Lebooboo dubai chocolate

CK && OL

Glossary

Introduction

AIxCC FInal Competition Overview

Organized by DARPA (with ARPA-H support), running from 2023–2025.

A 2-year AI-focused cybersecurity competition with $29.5 million in prizes.

Goal: Build autonomous systems (Cyber Reasoning Systems or CRSs) to automatically find and fix software vulnerabilities.

Motivation: As software becomes more complex, human-only methods can't keep up. Many zero-day vulnerabilities remain undetected.

Partners include top AI companies: Anthropic, Google, OpenAI.

Team Atlantis

ATLANTIS is an automated system developed to discover and fix software vulnerabilities by combining advanced techniques such as symbolic execution, directed fuzzing, and static analysis. What sets ATLANTIS apart is its deep integration of large language models (LLMs), which enhance the system's ability to understand code context, identify vulnerabilities more intelligently, and generate high-quality patches—going beyond the limitations of traditional program analysis. The system was designed to tackle key challenges in autonomous vulnerability discovery, including the need to scale across diverse programming languages like C and Java, maintain high precision while ensuring broad code coverage, and produce patches that are not only effective but also semantically correct. By blending modern AI with established security techniques, ATLANTIS pushes the boundaries of what automated cybersecurity tools can achieve. The full design, architecture, and implementation strategies are detailed in a technical report, and the entire system has been released as open-source software to support further innovation and community-driven advancement in automated vulnerability detection and patching.

AIxCC Final Competition Summary (2025)

DARPA's **AI Cyber Challenge (AIxCC)** aimed to advance cybersecurity using **AI-powered autonomous systems**. Unlike the earlier Cyber Grand Challenge, which worked on simplified test environments, AIxCC focused on securing **real-world open-source software** used in critical infrastructure.

Seven finalist teams competed in June 2025 at DEF CON 33, running their **Cyber Reasoning Systems (CRSes)** under strict **time, compute, and AI usage limits**. Each system had to:

- Discover bugs (vulnerabilities),
- Generate correct patches,
- Assess AI-generated vulnerability reports (SARIF),
- Bundle findings for better scoring.

Challenges were derived from **Google's OSS-Fuzz** and included "full," "delta," and "unharnessed" modes.

Atlantis

Team Atlanta's Winning Strategy

Team Atlanta won **1st place** with the **highest total score (392.76)** by:

- Finding **43 of 70 vulnerabilities (61%)**, more than any other team.
- Maintaining **91% accuracy**, avoiding false or low-quality submissions.
- Excelling in **bug discovery, patch generation**, and **bundled submissions**.

They strategically used:

- **$73.9K in Azure compute** and **$29.4K in LLM API credits**,
- Advanced LLMs (e.g. GPT-4o, Claude Opus) to generate more insightful and verbose output,
- Careful compute and rate-limit management to handle multiple concurrent challenge projects.

Their system, **ATLANTIS**, deeply integrated LLMs with symbolic execution, fuzzing, and static analysis, outperforming traditional approaches.

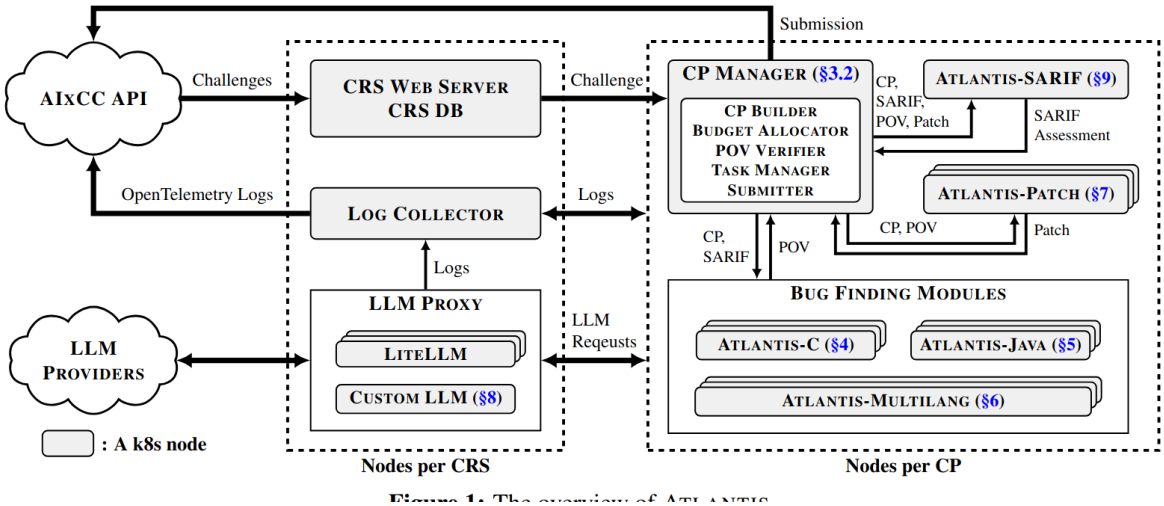


Figure 1 for reference

The primary goal of the AIxCC competition is to build a Cyber Reasoning System (CRS) that leverages Large Language Models (LLMs) to autonomously discover proof-of-vulnerabilities (PoVs) in challenge projects (CPs), generate semantically correct patches for the discovered vulnerabilities, and assess the accuracy of static analysis reports (SARIF). To meet these objectives, we developed ATLANTIS, a distributed, LLM-integrated CRS that satisfies the following design requirements.

R1: Support Multiple CPs Concurrently — ATLANTIS must handle multiple CPs in parallel under constrained resources. This includes simultaneously discovering PoVs, generating patches, and assessing SARIF reports for multiple CPs. The system must support efficient resource sharing and scheduling across overlapping CP batches and remain scalable under tight deadlines and dynamic workloads.

```C/C++
bool parallel::vector_pool::get_vector(unsigned owner, unsigned& n,
unsigned const*& ptr) {

    unsigned head = m_heads[owner];

    unsigned iterations = 0;

    while (head != m_tail || !m_at_end[owner]) {

        ++iterations;

        SASSERT(head < m_size && m_tail < m_size);

        bool is_self = owner == get_owner(head);

        next(m_heads[owner]);
```

```
        m_at_end[owner] = (m_heads[owner] == m_tail);
        if (!is_self) {
            n = get_length(head);
            ptr = get_ptr(head);
            return true;
        }
        head = m_heads[owner];
    }
    return false;
}
```

This is example code for parallel loops through the shared pool to fetch a clause vector for a solver thread. It ensures threads don't pick up their own clauses and can continue processing as new vectors become available.

R2: Fail-Safe Architecture — ATLANTIS must be resilient to individual CP or module failures. A failure in one task must not affect the processing of others. The system must support fault isolation per CP or subtask, include retry logic and timeouts, and maintain persistent logging and state tracking to support recovery and continuity of operations.

Python

```python
async def _watchdog(self):
    assert self._stop_evt is not None
    interval = self._WATCHDOG_INTERVAL

    while not self._stop_evt.is_set():
        for proc_id, proc in list(self.procs.items()):
            try:
                if proc.process.is_alive():
                    continue

                exitcode = proc.process.exitcode
                logger.warning(f"Executor: {proc_id} died (exit={exitcode}).")
```

```
                if (self._MAX_RESTART is not None and self._restart_cnt[proc_id] >=
self._MAX_RESTART):
                    logger.error(f"Executor: {proc_id} exceeds restart limit, skipping.")
                    continue

                logger.info(f"Executor: {proc_id} restarting...")
                self._restart_cnt[proc_id] += 1

                try:
                    proc.stop()
                except Exception:
                    logger.error(f"Executor: failed to stop process {proc_id}.",
exc_info=True)

                new_proc = self._proc_factories[proc_id]()
                new_proc.start()
                self.procs[proc_id] = new_proc
                logger.info(f"Executor: {proc_id} respawned
(#{self._restart_cnt[proc_id]}).")

            except Exception as e:
                logger.error(f"Executor: Error in watchdog for {proc_id}: {e}",
exc_info=True)

        try:
            await asyncio.wait_for(self._stop_evt.wait(), timeout=interval)
        except asyncio.TimeoutError:
            pass
```

R3: Efficient Budget Utilization (LLM + Azure) — Given budget constraints for LLM usage and Azure compute resources, ATLANTIS must prioritize tasks based on cost-effectiveness. It must dynamically allocate compute resources and rate-limit LLM calls to prevent overuse. The system should monitor and optimize token usage across modules to remain within budget while maximizing performance.

Python

```
def _update_metrics(self, response, cost):
    self.total_cost += cost
    self.request_count += 1
    if hasattr(response, "usage"):
        self.total_tokens += response.usage.total_tokens
```

```
        self.total_prompt_tokens += response.usage.prompt_tokens
        self.total_completion_tokens += response.usage.completion_tokens
```

Python

```python
def get_context_limit(self) -> int:
        """Returns the token limit for the given model name.

        Args:
            model_name (str): The name of the model.

        Returns:
            int: The token limit for the model.
        """
        model_data = TOKEN_COSTS.get(self.model_name, None)
        if model_data is None:
            logger.warning(f"Model {self.model_name} not found in TOKEN_COSTS")
            return 8192
        return model_data.get("max_input_tokens", 8192)

    def get_output_limit(self) -> int:
        """Returns the token limit for the given model name.

        Args:
            model_name (str): The name of the model.

        Returns:
            int: The token limit for the model.
        """
        model_data = TOKEN_COSTS.get(self.model_name, None)
        if model_data is None:
            logger.error(f"Model {self.model_name} not found in TOKEN_COSTS")
            return 4096
        return model_data.get("max_output_tokens", 4096)
```

R4: Observability and Cloud Infrastructure Compliance — To comply with AIxCC requirements, ATLANTIS must log telemetry data, such as LLM inputs and outputs, in the OpenTelemetry format. Additionally, it must be deployable on Azure using Terraform to ensure reproducibility, portability, and scalable infrastructure management.

Python

```
class OpenTelemetryConfig:
    exporter: Union[str, SpanExporter] = "console"
    endpoint: Optional[str] = None
    headers: Optional[str] = None

    @classmethod
    def from_env(cls):
        exporter = os.getenv("OTEL_EXPORTER", "console")
        endpoint = os.getenv("OTEL_ENDPOINT")
        headers = os.getenv("OTEL_HEADERS")
        return cls(exporter=exporter, endpoint=endpoint, headers=headers)
```

None

```
az group create --name example-tfstate-rg --location eastus
az storage account create --resource-group example-tfstate-rg --name
examplestorageaccountname --sku Standard_LRS --encryption-services blob
az storage container create --name tfstate --account-name examplestorageaccountname
--auth-mode login

terraform {
  backend "azurerm" {
    resource_group_name  = "example-tfstate-rg"
    storage_account_name = "examplestorageaccountname"
    container_name       = "tfstate"
    key                  = "terraform.tfstate"
  }
}
```

None

```
provider "azurerm" {
  features {}
  subscription_id = var.ARM_SUBSCRIPTION_ID
  tenant_id       = var.ARM_TENANT_ID
  client_id       = var.ARM_CLIENT_ID
  client_secret   = var.ARM_CLIENT_SECRET
}
```

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 launches three CRS-level nodes:
CRS-WEBSERVER, LOG COLLECTOR, and LLM PROXY.

The LOG COLLECTOR gathers and forwards logs to the organizers. The LLM PROXY, based on LiteLLM, centrally manages LLM usage to stay within budget and logs all LLM requests and responses.
CRS-WEBSERVER listens for incoming CPs or SARIF reports, stores CP metadata in the database, and spawns a CP-MANAGER instance on a CP-level Kubernetes node.

Each CP-MANAGER builds the CP, allocates Azure compute and LLM usage budgets proportionally, and launches Kubernetes nodes for bug-finding modules based on the CP type and harness count.

For C-based CPs, ATLANTIS-C and ATLANTIS-Multilang are deployed; for Java-based CPs, ATLANTIS-Java and ATLANTIS-Multilang are launched.

ATLANTIS-Patch and ATLANTIS-SARIF handle patch generation and SARIF assessment, respectively.
CP-MANAGER issues LiteLLM API keys with budgets calculated by total CPs and overall LLM budget.

Modules use these keys to ensure budget compliance and distribute RPM and TPM limits evenly. Only ATLANTIS-Patch uses a custom fine-tuned LLM. Bug-finding modules send PoVs to CP-MANAGER, which verifies crashes, deduplicates results using stack traces, submits unique PoVs to organizers, and forwards them to ATLANTIS-Patch and ATLANTIS-SARIF. After patch and SARIF report generation, CP-MANAGER bundles them with PoVs and submits the complete package to organizers.

CP-MANAGER consists of several key subcomponents. DB compiles the given CP with multiple configurations depending on the language (C or Java) and shares compiled artifacts via a Kubernetes shared file system. It also identifies fuzzing harnesses and source code locations, storing this data in JSON format for budget allocation and LLM-based analyses. BUDGET ALLOCATOR distributes both LLM and Azure budgets per CP based on remaining total budget and CP count. It reserves $500 of the LLM budget for ATLANTIS-SARIF, then splits the rest as 25% to ATLANTIS-Patch, 37.5% each to ATLANTIS-Multilang and ATLANTIS-C (or ATLANTIS-Java). Azure nodes are allocated with fixed nodes for CP-MANAGER, ATLANTIS-SARIF, and ATLANTIS-Multilang utilities, five nodes for ATLANTIS-Patch agents, and the remaining compute divided between ATLANTIS-Multilang and ATLANTIS-C/Java fuzzing tasks, scaling up to 15 nodes for ATLANTIS-C as budget permits. Unused budget is reallocated for future CPs.

POV VERIFIER checks submitted PoVs for reliable crashes with specific return codes (sanitizer crashes 1/77, timeouts 70, out-of-memory 71) in a matching environment. For delta-mode CPs, it confirms PoVs don’t crash the BASE version. PoV deduplication uses stack trace analysis with bug-type heuristics instead of textual

similarity to identify unique PoVs, limiting patch requests to 10 per fuzzer-sanitizer pair to conserve resources and reduce LLM/CPU usage.

TASK MANAGER orchestrates multi-step workflows across modules. Upon receiving SARIF reports from CRS-WEBSERVER, it forwards them to ATLANTIS-SARIF and then to bug-finders for directed fuzzing. Unique PoVs are sent to organizers and ATLANTIS-Patch, with polling for verification results. Verified PoVs trigger PoV–SARIF mapping requests. Generated patches are verified and then passed to ATLANTIS-SARIF for final assessment. Communication is via web APIs with state sync managed by Redis for concurrency.

SUBMITTER manages submitting PoVs, patches, and SARIF assessments to organizers, avoiding penalties for duplicate patch submissions via dynamic bundling. It groups PoVs under single representatives, manages patch selection, and bundles PoVs with patches and SARIF assessments to maximize competition points. A background process monitors and updates bundles continuously.

Performance results show ATLANTIS generated 1,003 PoVs (118 verified), produced 47 patches (87.2% success), and correctly assessed 8 of 10 SARIF reports, covering 62 harnesses with 54 unique results. Bundling success was 89.4%. ATLANTIS-Multilang contributed most verified PoVs through multi-tier LLM input generation, ATLANTIS-C excelled with multi-fuzzer ensembles and LLM seed augmentation, and ATLANTIS-Java focused on sink-centered analysis of API vulnerabilities. The system's strength lies in complementary specialization and coordinated AI-enhanced approaches addressing diverse vulnerability patterns.

Atlantic-C

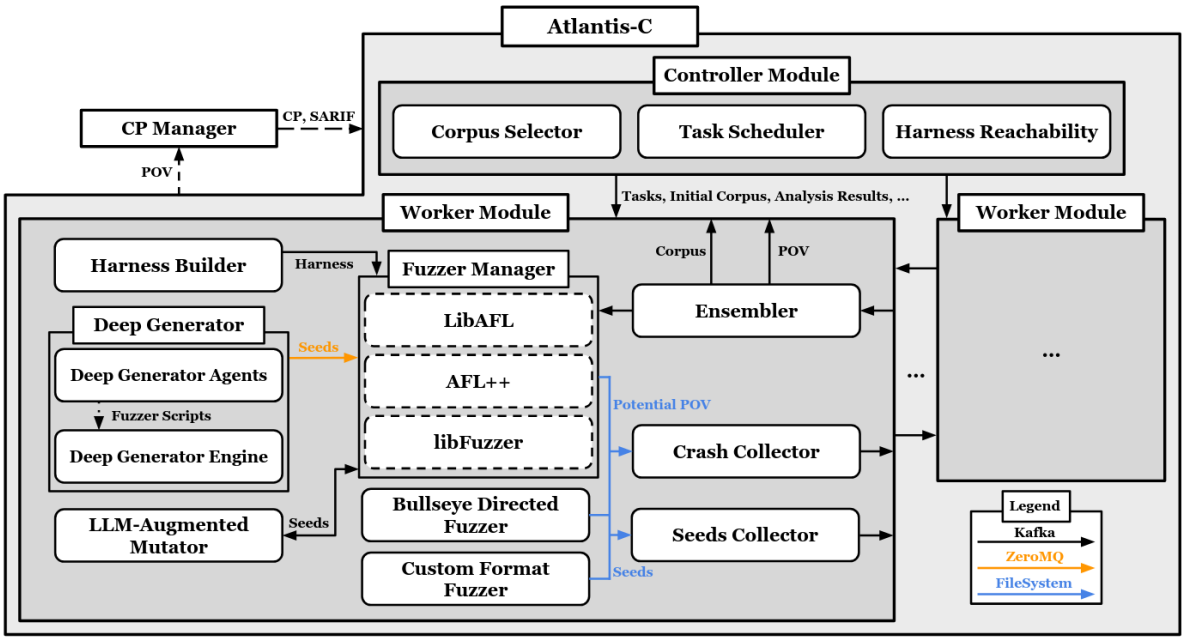


Figure 2: Overall Design of ATLANTIS-C

Atlantis-C targets vulnerability discover in C/C++ challenge projects by orchestrating a coordinated ensemble of fuzzers. They have found that individual engines such as libAFL, AFL++, and libFuzzer respond differently to harness structures, input formats, and coding idioms found in CPs, leading to bad performance when the

fuzzing is used alone. Atlantis-C containerizes and runs them side by side so that the containers can be attached to any engine to provide shared corpus management, feedback, and mutation strategies.

Atlantis-C is a multi-node, LLM augmented fuzzing system, based on a Kafka microservice architecture. Each microservice is a separate containerized service, meaning it can be split into two modules: the controller module and the worker module. When Atlantis-C is deployed onto multiple Kubernetes nodes, one of the nodes runs both a controller and a work mode, while the other nodes only run a worker module.

The controller module contains microservices that are responsible for overall task scheduling and interaction with the other systems outside of Atlantis-C. The algorithm is epoch-based for scheduling the fuzzing tasks and is responsible for the orchestrator.

WORKFLOW

HARNESSBUILDER is responsible for preparing each fuzzing harness for analysis by producing multiple instrumented builds tailored to different fuzzing engines and strategies. For each challenge, it generates builds compatible with **libFuzzer**, **LibAFL**, **AFL++**, as well as **directed** and **coverage-guided** fuzzing configurations. These builds are registered as discrete fuzzing tasks within the system. Each task includes metadata such as the harness path, build type, and target symbols or entry points.

The generated builds support the diverse fuzzing engines used by **ATLANTIS-C**, which leverages the differing strengths of each engine. Because engines vary in their responsiveness to coding idioms, input formats, and harness structures, the HARNESSBUILDER ensures coverage by preparing each harness in a format best suited for all engines. For example, libFuzzer builds use LLVM instrumentation and support in-process coverage merging; AFL++ builds may use forkserver-based execution and persistent loops; LibAFL builds allow more fine-grained corpus feedback and mutation scheduling.

The **HARNESSBUILDER** runs inside the ATLANTIS-C controller node and outputs are saved to a shared Kubernetes volume. These artifacts are then distributed to fuzzers via the **TASKSCHEDULER** and **FUZZERMANAGER**, which manage execution across nodes. By producing multiple fuzzing variants per harness up front, HARNESSBUILDER enables the ensemble system to experiment with and rebalance fuzzing strategies dynamically during execution, guided by signals such as SARIF alerts, OSV metadata, reachability scores, and crash discovery.

Multi-Fuzzer Integration

HARNESSBUILDER orchestrates multi-mode instrumentation for each fuzzing harness, enabling support for diverse fuzzing engines and strategies within **ATLANTIS-C**.

It supports **seven instrumentation modes**:

1. **LibFuzzer** – Standard OSS-Fuzz-compatible instrumentation.
2. **LibAFL** – Uses environment variable overrides to support custom fork-based execution.
3. **Compiler Optimizations** – Enables specific compiler flags for performance or sanitization.
4. **Source-Based Coverage** – Adds source-level coverage tracking via instrumentation flags.
5. **AFL++** – Requires deeper build pipeline modifications for forkserver support.
6. **Directed Fuzzer** – Adds instrumentation to prioritize specific paths (e.g., SARIF-guided).
7. **Artifact Extraction** – Post-processes binaries to recover source paths and auxiliary files.

Two mechanisms are used to implement builds:

- **OSS-Fuzz `infra/helper.py` with environment overrides**, where possible.

- **Direct Docker invocation** for custom modes not supported by OSS-Fuzz.

Direct invocation introduces risks of inconsistency, so HARNESSBUILDER includes validations and fallback checks to ensure reliability. The **artifact extraction mode**, for example, provides redundancy and fault isolation by mirroring a standard LibFuzzer build but maintaining it independently.

To avoid bottlenecks from sequential builds, **HARNESSBUILDER is deployed across all available CRS nodes**, allowing all instrumentation modes to be built **in parallel**.

This parallelization ensures that ATLANTIS-C can initialize fuzzers quickly at the beginning of each epoch. Each build result is stored on a shared volume and registered as a fuzzing task, making it accessible to the **TASKSCHEDULER**, **FUZZERMANAGER**, and participating fuzzers in the multi-fuzzer ensemble.

Time-based Task Scheduling

ATLANTIS-C uses a time-based scheduling mechanism to manage its fuzzing tasks efficiently. Instead of executing all fuzzing tasks in parallel, it divides execution into fixed time slices called “epochs,” typically lasting twenty minutes. Each fuzzing task corresponds to a specific combination of a fuzzing harness and instrumentation mode—such as libFuzzer, LibAFL, or AFL++. The time-based strategy allows ATLANTIS-C to respond dynamically to runtime events like SARIF announcements, reachability analysis updates, or fuzzer failures. During each epoch, the TASKSCHEDULER selects a set of fuzzing tasks equal to the number of available nodes, sends instructions to stop the currently running tasks that are not selected, and restarts containers with the new tasks. If the task queue is exhausted, it is refilled using weighted random sampling based on each task’s priority, ensuring high-priority tasks are scheduled more frequently. To avoid unnecessary overhead, epoch switches are skipped if the priorities are already well respected and no new updates have occurred. This system provides the flexibility needed to adapt to rapidly changing conditions during analysis, including the ability to implement robust fuzzer fallback mechanisms. When a fuzzing engine like LibAFL repeatedly fails or shows instability, it is dynamically replaced by a more stable alternative, such as AFL++ or libFuzzer, by setting its priority weight to zero and rescheduling a new task.

A critical component of ATLANTIS-C’s scheduling framework is its harness deprioritization strategy. In delta-mode challenges, where the target vulnerability lies within a specific code change, it is vital to focus compute resources on harnesses that can actually reach the affected code. However, a purely static pre-fuzzing analysis would be inefficient and potentially unreliable. Static tools like CodeQL can take hours to analyze large codebases and may generate false negatives—wrongly marking reachable code as unreachable. To balance precision with performance, ATLANTIS-C implements a hybrid approach. It starts with lightweight compilation-based analysis: each harness is compiled with aggressive optimization flags both before and after applying the patch. If the binaries remain unchanged, it is assumed that the harness cannot reach the patched code, and it is disabled.

Beyond this initial filtering, ATLANTIS-C integrates real-time reachability signals from two dynamic sources: BULLSEYE and ATLANTIS-SARIF. BULLSEYE is a directed fuzzer that targets specific code locations. If it reports a “target not found” error during instrumentation, this indicates that the harness cannot reach the target, and ATLANTIS-C removes it from future epochs. ATLANTIS-SARIF, on the other hand, builds a hybrid callgraph from both static analysis and dynamic execution monitoring. As fuzzing progresses and new function calls are observed, ATLANTIS-SARIF updates its callgraph and provides harness reachability results. These are stored in a shared directory, which ATLANTIS-C continuously monitors to deactivate harnesses that become irrelevant. This adaptive deprioritization system ensures that compute resources are consistently focused on the most promising paths to trigger vulnerabilities, optimizing both performance and precision.

Corpus Management

Because ATLANTIS-C runs numerous fuzzing tasks simultaneously across different Kubernetes nodes, and reuses the same harnesses across multiple epochs, the system must accumulate and share fuzzing progress in a distributed yet consistent way. Effective corpus management ensures that valuable inputs discovered during fuzzing are not lost, redundant inputs do not waste compute cycles, and every harness benefits from relevant coverage advancements.

The initial corpus used in ATLANTIS-C plays a crucial role in jump-starting fuzzing effectiveness. Rather than relying on arbitrary or random inputs, ATLANTIS-C sources a large, diverse, and semantically categorized seed corpus collected manually from over 400 open-source projects. These seeds are grouped into approximately 90 categories based on their semantic or structural domain—such as HTML documents, SQL queries, and JavaScript programs. This categorization enables ATLANTIS-C to match seeds to challenge projects in a targeted manner. Once a new challenge project is received, the CORPUSSELECTOR module launches an LLM-assisted two-step process. First, the LLM analyzes the structure and context of each fuzzing harness to infer expected input formats. Then, it performs a lightweight category-matching step to suggest relevant seed sets from the curated corpus. The resulting seed archives are loaded into the fuzzer's input directory, providing it with a rich and domain-specific starting point. Although loading large corpora may incur some short-term overhead, this is a worthwhile tradeoff that ultimately improves fuzzing speed and coverage discovery.

As fuzzing progresses, seeds are generated from several sources including the initial corpus, LLM-based generators, directed fuzzers like BULLSEYE, and results shared from related systems such as ATLANTIS-Multilang and ATLANTIS-SARIF. ATLANTIS-C supports three distinct methods for seed distribution and ingestion. The first method involves the use of Kafka and the B module. The SEEDSCOLLECTOR monitors designated seed directories and, upon detecting new inputs, sends them via Kafka to the ENSEMBLER. The ENSEMBLER deduplicates these seeds and pushes them to appropriate fuzzers, with LibAFL instrumentation including a Kafka consumer for direct ingestion. The second method uses filesystem-based seed sharing: SEEDSCOLLECTOR periodically compresses corpus directories and stores them in a shared location. These compressed corpora represent the deduplicated, accumulated seeds and can be reused by subsequent fuzzing tasks across different epochs. The third method involves low-latency seed injection using ZeroMQ, which is ideal for high-throughput seed generators such as DEEPGENERATOR. In this setup, fuzzers include a background thread that receives and buffers incoming seeds via ZeroMQ and consumes them during mutation stages.

The ENSEMBLER plays a central role in maintaining a unified view of the corpus across the system. It acts as the central testing hub for all newly generated or discovered seeds, assessing them for novelty, crashes, or timeout behavior. ATLANTIS-C leverages libFuzzer's "merge" mode, which efficiently incorporates new seeds into an existing corpus by testing whether they reach any code not yet covered. The ENSEMBLER maintains a master corpus directory that represents the total known coverage achieved by ATLANTIS-C. To avoid concurrency issues when multiple libFuzzer processes run in parallel, the ENSEMBLER creates symbolic link-based snapshots of the corpus before each merge operation, ensuring consistency and preventing seed loss. When new seeds are added to the master corpus, they are dispatched to both fuzzers and ATLANTIS-Multilang to expand test coverage.

Timeout bugs are within scope for the AIxCC competition, meaning inputs that exceed a configurable execution time—typically 25 seconds—can be considered scorable. However, the ENSEMBLER must balance the need to test for long-running inputs against the need to maintain high throughput. To resolve this, seeds are initially tested with a short per-seed timeout of one second. Any seeds that hit this timeout are placed in a "slow-seeds queue" for deferred processing. If a seed in this queue eventually surpasses the scorable timeout threshold, it is submitted to CP-MANAGER as a valid PoV. A per-batch timeout also guards against collections of slow seeds overwhelming the system, ensuring that low-value seeds do not consume disproportionate resources.

LLM Components

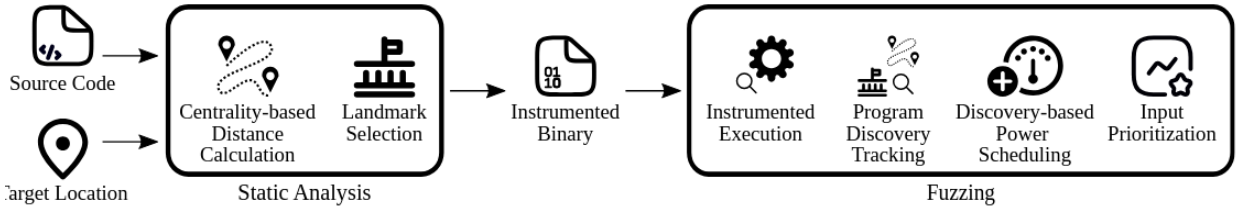
They built a research-oriented library called LIBAGENTS. This ibrary contains a finite-state machine to split a query into sub-queries, external plugins and LLM tools, and an answer evaluator to assess answer quality. Users enable the desired plugins and provide a query; the library then iterates over sub-queries and calls external plugins to obtain and evaluate answer

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

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.

DEEPGENERATOR is an LLM-based agent designed to dynamically generate fuzzer scripts tailored to a given challenge project (CP). It operates through a main engine that invokes multiple specialized LLM agents—namely *harness-analysis*, *delta-scan-analysis*, and *self-evolving*—to automate the script generation process. Upon receiving a CP, the engine initiates these agents to produce Python-based fuzzer scripts, which are then executed in a continuous loop to generate testcases. All generated testcases are streamed to fuzzers via ZeroMQ for immediate consumption. The *harness-analysis* agent inspects the structure of the target harness and associated source code to infer input formats and generate scripts accordingly. For delta-scan challenges, the *delta-scan-analysis* agent performs diff analysis on commits or pull requests, identifies suspicious code locations, and generates focused fuzzer scripts targeting those areas. To improve upon initial outputs, the *self-evolving* agent evaluates previously generated scripts using metrics such as coverage and error rate, and iteratively refines them by prompting the LLM for improved fuzzing strategies. DEEPGENERATOR is also integrated into ATLANTIS-Java, enabling similar functionality for Java-based targets.



The **LLM-Augmented Mutator** is a microservice integrated into ATLANTIS-C's fuzzing system to address scenarios where fuzzers become stuck—typically due to deep execution paths, complex program logic, or inputs requiring semantically valid structures that standard mutation strategies cannot generate. When a fuzzer exhibits signs of stagnation—detected by the absence of new seeds in its output directory for more than two minutes—the mutator activates and selects a recently added seed for analysis. It then collects detailed execution information for that seed, including the complete call stack, source-level execution traces,

and variable values at the entry and exit points of the deepest executed function. This runtime context, along with current coverage data, is packaged into a prompt and sent to an LLM to guide the generation of semantically informed mutations. These LLM-derived inputs have proven effective at unlocking new branches and resuming meaningful fuzzing activity. By leveraging deep program understanding and dynamic runtime insight, the LLM-Augmented Mutator enhances the ability of fuzzers to explore hard-to-reach areas of the codebase.

Bullseye: Directed Fuzzing

BULLSEYE is a custom-directed graybox fuzzer (DGF) developed to overcome limitations found in traditional DGFs, which often become trapped in local minima by aggressively pruning paths that appear irrelevant. While these approaches can reach target locations quickly, they risk missing alternative execution paths that might expose additional or subtler vulnerabilities. BULLSEYE addresses this by combining static analysis with dynamic runtime feedback to diversify path exploration while still maintaining a focus on target-oriented fuzzing. For target selection, it uses SARIF-reported locations in full-scan challenges and an LLM-assisted strategy to identify likely relevant lines in delta-scan mode. Static analysis is performed via an LLVM pass, computing a distance metric based on closeness centrality within the interprocedural control flow graph (ICFG) and selecting a set of “landmark” locations throughout the code. These landmarks are instrumented into the binary and tracked at runtime to measure a discovery metric, quantifying how much of the target-related program space has been explored.

During fuzzing, BULLSEYE integrates both the structural distance and discovery metrics into its power scheduler and input prioritization logic, favoring seeds that both approach the target and traverse novel regions of code. It is implemented with over 1,500 lines of code, split between the LLVM-based static analysis module and a modified version of AFL++ that adds roughly 500 lines for runtime integration. The system also benefits from AFL++'s persistent mode and shared memory fuzzing for reduced overhead and improved throughput. In evaluation against AFL++ across 11 AIxCC challenge targets, BULLSEYE achieved superior results in 7 out of 11 cases for time-to-exposure (TTE) and in 9 out of 11 for unique crash discovery (UC). Particularly notable was its success in target #6, where it exposed a bug in 4 out of 5 runs, whereas AFL++ failed entirely. These results demonstrate that BULLSEYE's combination of static and dynamic strategies provides meaningful advantages in complex fuzzing scenarios.

Atlantis-Java

ATLANTIS-Java is focusing primarily on Java CPV detection, basically meaning its goal is to observe as many Java vulnerabilities that are sink-centered security vulnerabilities that come from unsafe usage of sensitive APIs. This is sink-aware to address Java VR since Java typically does not have any memory vulnerabilities.

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

Their motivation for Java VR comes for the unsafe usage of sink APIs and their detection process can be modeled as a sink-centered exploration and exploitation workflow. The vulnerability lies in the OS command injection comment if a specific condition met. Line 20 shows taht the constructor serves as a sink API, a security-sensitive operation where attacker-controllable arguments.

There are two distinct phases in this sink-centered perspective:

- Sink Exploration (Lines 3-8, 12-19): This is the constraints the fuzzer has to reach to see/reach the sinkpoint, including the presence of header x-evil-backdoor with a value matching the SHA-256 hash “breaking the law” and non-empty command validation.

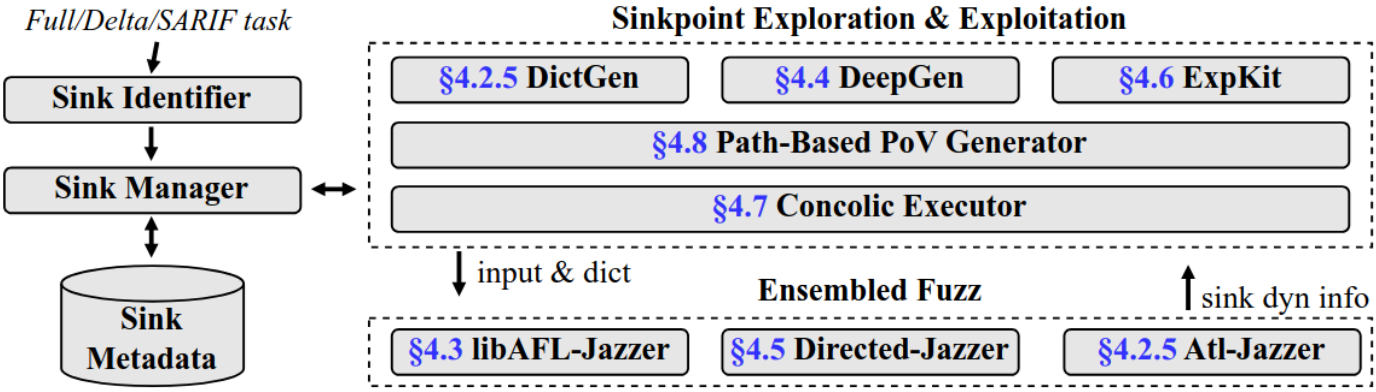


Figure 5: Overview of ATLANTIS-Java

- Sink exploitation (Lines 20-23): The fuzzer has to generate inputs that actually trigger the vuln, by setting cmds[0] to “jazze” in this case.

This two pattern approach is bog standard for Java vulnerabilities. Security issues are usually from dangerous API calls such as file operations, deserialization, command execution, network access, and template rendering. For each dangerous API call you need to craft malicious paths that bypass sanitization and validation.

(UNIQUE) Existing Java fuzzing solutions are mostly inherited from C/C++ fuzzers, and only are coverage-centered as opposed to knowing sink knowledge, thus treating all code paths equally and missing opportunities to generate actually interesting inputs. Atlantis-Java’s design is 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.

The Atlantis-Java design is centered on sinkpoints for Java vulnerability detection. Atlantis-Java (AJ) has an ensemble fuzzing pipeline that does input generation and execution. In this pipeline as well, the system has sink analysis on targets to identify relevant sinkpoints by doing dynamic/static program analysis.

Once it finds a relevant sinkpoint, it goes through the two phases we talked above previous, we use sinkpoint exploration to discover paths to the sink, and exploitation to actually generate inputs that trigger vulnerabilities at the sink. The components that are sink aware generate inputs/dictionaries that feed back into the pipeline, the pipeline provides dynamic execution feedback for sink analysis. The system maintains “beep seeds” that reach sinkpoints which are the start for the sinkpoint exploitation.

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′)
```

The Sinkpoint-Aware Fuzzing Loop is a fuzzing algorithm that continuously mutates inputs to a target program and monitors execution for new behaviors, crashes, or the triggering of predefined "sinkpoints" (security-critical locations). Inputs reaching sinkpoints are saved for further targeted analysis in an exploitation phase. This approach prioritizes deeper exploration of potentially vulnerable code paths.

ATLANTIS-Java introduces a unified, **sinkpoint-centered framework** for handling all AIxCC competition task types—**Full Mode**, **Diff Mode**, and **SARIF-based tasks**—by transforming them into lists of concrete sinkpoints. This transformation allows the system to treat every task through the same core mechanism: identifying, tracking, and exploiting calls to sensitive or security-critical APIs within the target Java application. In Full Mode, ATLANTIS targets all sinkpoints in the entire codebase; in Diff Mode, it narrows the focus to those affected by recent code changes; and in SARIF tasks, it directly targets sinkpoints mentioned in vulnerability reports. This abstraction enables ATLANTIS-Java to apply a consistent fuzzing and exploitation pipeline, regardless of the input task format, making the system both flexible and scalable across different scenarios.

At the heart of this approach is the **Sinkpoint-Aware Fuzzing Loop** (Algorithm 1), which extends traditional coverage-guided fuzzing by treating **sinkpoint reachability as a primary feedback signal**, alongside coverage and crashes. The system uses a customized Java agent instrumentation—**CodeMarkerInstrumentation**—integrated into a modified Jazzer fuzzer (Atl-Jazzer), which throws a `CodeMarkerHitEvent` whenever a sinkpoint is reached. This event includes a full stack trace and runtime context, which are persisted for downstream analysis. When a sinkpoint is hit, the input responsible is designated a **"beep seed"** and passed into a separate **exploitation phase**, which is explicitly decoupled from the exploration loop. This separation allows ATLANTIS-Java to apply advanced reasoning and constraint-solving techniques on real execution contexts (e.g., call stacks and input blobs), improving its ability to synthesize valid exploits—especially in cases where exploitation depends on subtle runtime behaviors such as bypassing sanitization or satisfying deep program invariants.

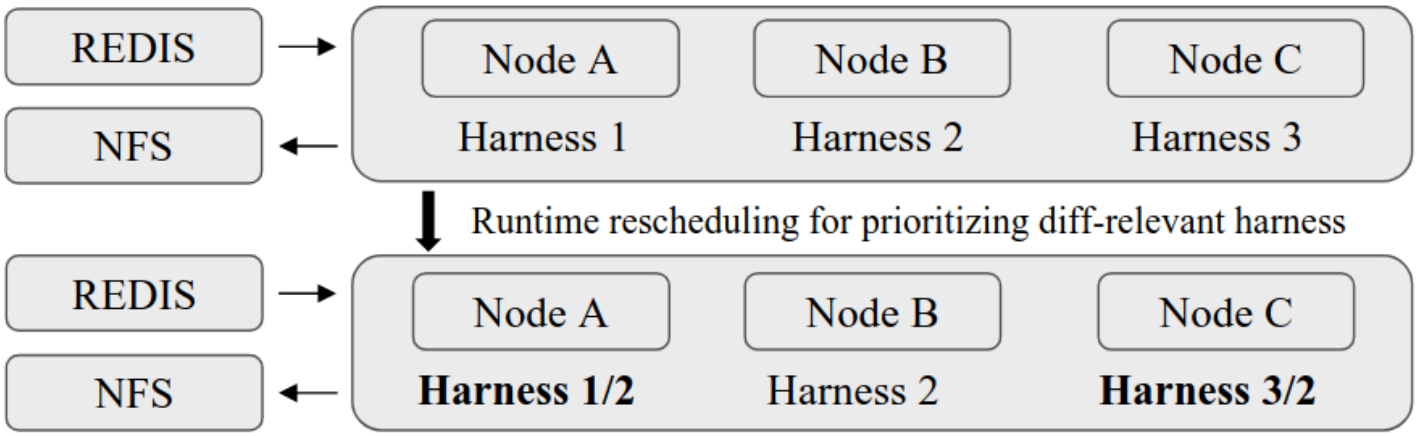


Figure 6: Runtime Rescheduling in ATLANTIS-Java for Diff Tasks

ATLANTIS-Java's **Infrastructure II – Ensemble Fuzzing** introduces a collaborative fuzzing strategy that merges the strengths of multiple fuzzing instances and input generation techniques to improve bug discovery. It performs corpus merging by collecting inputs from different fuzzers, deduplicating them based on code coverage, and redistributing the refined corpus to all participants. This enables individual fuzzers to benefit from each other's findings, improving coverage and depth. Beyond traditional fuzzers, the ensemble also integrates inputs generated by non-fuzzing components like large language models (LLMs) and concolic execution engines, bridging static and dynamic techniques under a single workflow. Additionally, the ensemble layer synchronizes sinkpoint metadata—such as whether a sink has been reached or exploited—ensuring consistency across fuzzing instances and enabling shared progress toward security-critical targets.

Infrastructure III – Sinkpoint Identification & Management forms the backbone of the sinkpoint-centric architecture. Using static analysis, ATLANTIS-Java locates calls to sensitive APIs by leveraging an extended sinkpoint list curated from Jazzer defaults, vulnerability benchmarks, academic research, and static analyzers. Sinkpoints are tracked in a central metadata store, where each is labeled with status markers like "unreached," "reached," "exploited," or "unexploitable." This metadata includes associated call graph data, relevant beep seeds, and any exploit attempts. Such granularity allows the system to intelligently allocate resources—focusing exploration on unreached or high-priority sinkpoints (e.g., those from SARIF reports) and avoiding waste on unproductive paths. All components, whether fuzzers or exploit generators, access this shared metadata to ensure they operate on the most up-to-date and strategically valuable information.

In **Infrastructure IV – Distributed Design**, ATLANTIS-Java employs a scalable and fault-tolerant architecture based on standard technologies like Redis and NFS. In Full Mode, the system assigns each harness to a unique node, enabling parallel, independent fuzzing across the entire codebase. For Diff Mode tasks, ATLANTIS performs **runtime rescheduling**—after either a fixed time period or once static analysis results converge—to redistribute nodes toward diff-relevant harnesses containing critical changes. This dynamic resource reallocation ensures that high-risk or newly modified code receives more fuzzing attention. When multiple nodes are reassigned to the same harness, the system synchronizes sinkpoint metadata

across nodes to avoid redundant work. The lightweight, resilient infrastructure supports seamless recovery from failures and is simple to deploy across various task types, enhancing both robustness and adaptability in competitive scenarios.

Infrastructure V – Component Overview highlights ATLANTIS-Java's integration of a wide array of specialized tools for both exploration and exploitation phases. These include variants of Jazzer (Directed, LibAFL-based, Atl-Jazzer), concolic executors, path-based PoV generators, and exploit synthesis engines like ExpKit. Some components are dual-purpose, supporting both sinkpoint discovery and exploit crafting. Tools like **DictGen** assist fuzzing by generating dictionaries for input mutation, while **Atl-Jazzer** enhances Jazzer by adding beep seed tracking and other improvements. All these tools plug into the shared infrastructure, particularly the sinkpoint manager and ensemble layer, enabling coordinated behavior across tasks. This tool ecosystem ensures that ATLANTIS-Java remains effective at all stages of vulnerability analysis, from initial path discovery to exploit generation, within its sinkpoint-centered framework.

LibAFL-Based Jazzer

The LibAFL based Jazzer component is designed to enhance mutation diversity beyond Jazzer's builtin mutators for generally improved sinkpoint exploration. While Jazzer provides libFuzzer mutations, it limits exploration breadth. Them integrating LibAFL's advanced mutation algorithms and scheduling strategies, helps to discover execution paths that Jazzer could miss, contributing to more sinkpoints.

They integrated into the LibAFL-libfuzzer project that is a drop-in replacement architecture, addressing slight technical challenges.

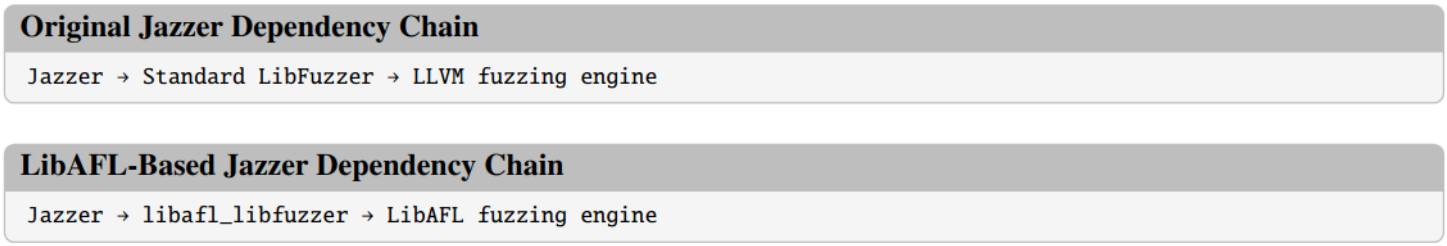


Figure 7: Jazzer Dependency Chain Comparison

The first issue was finding detection and lifecycle control, Java fuzzing requires specialized handling of findings and fuzzer lifecycle events that differ from binary fuzzing. Their solution to this challenge was to implement custom observers (JazzerFindingObserver and JazzerFuzzerStoppingObserver) that hook into Jazzer's __jazzer_set_death_callback mechanism. Allowing for crash artifact dumping when Java-specific vulnerabilites are detected.

The second issue is coverage feedback integration, the Jazzer bytecode instrumentation passes program counter information to sanitizer function for precise coveragetracking. They extended LibAFL's coverage infra with PC-aware variants (__sanitizer_cov_trace_cmp4_with_pc) that extracts and forwards this information to LibAFL for their feedback mechanism.

The third issue is that Jazzer relies on specialized string comparison behaviors. They implemented Jazzer-specific hooks including (__sanitizer_weak_hook_strstr and __sanitizer_weak_hook_compare_bytes) that integrates with LibAFL's comparison tracking.

The fourth issue is leveraging LibAFL's mutation diversity, they disabled Jazzer's custom mutation through runtime flags, which allow LibAFL's schedules and mutators to control the fuzzing process exclusively.

The last issue is that, in their testing the LibAFL-based jazzer can show different coverage results compared to standard jazzer in certain projects, showing how it is complementary to the reglar jazzer. This is probably from how LibAFL mutation scheduler prioitizes unexplored code regions. Our current LibAFL based jazzer lacks support for value profile feedback, which Jazzer regular uses. This shows weaker exploitation capabilities for vulnerabilities requiring specific value constraints.

DeepGenerator

This is the high-throughput framework that leverages LLM agents to generate seed mutation/generation scripts for fuzzers, thus making a diverse amount of seeds. The framework shows collaboration between the Atlantis-C and Atlantis-Java teams, by trying to address the limitations of traditional mutation strategies through context aware seed generation. The complete system consists of three core components: LibAgents, LibDeepGen, and customized fuzzers with Out of Fuzzer (OOF) mutation support.

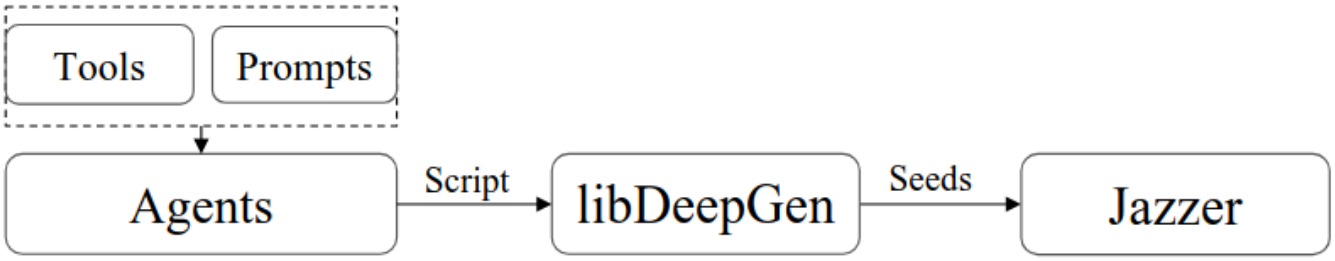


Figure 8: Overview of DEEPGENERATOR

DeepGenerator employes a straightforward architecture where LibAgents serves as the intelligent generation layer. This component uses a deep research framework capable of analyzing target prorams and generating context-aware Python mutation scripts. Libagents supports tools like common CMD utils, wrappers for code agents like Claude Code. The deep research capability allows LibAgents to explore codebase, understand API patterns and generate scripts that produce meaningful inputs.

The core runtime component, LibDeepGen manages the execution and scheduling of agent-generated scripts with highly optimized performance. It incorporates atomic operation-based buffers for script task scheduling and a combination of shared memory with ZeroMQ acting as seed storage and dispatch. This architecture enables extreme throughput, meaning thoughts of script switches per second and generating hundreds of thousands of seeds. When LibDeepGen executes scripts provided by LibAgents, the seeds are sent to fuzzers through shared memory coordinates transmitted via ZeroMQ messages.

How they integrated with their existing fuzzing infra, DeepGenerator requires fuzzers to support OOFMutator functionality. This is achieved by adding a ZeroMQ Client to the fuzzer's mutation engine, which enables a

batch retrieval of seeds that are used as mutation inputs. Atlantis-Java's ATL-Jazzer implements full OOF/Mutator support, which allows DeepGenerator's fast seed generation pipeline to actually be performant.

DeepGenerator in AtlantisJava, They primarily focused on initial corpus generation rather than its full potential for continuous mutation. For each harness, Atlantis-Java consolidates relevant info including harness code/task descriptions into prompts for LibAgents. LibAgent's deep research framework actively explores the CP codebase for harness-relevant context and utilizes existing test cases, API documentation, code patterns to understand harness generation. This allows for python scripts to be generated that achieve the goal of creating Python scripts that are diverse enough, outperforming random/syntax-unaware strategies. Atlantis-Java iteratively tells DeepGenerator to generate multiple python scripts, with a fixed quantity of seeds. DeepGenerator will filter out ineffective scripts so limiting to only high quality seeds.

DeepGenerator is remarkable in starting fuzzing campaigns. Coverage increases of 2952%, with over 20% of projects having improvements greater than 40%. These results highlight DEEPGENERATOR's ability to quickly explore diverse program states.

Sinkpoint Directed Fuzzing

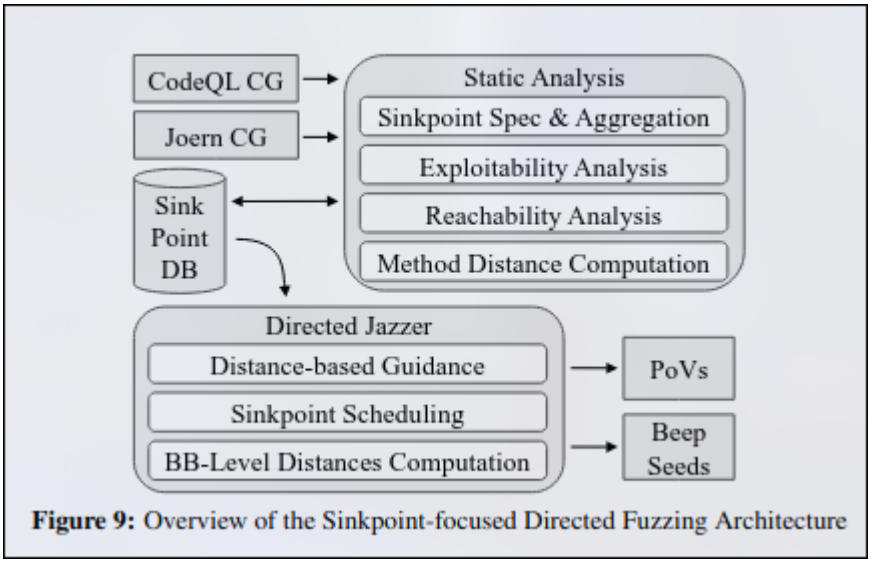


Figure 9: Overview of the Sinkpoint-focused Directed Fuzzing Architecture

This component combines static analysis and runtime feedback to guide fuzzing toward security-critical sinkpoints in Java code. Even though it's mainly designed to reach sinkpoints, it works just as well for exploiting them once hit.

The system has two main parts:

- Static analysis pipeline built using CodeQL and Soot, and
- A modified Jazzer that uses precomputed distances as fuzzing guidance.

Here's how it works: The system takes in call graphs (from other CRS modules) and sinkpoint data. Most of the heavy analysis happens before fuzzing starts, which saves time. That's because computing distances to targets (sinkpoints) during fuzzing is expensive. So instead of recalculating ICFGs and walking through graphs every time, ATLANTIS-Java precomputes function-level distances, and only does lightweight basic-block distance calculation at runtime.

During fuzzing, the system smartly schedules which sinkpoints to focus on next based on how reachable, exploitable, or competition-relevant they are. This makes directed fuzzing both efficient and practical in large Java programs.

Since the vast majority of Java vulnerabilities come from the unsafe usage of sink APIs, and their detection process can be modeled as sink-centered exploration and exploitation workflow. While Jazzer has a list of such APIs that it sanitizes, there are other APIs that are likely to trigger the sanitizer for the following clases:

-java.math.BigDecimal: BigDecimal can cause a DoS

-java.net.URL and java.net.URI: Networking APIs for SSRF

-javax.validation.Validator: May lead to a expression language injection -> RCE

-javax.xml.parsers.SAXParser: Parsing an XML -> SSRF

-org.apache.batik.transcoder.TranscoderInput: SVG Stream -> SSRF

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`.

Instead of analyzing every library dependency, their CRS extends the list of sink APIs to only scan the challenge problem code instead of all libraries. The strategic advantage of this approach helps with resource constraints. Analyzing this code + dependencies would take forever with Soot, which does not fit within our time the CRS has to analyze the challenge problem without starting the fuzzer. They can skip the analysis of the dependencies to reduce the analysis time to a matter of minutes in their benchmark set, leaving most of the analysis time for the directed fuzzer to run.

Atlantis-Multilang

https://team-atlanta.github.io/blog/post-crs-multilang/

6.1 Overview

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

Goal Overview

G1: Support fuzzing CPs written in any language (language agnostic)

CPs are in the OSS-Fuzz project format, implemented in various languages, these need to be supported regardless. Their approach is designed to actually work with any language in the OSS-Fuzz project.

language

Programming language the project is written in. Values you can specify include:

- `c`
- `c++`
- `go`
- `rust`
- `python`
- `jvm` (Java, Kotlin, Scala and other JVM-based languages)
- `swift`
- `javascript`

G2: Improve fuzzing performance by using LLMs

Most research had been on generating/mutating input more effectively. Tools limited to specific programming languages. Atlantis-Multilang built modules to utilize various levels of LLM usage.

G3: Optimize fuzzing pipelines and development

Built for scale, minimizing overhead, synchronizing among multiple fuzzing processes. AM must support the target languages, again built with modularity to support this.

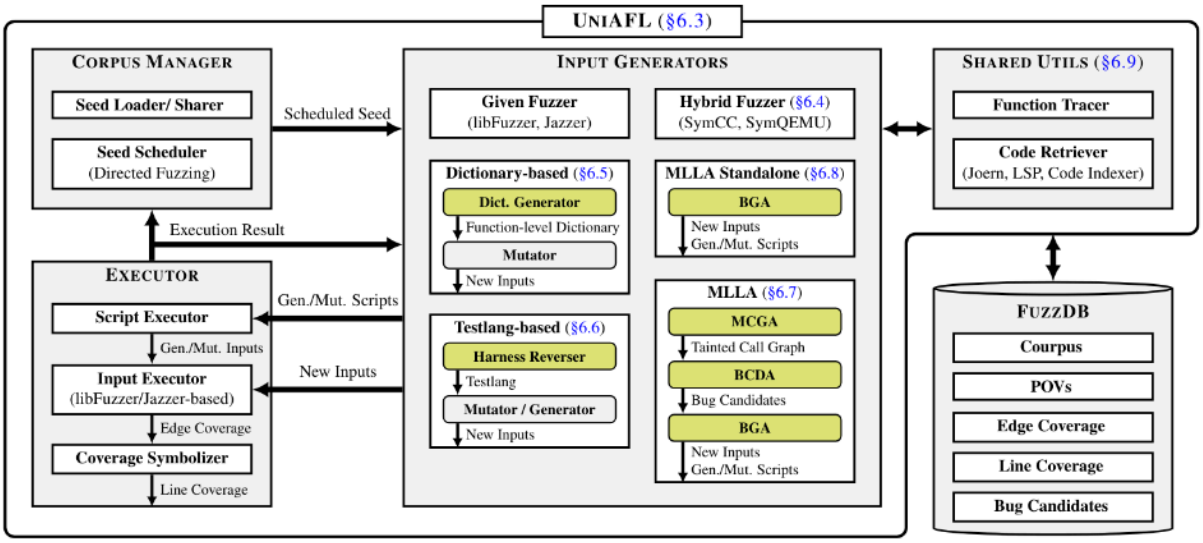


Figure 23: The overview of ATLANTIS-Multilang. The green boxes indicate LLM-powered modules.

Two components are UNIAFL and FUZZDB.

UNIAFL

UNIAFL is responsible for fuzzing and achieving the goals above.

FUZZDB

FUZZDB is used in storing intermediate outputs (corpus, POVs, edge coverage, line coverage, and bug candidates) produced by UNIAFL.

UNIAFL follows the same workflow as traditional fuzzers.

1. **Corpus Manager** picks a seed input to start fuzzing, especially directed fuzzing toward bug candidates, which are intermediate results of LLM-powered modules in UniAFL.
2. **Input Generators** then create new inputs either by mutating the seed or generating fresh inputs from scratch. Some of these input generators go a step further: instead of directly producing inputs, they generate Python scripts that generate or mutate inputs, giving us more flexibility in creating structured test cases.
3. **Script Executor** runs those Python scripts, turning them into actual inputs.
4. **Input Executor** feeds those inputs into the target harness and collects execution results along with coverage data.
5. **Coverage Symbolizer** converts raw edge coverage into line coverage — a crucial step, since the LLM-powered modules rely on line-level feedback because LLM cannot understand basic block addresses in raw coverage data.
6. Finally, based on execution results and coverage data, Corpus Manager updates the corpus and schedulers to guide directed fuzzing.

Input generators

The core of UNIAFL is not only the infra but also in its 6 input generation modules. Each module is built for a different amount of LLM dependence (“to ensure resilience when LLM usage is constrained, either due to budget limitations or infrastructure issues”).

- No LLMs
  - Given Fuzzer just runs the target harness
  - Hybrid Fuzzer performs hybrid fuzzing based on concolic execution (LLMs used to symbolically model functions)
- Limited LLM Usage
  - Dictionary-Based uses an LLM to infer a dictionary for a given function
  - Testlang-Based uses an LLM to decipher the input format for a harness and express it in TESTLANG, then generates and mutates based off of that
  - MLLA-Standalone uses a LLM to generate python scripts directly.
- Full LLM Usage
  - MLLA uses LLMs the most in order to derive tainted call graphs, identify bug candidates, and generate inputs/scripts

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.

Given fuzzer used as the baseline, on its own it struggled though

`The real breakthroughs came when the other input generators kicked in. They consistently provided meaningful new inputs that helped UniAFL break out of plateaus whenever the given fuzzer got stuck`

(UNIQUE)

6.2 Final Comp Results

71.2% of all verified PoVs (retroactively 69.2%)

Table 11 shows LLM usage and affected harnesses, yet it doesn't detail the incremental contributions(the generated corpora, line coverage, etc.) and only shows the final PoV.

Each submodule contributed to progressively expanding coverage.

- Given Corpus with 7.6% representing the initial corpus, then Given Fuzzer with 49.2%. "proving that quality seeds and robust fuzzing infrastructure remain the most reliable vulnerability discovery methods under competitive pressure"
- Hybrid fuzzer added 1.7% through offline LLM-derived symbolic models
    - without LLM runtime overhead
- Dictionary based mutations achieved 0.0%
    - "simple function-level enhancements require deeper contextual understanding to succeed"
- Testlang based generation got 6.8%
    - LLM pattern recognition on structured input formats
- MLLA Contributed 5.9%
    - In lieu of significant compute overhead, comprehensive LLM integration/discovering/complex vulns

The Shared Corpus row shows 23 PoVs originating from Atlantis-C or Java. Basically they fucked up where C/Java store their PoVs and accidentally put them in the seeds of multilang. Multilang inadvertently reported these duplicates. Lol

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

How UNIAFL supports fuzzing CPs in any language **G1**, and optimizing its fuzzing pipeline and development **G3**

Microservice-based Fuzzing, inspired by μFuzz.

- Running on separate cores requires frequent synchronization
- Maintaining the fuzzing state within each process
- Significant overhead
- Setup is not failsafe, the process terminates and core becomes idle

- Each runs as a standalone process with multiple threads, maintaining their own memory reducing costly synchronization.
- If one input generator crashes the other remain without disrupted other workflows
- Each process talks through shared memory and controls thread execution through semaphores

Input Executor

- The core component supporting CPs written in any language
- Protocol that delivers inputs for execution via shared memory to reduce overhead
- Existing fuzzers, libFuzzer, and Jazzer were modified to serve as a foundation for target harnesses.
- Protocol is target-agnostic, UNIAFL seamlessly supports

Script Executor

- Repeatedly executes python scripts for mutating and generating inputs
- Leverages shmem
    - Delivers to executor

Coverage Symbolizer

- Line coverage is needed because LLMs cannot understand edge coverage (bb addresses).
    - Most fuzzers only produce edge
- Gathers line coverage for all seed and all PoVs which is
    - JSON format for all functions, lines, and files
- Relies on language specific instrumentation to gather line coverage

Coverage Symbolizer for C

- Uses coverage build feature in OSS-Fuzz
    - Uses coverage sanitizer in LLVM
    - Some CPs cannot be compiled with `coverage` option
        - Then recover from edge coverage using Llvm-cov
            - Coverage build option is more precise than llvm-cov
- Coverage build option cant capture coverage for Povs due to the fact that “they often crash the target harness before line coverage is produced”
- Attach sanitizers to those built with cov
- LLVM was patched to hook internal functions, ensuring data flush even in the event of a crash or abort
    - Used to ensure POV coverage to deprioritize already triggered bug candidates in directed fuzzing

Jazzer

- Foundation fuzzer for java in OSS-Fuzz
- Uses JaCoCo to gain edge coverage and convert to line coverage

Directed Fuzzing

- “To better focus on generating POVs”... UNIAFL uses directed fuzzing for MLLA bug candidates
- Target set dynamically updates
    - Updates with MMLA
- Existing directed fuzzers typically rely on custom instrumentation
    - No dynamic targeting support
    - Some cannot handle multiple simultaneous targets
    - Custom instrumentation may not work with OSS-Fuzz CPs
- To address the limitations!
    - Seeds are given scores based on line coverage
        - Seeds associated with bug candidates receive higher scores
        - This is effective because MLLA contains key lines that must be executed
        - “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”
    - Seeds are scheduled with a weighted random strategy corresponding to the score
        - Increasing likelihood of triggering the vuln
- Directed fuzzing isn’t always ideal
    - If Mlla returns incorrect candidates
    - Risk of overfitting
        - Need mixed seed-selection strategy
            - 25% random
            - 25% random amongst previously worked seed
            - 50% selected based on score weight

6.4 Hybrid Fuzzing

Hybrid fuzzer based on concolic execution, used to explore “tight and complex branch conditions”

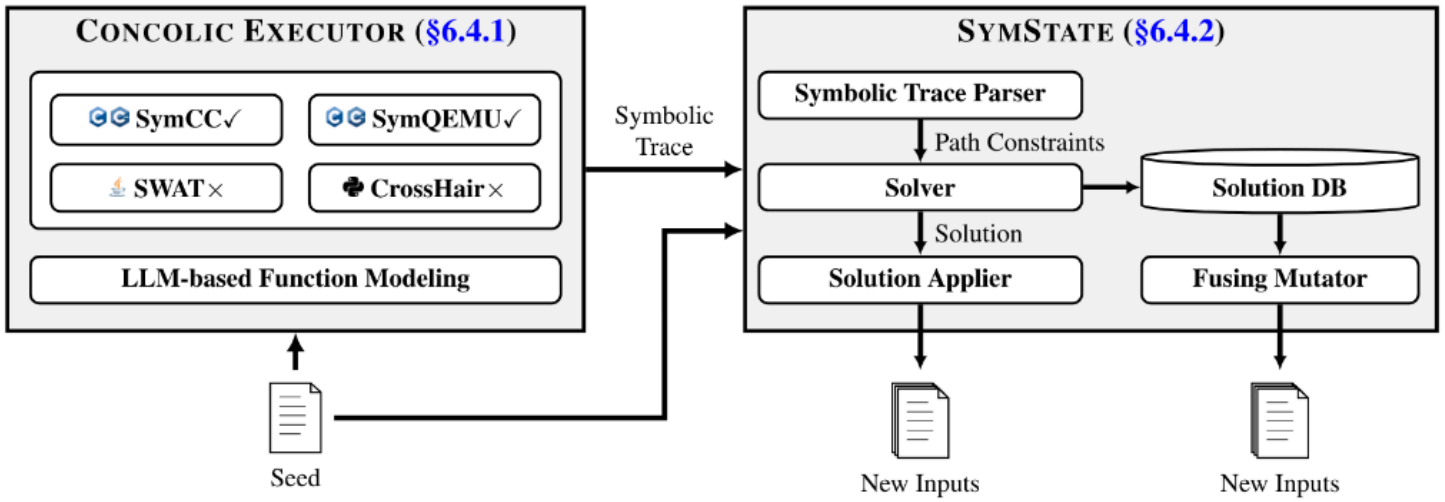


Figure 24: The overview of our HYBRID FUZZER. Check-marked boxes represent executors that were integrated into ATLANTIS-multilang for the final submission.

The concolic executor executes the instrumented target and produces a symbolic trace. Symstate takes the trace and produces new inputs.

Built on existing frameworks SymCC, SymQEMU, SWAT, CrossHair. Difficulties with limited support for functions in external libraries, and integration into a specific framework due to path constraint differences (how?).

- **C1** External function modeling, out-of-instrumentation
  - SymCC only models 30 libc functions
    - Literally ommitts strcmp and recv
    - Developed an offline LLM-powered module that systematically identifies and models OOI functions!
- **C2** Multilanguage support
  - Integrating multiple hybrid fuzzers is difficult due to variations in path constraint representations
    - I.e. SymCC maintains path constraints in memory as Z3_ast structs, SWAT uses SMT-LIB2 strings
  - To integrate the hybrid fuzzer was decoupled into the executes and the solver!
    - The executor is language dependent
    - The SymState isn't restricted
      - Therefore they can all share the same symstate module

Concolic Executor

- Design of the concolic executor
  - How they address the challenge of C1
  - How they made the CE robust to compilation errors, by modifying SymCC toolchain

- Offline LLM-based function modeling
    1. Three-way differential analysis used to pinpoint OOI functions and the values they "over-concretize" (values that should be symbolic but rendered concrete)
    2. LLM prompt to model using python
    3. Verify by repeating differential analysis with model

> Differential analysis:

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 $\mathcal{V}$

```
function DetectOOIFunctions(P, I_A, I_B)
    F ← ∅, V ← ∅
    Φ^A ← ConcolicExecute(P, I_A)
    Φ^A' ← ConcolicExecute(P, I_A)
    Φ^B ← ConcolicExecute(P, I_B)
    for i = 0 to |Φ^A| − 1 do
        (e_i^A, t_i^A) ← GetIthExprAndTaken(Φ^A, i)
        (e_i^B, t_i^B) ← GetIthExprAndTaken(Φ^B, i)
        (e_i^A', t_i^A') ← GetIthExprAndTaken(Φ^A', i)
        F ← F ∪ GetCalledFunctions(e_i^A)
        if t_i^A ≠ t_i^B then
            break
        if e_i^A ≠ e_i^A' then
            continue
        if e_i^A ≠ e_i^B then
            V ← V ∪ GetDifferentValues(e_i^A, e_i^B)
            break
    return F, V
```

- Model basically ConcolicExecutes, and checks each expression to see if they differ, then break.
    - GetDifferentValues only if the two different input's outputs differ.
    - Recv is an OOI cause symbolic expressions are different for inputs "AAAA" and "BBBB"
    - Some symbolic expressions may stem from non-deterministic behavior.
        - Cases are filtered with `(e^A_i != e^A'_i )`

- Symbolic trace
    - Path constraints generated during a single exec
    - Path constraints are tuples
        - Boolean symbolic expression, and t_is concrete evaluation
        - Leaf nodes are symbolic variables or concrete values
    - Based on this they developed the differential analysis algorithm above

- Loops until
    1. either branches are fully exhausted line 6
    2. Control flow diverges in line 11
    3. OOI is detected line 15
    4. Catch for non-deterministic compares in same input line 13
- “We consider the program to be correctly instrumented if there are no over-concretized values (V = ∅).”

After these are detected we move to

> Model OOI functions by prompting the LLM:

- Information gathered is presented and requested to the LLM, to model the functions using python code.
- Includes list of OOI functions and call sites
- Source location of over-concretized values and their values, and the problematic branch site

Example of recv:

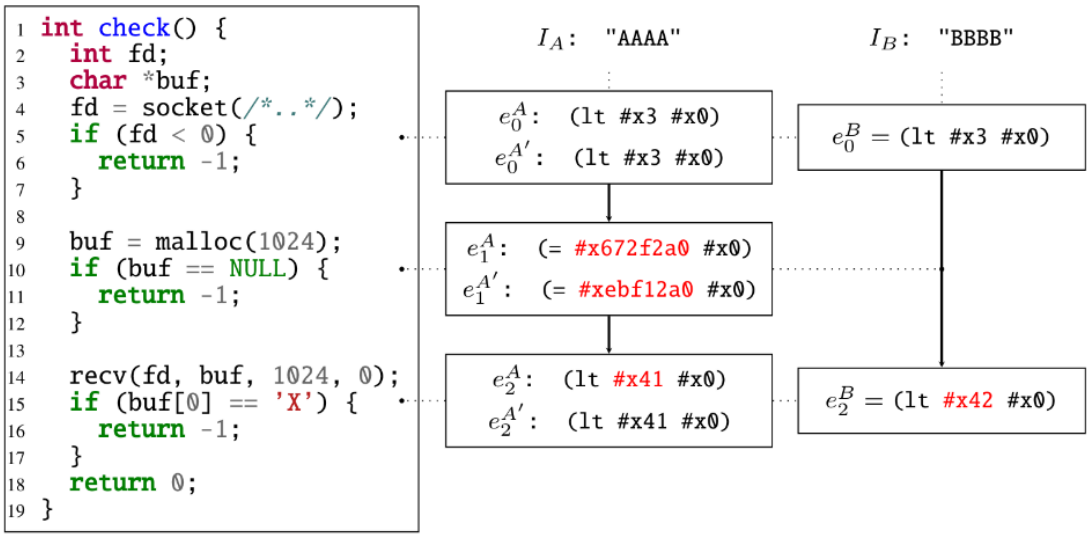


Figure 25: Three cases handled by our differential analysis algorithm within a single function.

This exemplifies the three cases ^

The process of modeling:

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.

> Model Verification

- Differential analysis is repeated to verify the model.
- Considered incorrect if over concretized values are still present.
- If the module successfully eliminates all, the model is added to the program's instrumentation
- P2 and P3 are repeated until the differential analysis terminates with an empty set of over-concretized values (V=0)

> Compilation Robustness

- Made to be robust against compilation failures
  - "Let's test OSS-FUZZ!"
- 130/411 compiled C/C++ projects in OSS-Fuzz when compiled failed using SymCC

    - Fixed issues and compiled 409 of the 411
- Implemented further SymQEMU fallback mechanism for SymCC failed compiles.
    - Executes the libfuzzer harness within SymQEMU (with worse performance due to TCG instrumentation)
    - Harness in fuzzing mode without seed directory, and hooked LLVMFuzzerTestOneInput
        - Inject input bytes directly into memory
        - Reuse TCG instrumentation
    - SymQEMU patches to skip SanitizerCoverage and AddressSanitizer instrumentation

6.4.2 SymState

4 Submodules:

1. Symbolic Trace Parser
2. Solver
3. Solution Applier - Input Gen
4. Solution DB - Shared across all cores, duplicated in each core ^

- Switches between two input generation “regimes”
    - Depends on whether the seed has been observed by any other core

For unobserved seeds the SYMSTATE follows a three-stage pipeline

1. SYMBOLIC TRACE PARSER on language specific traces (C/C++: Z3 AST; Java: SMT-LIB2 strings)
2. SOLVER generates satisfying assignments
3. SOLUTION APPLIER transforms these into concrete input bytes
4. Seed previously processed by other cores bypass the pipeline entirely, using their “novel” fusing mutator for cross-core solution reuse

Fusing Mutator

(UNIQUE)

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

- If precise symbolic traces existed they would be ideal, however we cannot gain these traces due to concretization issues
- Fusing Mutator utilizing solutions from other seeds

What they do

1. First selects random solutions from the solution DB
2. Sequentially applies cached solution to input, generating a new set of inputs

What we benefit

1. Fuzzer can create new inputs to satisfy constraints of multiple seeds
    a. Allowing exploration of novel paths
2. Performance improvement by reusing solves and eliminating redundancy with solving overhead

Auxiliary Symbols

Solution Applier expresses path constraints in higher-level symbols, these auxiliary symbols significantly improve SYMSTATE's performance by eliminating any byte level representation in all path constraints.

Basically a symbol like ScanExtract is able to be the variadic argument where the format string is, and allow the solver to reason sscanf arguments directly over individual bytes. After SolutionApplier uses the inverse operation to produce the byte-level replacements.

6.4.3 Discussion

LLM-Based Function Modeling

The model ran, modeling pre-competition modeling 13 functions. Yet not ran during competition for strategic reasons

1. LLM generated models produced frequent incorrect results
    a. Prompt engineer better skill issue
    b. TODO implement 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. Differential analysis was resource intensive
    a. `concretenes checks` were disabled and increased massive path constainrs
    b. Future work to make concolic executor more efficient with sysman and leansym papers

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"

"In practice, dictionary-based fuzzing achieves better coverage when combined with other techniques"

Supposed to be used complementary with TESTLANG-BASED fuzzing in 6.6, because of the focus on structural aspects of input

- Function level dictionary approach and only applying them to the associated functions

This one is kinda just straight forward, the prompt an AI to analyze a function and build the dictionary on the fly.

- Takes in source code and corpus input with coverage information
- Selects functions to target for generations
- Guided by suspicious functions identified by MLLA
    - Reduces search space

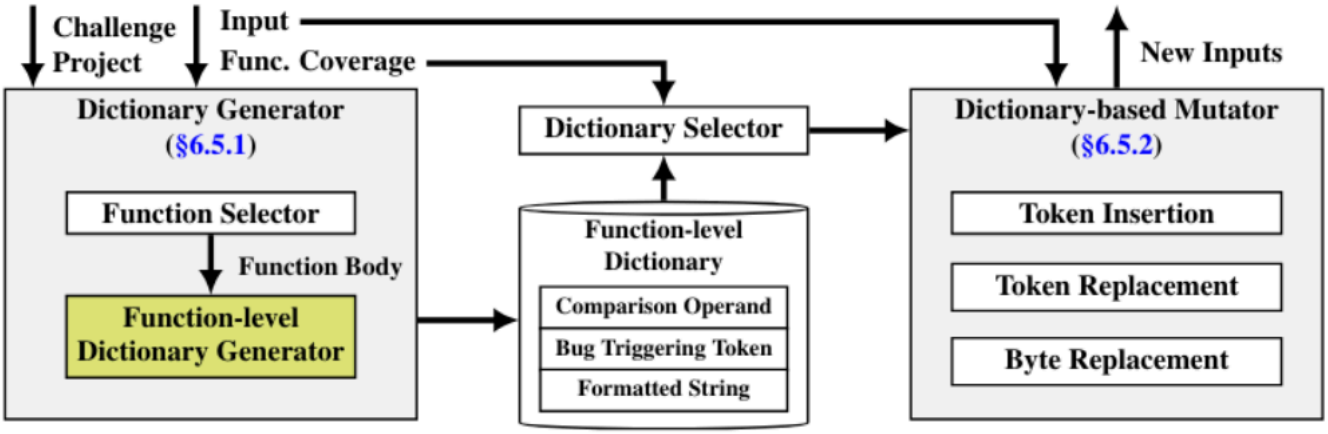


Figure 27: An overview of the function-level dictionary-based input generation.

6.5.1 Function level dictionary generator

Function Selector selects based off of input coverage and bug candidate information.

Prioritizing bug triggering tokens (i.e. long strings)

Used GPT-40 with comprehensive prompt engineering

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

Comparison Operand, minimizing tokens

“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.”

“For example, if the target function handles SQL queries, LLMs will produce SQL injection payloads (e.g., ‘’ OR ’1’=’1’‘) as tokens.”

“(e.g., for a function that parses User-Agent strings, it might generate "Mozilla/5.0 (X11; Linux x86_64)").”

6.5.2 Function level dictionary based mutator

Combines dictionary based strategies with traditional random mutations

1. Token insertion
    a. Randomly in input
2. Token Replacement
    a. Random chunk replaced with random token
        i. Structurally valide, semantically different
3. Byte replacement
    a. Small byte changes replaced with tokens
        i. For edge case behaviours

6.6 Testland-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.”

- Leveraging well known format specifications and LLM-generated python scripts
    - Partial specifications to focus the LLM on interesting parts of input format
- LLMs to reverse engineer, reduce computation cost for static analysis

- Guided fuzzing with LLM
    - Evaluating fuzzer feedback
    - Prioritize fuzzing directions with analysis strategy of the Harness Reverser
        - LLM-opinionated

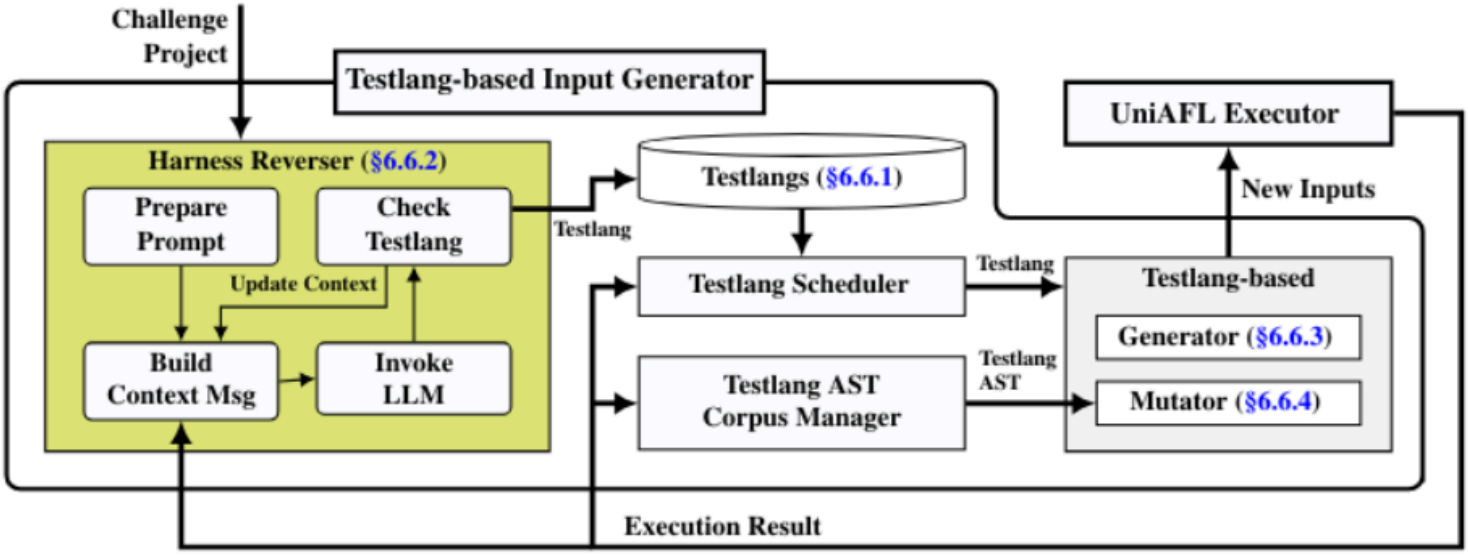


Figure 29: Overview of the Testlang-based input generator.

1. Harness Reverser
    a. Extracts input formats
2. Testlang and python files
    a. Outputs from Harness Reverser, and gen and mutate engines

When CP gives the harness etc., the Harness Reverser analyzes the target harness code and broader coder to understand input data.

Harness Reverser may also write python scripts

Data used in Testland scheduler.

Generated inputs executed by UniAFL executor and result in code coverage

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.

1. LLM friendly design for on the fly generation
2. Semantics of data fields
3. Generation extensions using external tools
4. Oracle provision for output quality evaluation
5. Partial update supports

    (these have a whole page of detail, probably irrelevant)

```
{
  // ... 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

“LLM-powered system that automatically analyzes target programs to extract input format specifications and generate corresponding Testlang descriptions”

“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”

Auxiliary Python generates are proposed to help handle complex type structures

Atlantis-Patching

Custom LLMs in Atlantis-Patching

Atlantis-SARIF

Benchmark

0-day Bugs