Patching Agents. The difficulty of bug detection does not necessarily correlate with the difficulty of patch generation. For example, stack buffer overflows may be challenging to trigger due to input constraints, yet the corresponding patch such as bounds checking, is often straightforward. More compelling patching scenarios arise when the crash log provides limited diagnostic value, requiring extensive static or dynamic analysis to locate the root cause and generate a correct patch without breaking existing functionality. To capture these cases, we included benchmarks where the bug may be easily discovered by a simple fuzzer, but repairing it demands non-trivial reasoning and code understanding.



(a) Repositories (b) Fuzzing Harnesses (c) Vulnerabilities

Figure 62: Distribution of benchmark components across C/C++ and Java

10.2 Benchmark Statistics

Each benchmark comprises a repository, and a repository may include multiple fuzzing harnesses that exercise the codebase and its injected vulnerabilities. Notably, most repositories in our benchmarks are based on well-known open-source projects rather than small synthetic examples. Figure 62 summarizes the benchmark statistics across C/C++ and Java. Out of the 80 repositories, 43 (53.8%) are implemented in C/C++ and 37 (46.2%) in Java. Fuzzing harnesses are more prevalent in the C/C++ set, with 237 harnesses (66.0%) versus 122 (34.0%) for Java. For vulnerabilities, we included 130 cases in C/C++ (46.1%) and 152 in Java (53.9%), for a total of 282 vulnerabilities. Overall, the benchmarks capture a balanced representation across the two languages while ensuring coverage in all three categories.

C/C++ Benchmarks. Following the OSS-Fuzz standard, our C/C++ benchmarks include vulnerabilities detected by ASan, MSan, and UBSan. In total, we constructed 56 C/C++ benchmarks comprising 130 vulnerabilities, 2.3 vulnerabilities per benchmark on average. Of these, 122 were detected by ASan, 4 by UBSan, and 4 by MSan. Table 23 lists the vulnerability types included in our suite, with brief descriptions and occurrence counts; the most frequent types correspond to those commonly observed in real-world software, reflecting our combination of real-world vulnerability backports and synthetic examples. In addition, our benchmarks are large enough to evaluate the capability of ATLANTIS. Each of them contains 1,715 source files on average while the mean lines of code per benchmark is 454K.

Java Benchmarks. OSS-Fuzz uses Jazzer for fuzzing Java projects and employs Jazzer's sanitizers to detect Java-specific vulnerabilities such as command injection and file-path traversal. It also leverages ASan to capture memory corruption bugs in native code invoked via the Java Native Interface. Based on this setup, we constructed 40 Java benchmarks comprising 152 vulnerabilities, 3.8 vulnerabilities per benchmark. Table 24 summarizes the vulnerability types included in the suite. In addition, we made our benchmarks large enough to test capability of ATLANTIS. Each of them contains 2,473 source files and 295K lines of code on average.

10.3 Constructing Realistic and Diverse Benchmark

Our benchmark combines backported vulnerabilities, newly added projects, and synthetic cases to balance realism and diversity, avoiding overfitting to narrow dataset. We backport CVEs and OSS-Fuzz vulnerabilities into their original projects, grounding the benchmark in real bug distributions. Even if patches appear in an LLM's pretraining data, solving tasks remains challenging since patching systems must correctly adapt and reapply fixes in new contexts. To broaden coverage, we extend beyond OSS-Fuzz by adding projects from new domains and injecting synthetic vulnerabilities in challenging locations to reach. By maintaining 60–70% backported vulnerabilities with synthetic ones, the benchmark ensures fairness against pretraining leakage while still providing fresh challenges for automated vulnerability discovery and repair.

Vulnerability Type	Description	Count (%)
ABRT	Program aborted, typically due to an assertion failure or fatal runtime error.	1 (0.8%)
DoubleFree	Attempting to deallocate the same memory region more than once.	4 (3.1%)
DynamicStackBufferOverflow	Overflow on a dynamically allocated stack buffer, such as one created with `alloca`.	2 (1.5%)
FPE	Floating-point exception due to invalid arithmetic operation, such as division by zero.	3 (2.3%)
GlobalBufferOverflow	Out-of-bounds access to a global or static buffer.	5 (3.8%)
HeapBufferOverflow	Writing or reading beyond the bounds of a heap-allocated buffer.	46 (35.1%)
HeapUseAfterFree	Accessing heap memory after it has been deallocated.	9 (6.9%)
ILL	Illegal instruction encountered, possibly due to corrupt or invalid code execution.	2 (1.5%)
IntraObjectOverflow	Overflow within fields of a single object.	1 (0.8%)
MemcpyParamOverlap	Source and destination buffers in `memcpy` overlap.	1 (0.8%)
MemoryLeak	Allocated memory is not freed.	1 (0.8%)
NegativeSizeParam	A function received a negative value for a size parameter.	1 (0.8%)
OutOfMemory	Excessive memory usage beyond the allowed threshold.	1 (0.8%)
SEGV	Segmentation fault due to invalid memory access.	31 (23.7%)
SignedIntegerNegation	Negating the most negative 64-bit signed integer.	1 (0.8%)
SignedIntegerOverflow	Signed integer exceeds its maximum or minimum representable value.	2 (1.5%)
StackBufferOverflow	Buffer overflow in stack memory.	11 (8.4%)
StackBufferUnderflow	Accessing memory before the start of a buffer on the stack.	1 (0.8%)
StackUseAfterReturn	Accessing stack memory from a function that has already returned.	1 (0.8%)
StackUseAfterScope	Accessing stack memory outside its declared lifetime.	1 (0.8%)
Timeout	Execution exceeded time limits.	3 (2.3%)
UnknownCrash	A crash with an undetermined cause.	1 (0.8%)
UseOfUninitializedValue	Use of a variable before it has been initialized.	1 (0.8%)

Table 23: C/C++ vulnerability types with counts and descriptions

Vulnerability Type	Description	Count (%)
ArbitraryLibraryLoad	Loading attacker-specified native libraries via `System.load()` or similar APIs.	2 (1.3%)
ArrayIndexOutOfBounds	Accessing an array index outside its valid range.	1 (0.7%)
ArraySizeLimitExceeded	Attempting to allocate an array larger than the JVM allows.	1 (0.7%)
CommandInjection	Unsanitized input is used in command execution.	26 (17.1%)
HeapBufferOverflow	Writing or reading beyond the bounds of a heap-allocated buffer.	1 (0.7%)
IllegalState	Inconsistency in decompression or data processing logic.	1 (0.7%)
IndexOutOfBounds	Attempt to access a list or string with an invalid index.	3 (2.0%)
InvalidFree	Simulated in Java via incorrect use of custom memory management or JNI.	1 (0.7%)
LDAPInjection	Injection into LDAP queries.	2 (1.3%)
OutOfMemory	Excessive memory usage beyond the allowed threshold.	9 (5.9%)
PathTraversal	Manipulating file paths to access unintended files.	9 (5.9%)
RegexInjection	Malicious input causes catastrophic backtracking in regex evaluation.	5 (3.3%)
RemoteCodeExecution	Input leads to arbitrary code execution via unsafe reflection or dynamic evaluation.	52 (34.2%)
RemoteJNDILookup	Remote lookup via JNDI for arbitrary object deserialization.	2 (1.3%)
ScriptEngineInjection	Unsafe evaluation of user-controlled input in scripting engines.	2 (1.3%)
ServerSideRequestForgery	The application makes unauthorized HTTP requests due to attacker-controlled input.	17 (11.2%)
SQLInjection	Manipulated input in SQL queries.	2 (1.3%)
StackOverflow	Deep or infinite recursion exceeding call stack limits.	8 (5.3%)
Timeout	Execution exceeded time limits.	2 (1.3%)
XPathInjection	Injection of input into XPath queries.	6 (3.9%)

Table 24: Java vulnerability types with counts and descriptions

11 0-day Bugs

11.1 SQLite3: Off-by-one Read

We discovered a 0-day vulnerability in SQLite3's FTS5 (Full-Text Search 5) module, specifically within the trigram tokenizer implementation. The trigram tokenizer breaks text into overlapping three-character sequences for substring matching in full-text searches.

The vulnerability occurs when users create a virtual table with incomplete tokenizer configuration. While FTS5 expects tokenizer options as key-value pairs (e.g., `case_sensitive 1`), it fails to validate argument counts when a key lacks its corresponding value before accessing array elements.

This results in an off-by-one read beyond the argument array bounds (`azArg[i+1]` when `i+1 >= nArg`). While this could potentially lead to information disclosure in other contexts, SQLite3's defensive programming practices mitigate the impact. The library uses `sqlite3_malloc` (which zero-initializes memory) for the argument array allocation, ensuring that out-of-bounds reads return NULL rather than arbitrary memory content. Consequently, the vulnerability manifests as a NULL pointer dereference when the code attempts to examine the non-existent value.

Root Cause Analysis. The vulnerability stems from insufficient bounds checking in the `fts5TriCreate` function. The code iterates through tokenizer arguments in pairs (key-value), incrementing the loop counter by 2:

```
 static int fts5TriCreate(
   void *pUnused,
   const char **azArg,
   int nArg,
   Fts5Tokenizer **ppOut
 ){
   int rc = SQLITE_OK;
   TrigramTokenizer *pNew = sqlite3_malloc(sizeof(*pNew));
   UNUSED_PARAM(pUnused);
   if( pNew==0 ){
     rc = SQLITE_NOMEM;
   }else{
     int i;
     pNew->bFold = 1;
     pNew->iFoldParam = 0;
     for(i=0; rc==SQLITE_OK && i<nArg; i+=2){
      // NOTE. off-by-one
★     const char *zArg = azArg[i+1];
      if( 0==sqlite3_stricmp(azArg[i], "case_sensitive") ){
        // NOTE. null dereference
★       if( (zArg[0]!='0' && zArg[0]!='1') || zArg[1] ){
          rc = SQLITE_ERROR;
        }else{
          pNew->bFold = (zArg[0]=='0');
        }
      }
   ...
}
```

At line 18, the code accesses `azArg[i+1]` without verifying that `i+1 < nArg`. When users provide an odd number of arguments (e.g., only a key without a value), this results in reading beyond the argument array bounds. The subsequent dereference at line 21 (`zArg[0]`) triggers a NULL pointer dereference, causing the program to crash.

PoC. The following SQL statements demonstrate the vulnerability:

```
CREATE VIRTUAL TABLE t
  USING fts5(s, tokenize='trigram case_sensitive');
CREATE VIRTUAL TABLE t
  USING fts5(s, tokenize='trigram remove_diacritics');
```

Both statements omit the required value for their respective options (`case_sensitive` and `remove_diacritics`), triggering the off-by-one read and subsequent crash. While this vulnerability might appear straightforward in hindsight, it represents a genuine security issue in one of the world's most widely deployed database engines. Our team was the only one to identify this 0-day vulnerability during the competition, demonstrating the effectiveness of our automated bug-finding approach—particularly notable given that we had no prior knowledge that SQLite3 was the target project.

Patch Analysis. ATLANTIS generated a patch addressing the vulnerability with proper bounds checking:

```
diff --git a/ext/fts5/fts5_tokenize.c b/ext/fts5/fts5_tokenize.c
index f12056170..552f14be9 100644
--- a/ext/fts5/fts5_tokenize.c
+++ b/ext/fts5/fts5_tokenize.c
@@ -1299,8 +1299,10 @@ static int fts5TriCreate(
     pNew->bFold = 1;
     pNew->iFoldParam = 0;
     for(i=0; rc==SQLITE_OK && i<nArg; i+=2){
-      const char *zArg = azArg[i+1];
-      if( 0==sqlite3_stricmp(azArg[i], "case_sensitive") ){
+      const char *zArg = (i+1 < nArg) ? azArg[i+1] : NULL;
+      if (zArg == NULL) {
+        rc = SQLITE_ERROR;
+      } else if( 0==sqlite3_stricmp(azArg[i], "case_sensitive") ){
         if( (zArg[0]!='0' && zArg[0]!='1') || zArg[1] ){
           rc = SQLITE_ERROR;
         }else{
```

The patch implements a two-layer defense:

- **Bounds checking**: Before accessing `azArg[i+1]`, the patch verifies that `i+1 < nArg` using a ternary operator, preventing the off-by-one read.
- **NULL validation**: If the bounds check fails or if the argument is missing, `zArg` is set to NULL, and an explicit check returns `SQLITE_ERROR` before any dereference occurs.

This fix ensures that incomplete tokenizer configurations are properly rejected with an error rather than causing a crash. The patch was generated in approximately 15 minutes, including the entire build, patch generation, iterative refinement, and correctness validation process.

11.2 SQLITE3: Use-after-free

We identified another 0-day vulnerability in SQLite3's LSM1 (Log-Structured Merge-tree) extension, specifically a UAF bug affecting virtual table cursor management. This bug manifests when executing queries with multiple equality conditions (e.g., `WHERE key IN ('key_0','key_1')`) against an LSM1 virtual table.

The vulnerability arises from improper cache invalidation in the LSM1 cursor implementation. When processing unique key lookups, the cursor caches decoded column data but fails to reset this cache between iterations, leading to use-after-free conditions when the underlying memory is reallocated.

Root Cause Analysis. The bug occurs due to an optimization in the `lsm1Next` function that only clears cached data when the result is non-unique. `bUnique` indicates true if no more than one row of output.

```c
// ext/lsm1/lsm_vtab.c
static int lsm1Next(sqlite3_vtab_cursor *cur){
  lsm1_cursor *pCur = (lsm1_cursor*)cur;
  int rc = LSM_OK;
  if( pCur->bUnique ){
    pCur->atEof = 1;
  }else{
    /* ...code continues... */
*   pCur->zData = 0; // BUG: Only when NOT bUnique
  }
  return rc==LSM_OK ? SQLITE_OK : SQLITE_ERROR;
}
```

The `pCur->zData` pointer caches decoded value data from the current row. When `bUnique` is set, the cache is not cleared in `lsm1Next`. However, subsequent queries reuse the same virtual table cursor, and if the new value requires more memory, `realloc` frees the old buffer while `pCur->zData` still points to it.

PoC. The following SQL demonstrates the vulnerability:

```sql
.load "./lsm.so"

CREATE VIRTUAL TABLE test_1337 \
  USING lsm1 ('test_1337.lsm', key, TEXT, value TEXT);

INSERT INTO test_1337 VALUES ('key_0', 'value_0');
INSERT INTO test_1337 VALUES ('key_1', 'value_1A');

SELECT value FROM test_1337 WHERE key IN ('key_0','key_1');
```

The first `SELECT` for `key_0` sets `bUnique=1` and caches the value data. The second lookup for `key_1` triggers a reallocation due to the longer value, but `lsm1DecodeValues` uses the stale `pCur->zData` pointer, resulting in a use-after-free read.

Security Impact. This vulnerability enables information disclosure through controlled heap manipulation. By carefully sizing the value data, we showed that attackers can leak sensitive information:

- **Heap addresses**: Through tcache metadata (16-byte allocations)
- **Libc addresses**: Via unsorted bin pointers (1024-byte allocations)

The leak is particularly powerful on Ubuntu 20.04 (glibc 2.31), where `tcache_perthread_struct` pointers reveal heap base addresses. On newer systems (glibc 2.34+), the `tcache_key` provides less useful information but still indicates heap activity.

Patch Analysis. The fix ensures `pCur->zData` is always reset when advancing the cursor:

```
diff --git a/ext/lsm1/lsm_vtab.c b/ext/lsm1/lsm_vtab.c
index 8c21923e1a..cfc80de883 100644
--- a/ext/lsm1/lsm_vtab.c
+++ b/ext/lsm1/lsm_vtab.c
@@ -389,7 +389,7 @@ static int lsm1Next(sqlite3_vtab_cursor *cur){
         if( c>0 ) pCur->atEof = 1;
       }
     }
-    pCur->zData = 0;
   }
+  pCur->zData = 0;
   return rc==LSM_OK ? SQLITE_OK : SQLITE_ERROR;
 }
```

This simple one-line change moves the cache reset outside the conditional block, ensuring stale pointers are cleared regardless of the `bUnique` flag. The patch maintains backward compatibility while eliminating the use-after-free condition.

11.3 APACHE COMMONS COMPRESS: Out-of-bounds Write

We discovered a 0-day vulnerability in Apache Commons Compress, specifically in the LZW decompression system used for processing ZIP archives with unshrinking compression. The vulnerability leads to an infinite loop during decompression, ultimately causing a negative array index write attempt.

Root Cause Analysis. The bug exists in the `expandCodeToOutputStack` method of `LZWInputStream`. This method traverses a prefix lookup table to decompress LZW codes, decrementing an index while writing to an output buffer:

```
protected int expandCodeToOutputStack(final int code, final boolean addedUnfinishedEntry)
    throws IOException {
    for (int entry = code; entry >= 0; entry = prefixes[entry]) {
        outputStack[--outputStackLocation] = characters[entry];
    }
    if (previousCode != -1 && !addedUnfinishedEntry) {
        addEntry(previousCode, outputStack[outputStackLocation]);
    }
    previousCode = code;
    previousCodeFirstChar = outputStack[outputStackLocation];
    return outputStackLocation;
}
```

The vulnerability occurs when the `addEntry` method creates a cycle in the prefix lookup table. During normal operation, `addEntry` builds the compression dictionary by linking previous codes:

```
protected int addEntry(final int previousCode, final byte character,
                       final int maxTableSize) {
    if (tableSize < maxTableSize) {
        prefixes[tableSize] = previousCode;
        characters[tableSize] = character;
        return tableSize++;
    }
    return -1;
}
```

Through crafted input, we can manipulate the decompression state to create a 2-cycle between entries 257 and 295 in the prefix table. When `expandCodeToOutputStack` traverses this cyclic structure, it enters an infinite loop where `outputStackLocation` decrements indefinitely, eventually attempting to write at index -1.

Our analysis revealed the exact sequence leading to the crash. The decompression initially proceeds normally, building the dictionary with entries 0–256. At a critical point, the following operations occur:

- Entry 257 is added with `previousCode=295`
- Entry 295 is added with `previousCode=257`
- When code 295 is expanded, the traversal alternates: 295 → 257 → 295 → 257 → ...

The output stack location decrements from 8192 down through 0, then attempts to access index -1, triggering an `ArrayIndexOutOfBoundsException`.

PoC. We crafted a malicious ZIP archive that triggers the vulnerability. The 255-byte payload contains carefully constructed LZW-compressed data that creates the cyclic dependency:

```
0000: 504b 0304 2e00 0000 0c00 84b6 ba46 72b6  PK...........Fr.
0010: 0063 7700 fe00 6b00 0000 0300 1c00 6262  .cw...k.......bb
0020: 6255 5409 0003 e7ce 6455 f3ce 6455 7578  bUT.....dU..dUux
0030: 0b00 5704 8800 5c13 f904 0100 0042 5a68  ..W...\......BZh
0040: 3931 4159 2653 5962 e44f 5100 000d d180  91AY&SYb.OQ.....
```

Security Impact. While the immediate effect is a denial of service through an unhandled exception, the vulnerability has broader implications:

- **Memory corruption potential**: The negative index write could corrupt adjacent memory structures in languages without bounds checking.
- **Attack surface**: Any application using Apache Commons Compress to handle untrusted ZIP files is vulnerable.
- **Bypass potential**: The infinite loop occurs before size validation checks, potentially bypassing security controls.

This vulnerability demonstrates how subtle logic errors in compression algorithms can lead to severe security issues, particularly when handling complex data structures like cyclic graphs in a linear traversal context.

11.4 SQLite3: SIGBUS in LSM1 Extension

We discovered a SIGBUS vulnerability in SQLite3's LSM1 extension that manifests when multiple virtual tables reference the same database file through different paths. The vulnerability stems from improper handling of shared database connections when the same file is accessed via both absolute and relative paths.

Root Cause Analysis. The bug occurs due to flawed path comparison logic in the LSM1 database connection manager. When SQLite3 opens an LSM1 virtual table, it attempts to reuse existing database connections by comparing canonical paths. However, the path resolution mechanism fails to properly normalize paths, treating `/tmp/tmp.lsm` and `../../../../../../tmp/tmp.lsm` as distinct databases despite referencing the same underlying file.

This leads to a race condition during database closure. When the first connection closes, it truncates the shared file to zero bytes. The second connection, unaware of this modification, continues accessing the now-invalid memory mapping, resulting in a SIGBUS when attempting to read from the truncated file's former address space.

PoC. The following SQL demonstrates the vulnerability:

```
.load ./lsm

CREATE VIRTUAL TABLE lsm_table_1 USING lsm1
    ('/tmp/tmp.lsm', id, TEXT, data);
CREATE VIRTUAL TABLE lsm_table_2 USING lsm1
    ('../../../../../../tmp/tmp.lsm', id, TEXT, data);
```

Executing these statements triggers a SIGBUS during database shutdown as SQLite3 attempts to access memory that was unmapped when the first table's connection truncated the shared file.

The vulnerability manifests through the following sequence of events:

- Both virtual tables map the same file but LSM1 treats them as separate databases
- During shutdown, the first connection acquires an exclusive lock (`LSM_LOCK_DMS2`)
- The `doDbDisconnect` function truncates the database file via `dbTruncateFile`
- The second connection still holds a memory mapping to the now-truncated file
- When `lsmCheckpointId` attempts to read from `0x7ffff6f32000`, it triggers SIGBUS

Security Impact. While this vulnerability primarily causes denial of service through application crashes, it reveals deeper architectural issues:

- **Path confusion**: Applications using relative paths may inadvertently create conflicting database connections
- **Resource exhaustion**: Attackers could create multiple virtual tables with varying paths to the same file, causing repeated crashes
- **Data integrity**: The unexpected truncation could lead to data loss if other processes are accessing the same database file

The vulnerability highlights the importance of proper path canonicalization and resource tracking in database systems, particularly when dealing with memory-mapped files and shared resources.

12 Conclusion

Our team started this competition as AI skeptics but has now become a strong advocate for using LLMs in hardcore, traditional security tasks. Thanks to AIxCC, we had the opportunity to gain first-hand experience with LLMs—in fact, evolving together with state-of-the-art LLM services over the past two years, and realized that the majority of security tasks once believed impossible are now feasible with LLMs. As the speed of improvement in foundation models and surrounding tools is unprecedentedly fast, it is difficult to imagine how this will transform our security industries and research in the coming years. Our team will continue innovating in this area of research and seeking ways to conduct follow-up research and development together with various stakeholders—OpenAI has already started collaborating with our team, and we hope to contribute to the mission of DARPA and government organizations in the future.

The implementation and benchmarks will be made publicly available: https://github.com/Team-Atlanta/. We plan to release the details of our CRS via a series of blog postings at https://team-atlanta.github.io/ as well as through publication in academic conferences.

13 Acknowledgment

As an open track team, it would not have been possible to dedicate our time for this journey without the support of our organizations. We sincerely thank Georgia Tech, Samsung Research, KAIST, and POSTECH.

References

[1] *Proceedings of the 32th USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[2] Anthropic. Prompt engineering techniques, 2024. URL https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/overview. Accessed: 2025-01-14.

[3] Anthropic. Use XML tags in Claude prompts, 2024. URL https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/use-xml-tags. Accessed: 2025-01-14.

[4] Anthropic. Claude code overview. https://docs.anthropic.com/en/docs/claude-code/overview, 2025. Accessed: 2025-07-27.

[5] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert. Nautilus: Fishing for deep bugs with grammars. In *NDSS*, volume 19, page 337, 2019.

[6] J. Chen, W. Han, M. Yin, H. Zeng, C. Song, B. Lee, H. Yin, and I. Shin. SymSan: Time and Space Efficient Concolic Execution via Dynamic Data-Flow Analysis. In *USENIX Security Symposium (Security)*. USENIX Association, Aug. 2022. URL https://www.usenix.org/conference/usenixsecurity22/presentation/chen-ju.

[7] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, pages 3–9, 1978.

[8] Y. Chen, R. Zhong, Y. Yang, H. Hu, D. Wu, and W. Lee. μFUZZ: Redesign of parallel fuzzing using microservice architecture. In *Proceedings of the 32th USENIX Security Symposium (Security)* SEC [1].

[9] M. R. Clarkson and F. B. Schneider. Hyperproperties. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 51–65, 2008. doi: 10.1109/CSF.2008.7.

[10] A. Crump, A. Fioraldi, D. Maier, and D. Zhang. Libafl libfuzzer: Libfuzzer on top of libafl. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, pages 70–72, 2023. doi: 10.1109/SBFT59156.2023.00021.

[11] U. Ctags. ctags. https://github.com/universal-ctags/ctags, 2025. Accessed: 2025-07-27.

[12] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis. Retracer: Triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, New York, NY, May 2016.

[13] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and I. Yun. REPT: Reverse Debugging of Failures in Deployed Software. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, Oct. 2018.

[14] DARPA. AI Cyber Challenge (AIxCC), 2024. URL https://aicyberchallenge.com/. Accessed: 2025-09-09.

[15] DARPA. Final Competition Procedures and Scoring Guide, 2025. URL https://aicyberchallenge.com/final-competition-procedures-and-scoring-guide/. Accessed: 2025-09-09.

[16] P. Deligiannis, A. Lal, N. Mehrotra, R. Poddar, and A. Rastogi. RustAssistant: Using LLMs to fix compilation errors in Rust code. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 267–279. IEEE Computer Society, 2024.

[17] T. E. Felix Machtle, Florian Sieck. Swat: Modular dynamic symbolic execution for java applications using dynamic instrumentation. https://github.com/SWAT-project/SWAT, 2024. Accessed: 2025-07-31.

[18] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[19] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 29th ACM Conference on Computer and Communications Security*, CCS '22, pages 1331–1347, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394505. doi: 10.1145/3548606.3560602. URL https://doi.org/10.1145/3548606.3560602.

[20] Y.-F. Fu, J. Lee, and T. Kim. autofz: Automated Fuzzer Composition at Runtime. In *Proceedings of the 32th USENIX Security Symposium (Security)* SEC [1].

[21] P. Gauthier. Aider: Ai pair programming in your terminal. https://aider.chat/, 2025.

[22] Google. OSS-Fuzz: Continuous Fuzzing for Open Source Software, 2024. URL https://github.com/google/oss-fuzz. Accessed: 2025-01-14.

[23] GraalVM. Espresso. https://www.graalvm.org/latest/reference-manual/espresso/.

[24] J. Gu, X. Jiang, Z. Shi, H. Tan, X. Zhai, C. Xu, W. Li, Y. Shen, S. Ma, H. Liu, et al. A survey on llm-as-a-judge. *arXiv preprint arXiv:2411.15594*, 2024.

[25] R. Hodován, Á. Kiss, and T. Gyimóthy. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM*

SIGSOFT international workshop on automating TEST case design, selection, and evaluation, pages 45–48, 2018.

[26] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, 2012.

[27] Z. Jiang, X. Jiang, A. Hazimeh, C. Tang, C. Zhang, and M. Payer. Igor: Crash deduplication through root-cause clustering. In *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*, Virtual Event, Nov. 2021.

[28] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

[29] LLVM Project. libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed: 2024.

[30] L. Mei, J. Yao, Y. Ge, Y. Wang, B. Bi, Y. Cai, J. Liu, M. Li, Z.-Z. Li, D. Zhang, C. Zhou, J. Mao, T. Xia, J. Guo, and S. Liu. A survey of context engineering for large language models, 2025. URL https://arxiv.org/abs/2507.13334.

[31] X. Mi, S. Rawat, C. Giuffrida, and H. Bos. LeanSym: Efficient Hybrid Fuzzing Through Conservative Constraint Debloating. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '21, pages 62–77, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450390583. doi: 10.1145/3471621.3471852. URL https://doi.org/10.1145/3471621.3471852.

[32] A. Neelakantan, T. Xu, R. Puri, A. Radford, J. M. Han, J. Tworek, Q. Yuan, N. Tezak, J. W. Kim, C. Hallacy, et al. Text and code embeddings by contrastive pre-training. *arXiv preprint arXiv:2201.10005*, 2022.

[33] OpenAI. Prompt engineering guide, 2024. URL https://platform.openai.com/docs/guides/prompt-engineering. Accessed: 2025-01-14.

[34] D. Opitz and R. Maclin. Popular ensemble methods: An empirical study. *Journal of artificial intelligence research*, 11: 169–198, 1999.

[35] Oracle. jstack - Stack Trace. https://docs.oracle.com/javase/7/docs/technotes/tools/share/jstack.html.

[36] S. Poeplau and A. Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198, Berkeley, CA, USA, Aug. 2020. USENIX Association. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau.

[37] S. Poeplau and A. Francillon. SymQEMU: Compilation-based symbolic execution for binaries. In *Network and Distributed System Security Symposium*, NDSS '21. The Internet Society, 2021. URL https://www.ndss-symposium.org/ndss-paper/symqemu-compilation-based-symbolic-execution-for-binaries/.

[38] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36, 2015.

[39] P. Schanely. CrossHair: An analysis tool for Python that blurs the line between testing and type systems, 2024. URL https://github.com/pschanely/CrossHair. GitHub repository.

[40] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, M. Zhang, Y. K. Li, Y. Wu, and D. Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/2402.03300, 2024. URL https://doi.org/10.48550/arXiv.2402.03300.

[41] Tree-sitter. Tree-sitter. https://tree-sitter.github.io/tree-sitter/, 2025. Accessed: 2025-07-27.

[42] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

[43] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

[44] J. Wei, X. Wang, D. Schuurmans, M. Bosma, brian ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou. Chain of thought prompting elicits reasoning in large language models. In A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=_VjQlMeSB_J.

[45] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

[46] Y. Zhang, J. Wang, D. Berzin, M. Mirchev, D. Liu, A. Arya, O. Chang, and A. Roychoudhury. Fixing security vulnerabilities with ai in oss-fuzz, 2024. URL https://arxiv.org/abs/2411.03346.

A Bug-Finding Module Performance by Harness

This appendix provides detailed bug-finding performance data for fuzzing harnesses in the AIxCC final competition based on available logs. The results are organized by programming language: Table 25 covers 46 C harnesses (35 individual harnesses plus 11 grouped with zero passed PoVs), Table 26 covers 25 Java harnesses (13 individual harnesses plus 12 grouped with zero passed PoVs), plus 1 unknown harness (1 PoV, 1 passed, 1 patch). For each harness, we report five key metrics:

Total PoVs. The total number of proof-of-vulnerabilities generated by all bug-finding modules for each harness, regardless of their final verification status.

Passed PoVs. PoVs that successfully triggered vulnerabilities and passed all verification checks: (1) crashed the harness with appropriate return codes, (2) for delta-mode harnesses, crashed only the HEAD version but not the BASE version, and (3) passed deduplication analysis using stack trace signatures. These contribute directly to the team's competition score.

Duplicate PoVs. PoVs that successfully triggered real vulnerabilities but were identified as duplicates of previously submitted PoVs through stack trace analysis and crash signature matching. While these represent legitimate vulnerability discoveries, they do not contribute to scoring due to redundancy.

Passed Patches. Patches that successfully remediated identified vulnerabilities. These must pass verification showing that (1) the patched version no longer crashes with the original PoV and (2) the patch does not break existing functionality. Successful patches contribute additional points to the competition score.

Successful Bug-Finding Modules. The specific ATLANTIS components that successfully generated passed or duplicate PoVs for each harness. This shows which approaches successfully generated passed or duplicate PoVs for different harness types.

These granular results show how different bug-finding modules performed across harness types.

The harness-level data reveals several key insights. Among C harnesses, `exif_from_data_fuzzer` (18 passed PoVs) and `curl_fuzzer_dict` (13 passed PoVs) emerged as the most productive targets, both benefiting from multiple active bug-finding modules working in combination. Harnesses with multiple successful modules, such as `exif_from_data_fuzzer`, engaged up to 6 different approaches simultaneously.

Java harnesses showed lower pass rates relative to total PoVs generated. `ExpanderFuzzer` generated 208 PoVs with 6 passed verification, while `CompressorGzipFuzzer` generated 84 PoVs with 2 passed.

Based on the available log data, ATLANTIS generated 1,003 PoVs total, with 118 passing final verification and 151 identified as duplicates. Of the harnesses captured in these logs, 23 (32%) produced no verified vulnerabilities.

Harness	PoVs			Patches	Successful Bug-Finding
	Total	Passed	Dup.	Passed	Modules
TestFuzzCryptoCertificateDataSetPEM	7	1	0	1	Multilang.testlang_input_gen
curl_fuzzer	8	1	0	1	Multilang.given_fuzzer
curl_fuzzer_dict	16	13	0	1	Multilang.concolic_input_gen, Multilang.given_fuzzer
curl_fuzzer_ftp	3	1	0	1	Multilang.mlla
curl_fuzzer_http	8	1	0	0	C
curl_fuzzer_rtsp	9	5	0	1	C
curl_fuzzer_tftp	23	2	0	0	Multilang.given_fuzzer
exif_from_data_fuzzer	104	18	2	1	Multilang.given_corpus, Multilang.concolic_input_gen, Multilang.given_fuzzer, Multilang.shared_corpus, Multilang.testlang_input_gen, C
exif_loader_fuzzer	38	7	0	0	Multilang.given_fuzzer
fuzz	84	6	66	4	Multilang.given_corpus, Multilang.given_fuzzer, Multilang.mlla.gen, C
fuzz-catalog	11	3	0	0	Multilang.given_fuzzer
fuzz-link-parser	5	3	0	0	Multilang.given_corpus
fuzz-netdev-parser	5	2	0	0	Multilang.given_corpus, Multilang.testlang_input_gen
fuzz-network-parser	5	2	0	0	Multilang.given_corpus, Multilang.mlla
fuzz-udev-rule-parse-value	3	2	0	0	Multilang.given_fuzzer
fuzz-unit-file	1	1	0	0	Multilang.given_corpus
fuzz_encode_stream	4	1	0	1	Multilang.given_fuzzer
handler_aim	1	1	0	1	Multilang.given_fuzzer
handler_ber	2	1	0	1	Multilang.given_fuzzer
handler_gvcp	1	1	0	1	Multilang.given_corpus
handler_icmp_extension	1	1	0	1	Multilang.given_fuzzer
handler_irc	1	1	0	1	Multilang.given_fuzzer
handler_json	2	1	0	1	Multilang.given_fuzzer
handler_netbios	2	2	0	1	Multilang.given_fuzzer
handler_telnet	9	5	0	3	Multilang.given_fuzzer
handler_wlan_centrino	4	1	0	0	Multilang.given_fuzzer
handler_wlan_noqos	3	1	0	1	Multilang.mlla.mut
handler_wlan_withfcs	4	1	0	1	Multilang.given_fuzzer
handler_zbee_zdp	1	1	0	1	Multilang.given_fuzzer
json_fuzz	1	1	0	1	Multilang.given_fuzzer
lint	1	1	0	0	Multilang.given_fuzzer
reader	1	1	0	0	Multilang.given_fuzzer
schema	8	3	0	0	Multilang.given_fuzzer, C
xml	1	1	0	0	Multilang.given_fuzzer
xpath	2	1	0	0	Multilang.given_fuzzer
Other C harnesses (11)	23	0	1	0	-
Total C harnesses (46)	**402**	**94**	**69**	**25**	**Multiple finders**

Multilang: ATLANTIS-Multilang, C: ATLANTIS-C.

Table 25: Performance breakdown for C harnesses in the AIxCC final competition.

Harness	PoVs			Patches	Successful Bug-Finding Modules
	Total	Passed	Dup.	Passed	
CompressorGzipFuzzer	84	2	25	1	Java
DomXfaParserFuzzer	2	1	0	1	Java
DomXmpParserFuzzer	3	1	0	1	Java
ExcelImExportServiceFuzzer	4	2	0	0	Multilang.given_fuzzer, Java
ExpanderFuzzer	208	6	41	3	Multilang.given_fuzzer, Multilang.shared_corpus, Multilang.testlang_input_gen, Java
HtmlFuzzer	19	1	0	1	Multilang.given_fuzzer
JsonFuzzer	10	3	0	2	Multilang.given_fuzzer, Multilang.testlang_input_gen, Java
PDFExtractTextFuzzer	4	1	0	1	Java
PDFOCRFuzzer	3	2	0	1	Java
PDFStreamParserFuzzer	3	1	0	1	Java
PrivateDictionaryFuzzer	6	1	0	1	Multilang.mlla.gen
SimpleLoggerFuzzer	2	1	0	1	Multilang.given_fuzzer
TextAndCSVParserFuzzer	2	1	0	1	Java
Other Java harnesses (12)	250	0	16	0	-
Total Java harnesses (25)	**600**	**23**	**82**	**15**	**Multiple finders**

Multilang: ATLANTIS-Multilang, Java: ATLANTIS-Java

Table 26: Performance breakdown for Java harnesses in the AIxCC final competition.