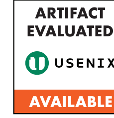

OSS-CRS: Liberating AIxCC Cyber Reasoning Systems for Real-World Open-Source Security

Andrew Chin[†] Dongkwan Kim[‡*] Yu-Fu Fu[†] Fabian Fleischer[†] Youngjoon Kim[†]
HyungSeok Han[‡] Cen Zhang[‡*] Brian Junekyu Lee[†] Hanqing Zhao[†] Taesoo Kim[†‡]
[†] Georgia Institute of Technology, [‡] Microsoft

Abstract

DARPA's AI Cyber Challenge (AIxCC) showed that cyber reasoning systems (CRSs) can go beyond vulnerability discovery to autonomously confirm and patch bugs: seven teams built such systems and open-sourced them after the competition. Yet the majority of open-sourced CRSs remain largely unusable outside their original teams, each bound to the competition cloud infrastructure that no longer exists. We present OSS-CRS, an open, locally deployable framework for running and combining CRS techniques against real-world open-source projects, with budget-aware resource management. We ported components from every AIxCC finalist CRS. Using the first-place system (ATLANTIS), we discovered 7 previously unknown bugs (one of high severity) across 6 OSS-Fuzz projects. OSS-CRS is publicly available as a sandbox project in the OpenSSF.

1 Introduction

AI-generated vulnerability reports [3, 19, 39, 50] are increasing the burden on open-source maintainers. The curl project shut down its bug bounty program after AI-written submissions overwhelmed reviewers with unconfirmed findings [48, 49]; FFmpeg maintainers criticized Google for reporting valid bugs without providing patches, calling the reports "CVE slop" [44]. In both cases, AI automation stops at discovery, leaving maintainers to validate findings and write fixes.

DARPA's AI Cyber Challenge (AIxCC, 2023–2025) [14] demonstrated that the gap between vulnerability discovery and validated remediation can be closed. Seven finalist teams built autonomous cyber reasoning systems (CRSs) that go beyond discovery: each CRS dynamically confirms a vulnerability with a proof of vulnerability (PoV) and synthesizes a patch validated against that PoV [13]. After the competition, all seven teams open-sourced their systems, in principle making it possible to deploy this capability against any open-source project.

Yet over half a year later, these CRSs remain largely unusable outside their original teams: each system is bound to team-specific infrastructure and interfaces. This adoption gap blocks the communities that should benefit from these systems: researchers cannot run comparable cross-CRS experiments under consistent settings; CRS developers lack a stable integration contract; and security practitioners cannot adopt deployable, budget-aware workflows that produce actionable findings.

To understand what prevents adoption, we analyzed the open-sourced codebases of all seven AIxCC finalists and identified three deployment barriers: ❶ *Infrastructure duplication*, each team independently rebuilt the same platform services; ❷ *Cloud lock-in*, every system targets the competition's Azure and Kubernetes environment, which was decommissioned after the finals; and ❸ *Monolithic design*, analysis techniques are embedded in monolithic systems, preventing researchers from comparing or combining them across teams. Even ATLANTIS, the first-place system [25], requires over 20 Azure virtual machines and cannot target new projects without its original cloud environment.

To address these barriers, we present OSS-CRS, an open framework for developing, running, and composing CRSs on real-world open-source projects. OSS-CRS provides a shared infrastructure layer including LLM budget management and cross-CRS artifact exchange, so that CRS developers can focus on analysis logic rather than platform engineering. It adopts the OSS-Fuzz [2] project format as its target interface, enabling any integrated CRS to target over 1,000 OSS-Fuzz projects without per-project customization. To validate the framework, we ported all seven CRSs archived from the AIxCC finals competition. We ran ATLANTIS against 8 OSS-Fuzz projects, discovering 7 previously unknown bugs, including one of high severity. OSS-CRS and all integrated CRSs are publicly available at `https://github.com/ossf/oss-crs`.

This paper makes the following contributions:

- An *empirical analysis* of all seven AIxCC finalist CRS codebases, identifying three deployment barriers that

[*] Work was done at Georgia Institute of Technology.

prevent practical reuse: infrastructure duplication, cloud lock-in, and monolithic design (§2).
- OSS-CRS, an open-source framework that addresses these barriers through a unified execution model and standard interface, budget-aware resource management across CPU, memory, and LLM usage, and support for combining CRS techniques across systems (§3).
- *Real-world validation*: porting ATLANTIS to OSS-CRS and discovering 7 previously unknown bugs (one of high severity) across 6 OSS-Fuzz projects, showing that competition-grade CRS techniques can be deployed without cloud infrastructure (§4, §5).

2 Background and Motivation

2.1 Cyber Reasoning Systems

A *cyber reasoning system* (CRS) is an autonomous agent that discovers and repairs software vulnerabilities without human intervention. The concept originated in DARPA's Cyber Grand Challenge (CGC, 2014–2016) [12, 52], in which teams built systems to attack and defend custom binaries. AIxCC [14] extended the concept to real-world open-source software in C and Java, using challenge targets drawn from OSS-Fuzz projects [2].

CRS capabilities. Traditional security tools typically focus on a single stage of vulnerability management: fuzzers discover crashes, static analyzers flag potential flaws, and program repair systems synthesize patches, but integrated pipelines from discovery through a validated fix remain uncommon.

A CRS unifies these stages. *Bug finding* generates a proof of vulnerability (PoV), an input that triggers abnormal execution such as a crash or sanitizer violation. *Bug fixing* synthesizes a patch and validates it by rebuilding the target and rerunning tests, confirming that the PoV no longer triggers while preserving the program's original functionality. A CRS can accept a variety of inputs: an entire source tree, a commit-level code diff, a SARIF report [38] from a static analyzer, or a fuzzing seed corpus.

Practical scenarios. This end-to-end capability supports several deployment settings. In CI/CD, a pipeline submits a pull-request diff and receives a PoV or confirmation of no bugs through a stable, machine-readable interface. For security practitioners, the system turns static-analysis findings into validated patches while enforcing spending limits. For research, multiple CRSs run on the same source tree with matched inputs, so results are reproducible and directly comparable.

Infrastructure requirements. These scenarios create concrete system requirements. A CRS must coordinate multiple tools across stages, pass artifacts such as crash inputs, PoVs, candidate patches, and build outputs between them, and recover when a stage fails. LLM calls span bug finding, triage, and patch generation, thus token use must be budgeted and capped like CPU time and memory. AIxCC allocated $50,000 in LLM credits per team; without budget controls, a single CRS run can exceed $1,000 per hour [4]. Orchestration, artifact exchange, and budget-aware execution are therefore core design problems, not implementation details.

OSS-Fuzz. The AIxCC competition drew its targets from Google's OSS-Fuzz [2], which provides reproducible, containerized builds for over 1,000 open-source projects. Each project defines a Dockerfile and build script that compile the target project with sanitizer instrumentation, and supplies fuzz targets that consume fuzzer-generated inputs. For CRS deployment, this offers a standardized way to build and test many projects without per-project setup. However, OSS-Fuzz runs one fuzzer per container and does not provide multi-component orchestration, cross-stage artifact exchange, or budget management.

Table 1: Deployment characteristics of the AIxCC finalist CRSs. "Local": can run end-to-end on a single machine against a new target without cloud provisioning. "Composable": analysis components can be extracted and recombined across systems.

CRS	Comp.	Infra	Middleware	Local	Composable
ATLANTIS [25]	9+	TF+K8s	Ka/PG+R*	×	×
BUTTERCUP [8]	14+	TF+Helm	R/M	○	×
ROBODUCK [4]	1	TF+VMs	–/S*	○	×
FUZZINGBRAIN [46]	4	TF+K8s	–/–*	○	×
ARTIPHISHELL [6]	53	TF+Helm	RMQ/PG+N*	△	×
BUGBUSTER [7]	16	TF+Helm	RMQ/PG+R*	×	×
LACROSSE [30]	3+	TF+VMs	RMQ/–	×	×

TF = Terraform, K8s = Kubernetes, Ka = Kafka, RMQ = RabbitMQ. R = Redis, PG = PostgreSQL, M = MongoDB, S = SQLite, N = Neo4j. * Uses LiteLLM [5] for LLM proxy and routing. ○ Post-competition standalone version available. △ Post-competition local deployment guide available.

2.2 From Competition to Deployment

To assess whether AIxCC CRSs can be deployed outside the competition, we analyzed the open-sourced repositories and deployment artifacts of all seven finalists [4, 6–8, 25, 30, 46]. We find three recurring barriers: *infrastructure duplication*, *cloud lock-in*, and *monolithic design*. These findings align with Zhang *et al.* [58], who systematize the techniques of all seven AIxCC finalists and conclude that "the real bottleneck is not technique capability but robust integration into autonomous systems." The next sections show where deployment breaks down in practice.

2.3 Barrier 1: Infrastructure Duplication

Table 1 summarizes the deployment characteristics of each team, where *Comp.* denotes the number of independently built container images. The *Infra* and *Middleware* columns show that teams selected different tools yet converged on

similar platform roles: container orchestration via Terraform with Kubernetes[28] or Helm[23], and coordination backends such as Kafka[1], RabbitMQ[42], Redis[43], and PostgreSQL[40]. Five of seven teams deployed LiteLLM [5] as their LLM gateway.

The repositories reveal further overlap not visible in the table: each team independently built LLM budget tracking, cost enforcement, and model routing logic on top of its proxy, as well as test environments for applying patches, rebuilding targets, and validating PoVs. As noted in §2.1, LLM budget control is a shared requirement, yet each team built this logic independently. The barrier is not tool diversity itself, but the repeated integration effort: overlapping infrastructure roles reimplemented across all seven teams.

2.4 Barrier 2: Cloud Lock-in

The AIxCC competition provisioned each team with dedicated Azure virtual machines orchestrated by Kubernetes. That environment has since been shut down, and the released artifacts still target infrastructure that no longer exists. For example, ATLANTIS, the first-place system [25], requires 20+ Azure VMs, 42 TiB of cloud storage, and runtime Azure SDK calls to scale Kubernetes node pools, despite open-sourcing its full codebase.

As the *Local* column of Table 1 shows, only half the teams added local execution support through standalone releases or local deployment guides [4, 6, 8, 46]. These efforts show that cloud decoupling is possible. However, a local environment must still provide the resource controls and isolation that cloud platforms supply: CPU and memory quotas, network separation between CRSs, and LLM budget limits. Without these, multiple CRSs cannot run on a single machine without contention or cost overruns.

2.5 Barrier 3: Monolithic Design

Beyond infrastructure and deployment, a structural obstacle remains: every CRS is a monolith with no modular internal interfaces. The seven finalist teams developed distinctive, often complementary techniques: LLM-based input mutation and hybrid fuzzing [25], LLM-first PoV generation [4], grammar-based fuzzing with LLM-generated grammars [6], expertise-driven multi-agent patching [8], diverse LLM strategy ensembles [46], traditional analysis with LLM-augmented patching [7], and multi-LLM workflow coordination [30]. Comparing and combining these techniques would reveal which approaches outperform others and whether ensembling improves overall results.

Yet as the *Composable* column of Table 1 confirms, no CRS exposes interfaces for component-level extraction. If ATLANTIS has a stronger fuzzer and BUTTERCUP has a stronger patcher, a researcher cannot combine them without reimplementing one inside the other. The AIxCC competition evaluated end-to-end system outputs but provided no way to attribute results to individual techniques; even Zhang *et al.*'s survey [58] could describe each team's methods but not compare them experimentally. As long as CRSs remain monolithic, techniques cannot be isolated, evaluated, or transferred beyond their original systems.

3 System Design

To address the barriers identified in §2, we propose OSS-CRS, a locally deployable infrastructure for running and combining existing CRS techniques. OSS-CRS is not itself a CRS; it is the platform on which CRSs are developed, deployed, and composed. It provides three capabilities: 1) a unified three-phase execution model and standard interface (`libCRS`) that remove per-team infrastructure duplication; 2) local execution with resource controls and isolation across CPU, memory, network, and LLM budgets, removing cloud dependencies; and 3) cross-CRS artifact exchange that enables combining techniques from independently developed CRSs without requiring a single monolithic pipeline. The following subsections detail how OSS-CRS realizes these capabilities.

3.1 Architecture Overview

Figure 1 illustrates the architecture of OSS-CRS. The OSS-CRS user interface defines a three-phase lifecycle (*prepare*, *build-target*, *run*) that is driven by a single configuration file that deploys CRSs against target projects. CRSs run in isolated Docker containers with dedicated CPU, memory, and LLM budget allocations; separate networks prevent direct inter-CRS communication. Shared infrastructure provides common services: the *LiteLLM proxy* handles model routing and budget enforcement, *file-based storage* persists artifacts (seeds, PoVs, patches, bug-candidates), and the *exchange sidecar* manages artifact flow between CRSs. CRSs interact with the platform through `libCRS`, which provides APIs for downloading targets, submitting findings, and validating patches.

3.2 OSS-CRS Interface

From the user's perspective, OSS-CRS only requires two inputs: a target project with source code and a single configuration file that specifies which CRSs to deploy along with their resource allocations. Optionally, users can also provide code diffs or bug candidates for targeted bug-finding in environments such as CI/CD pipelines. The user then sequentially runs three commands that set up CRSs, compile the target, and launch the analysis campaign. The main output artifacts are discovered bugs as PoVs and patches that fix them.

Three-phase lifecycle. OSS-Fuzz combines building and running into a single workflow: build the fuzz target, then run it. OSS-CRS introduces a three-phase model (*prepare*,

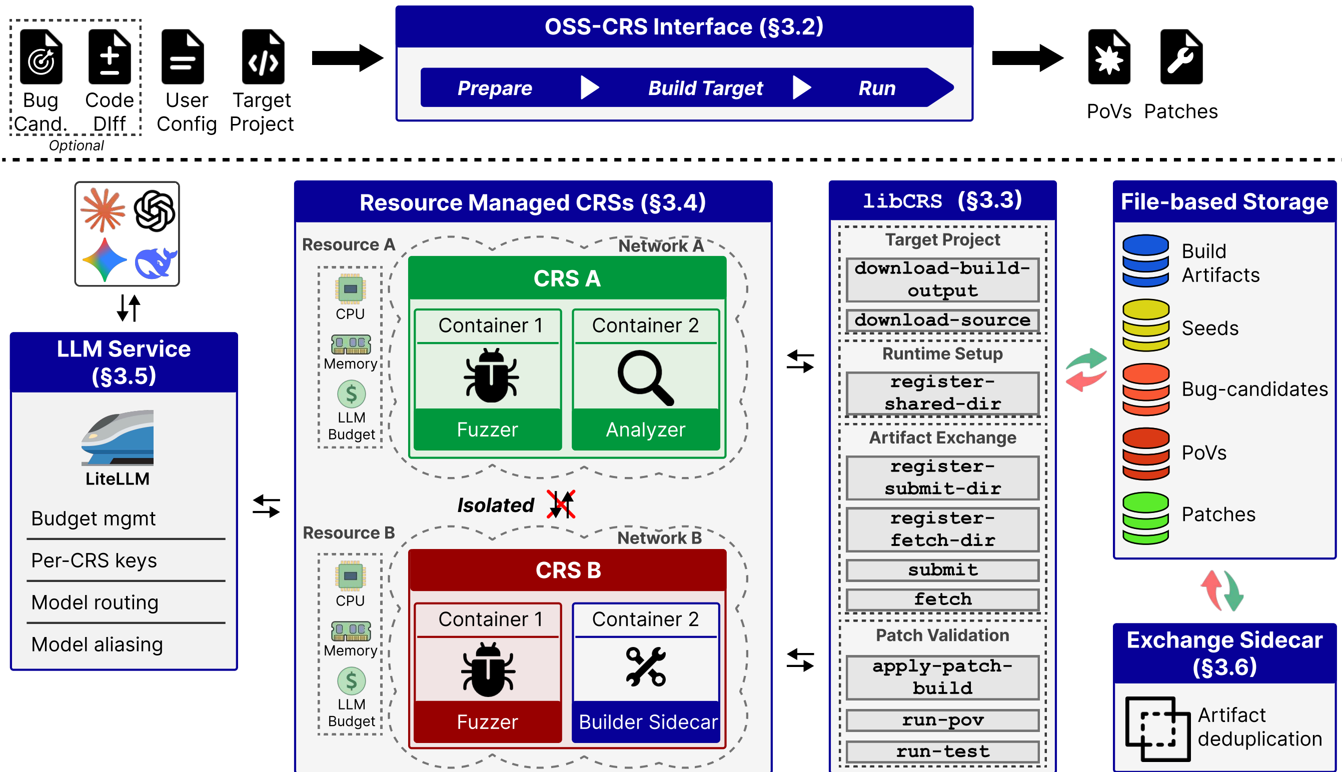

Figure 1: Architecture overview of OSS-CRS. Users provide a target project and configuration (optionally with code diffs or bug candidates) and receive PoVs and patches as outputs through the three-phase interface (*prepare*, *build-target*, *run*). CRSs run in resource-managed containers with isolated networks and interact with the platform through `libCRS`. All LLM calls are routed through the LiteLLM proxy, which handles model routing and per-CRS budget enforcement. The exchange sidecar deduplicates artifacts in file-based storage and synchronizes them across all CRSs.

build-target, *run*) that separates CRS setup from target compilation and running, enabling caching and modularity across diverse CRS architectures. The *prepare* phase builds CRS container images and their dependencies. These images contain the CRS's analysis tools but have no knowledge of the target project. A prepared CRS can analyze any compatible target without rebuilding, amortizing setup cost across multiple analysis campaigns. The *build-target* phase constructs images for target compilation and compiles the target project. Separating this phase from *prepare* allows CRSs to construct target-dependent containers while keeping target-independent components cached. The *run* phase launches all CRS containers and executes the analysis campaign. CRSs operate concurrently, exchanging artifacts through a shared directory and synchronizing findings via the `libCRS` interface. Users can specify a timeout or manually terminate the campaign.

Figure 2 summarizes the Docker workflow across three layers. The top layer shows Docker images: *prepare* builds CRS dependency images, *build-target* adds the target project and CRS builder images, and *run* produces the final CRS runner images. The middle layer shows containers: *build-target* runs instrumentation containers that compile the target, and *run* launches CRS runtime containers. The bottom layer tracks file artifacts: target source feeds into build artifacts and seed corpora during *build-target*, and the *run* phase produces PoVs, patches, and new seeds.

Single configuration. Users only need to define a single configuration file (`crs-compose.yaml`) that specifies CRSs, resource allocations, runtime environment, and LLM settings for all three phases. Adding or removing CRSs requires editing this one file, not coordinating multiple configurations. Users can also specify resource constraints without understanding CRS internals. The same file reproduces orchestration and resource settings across different machines.

Targeted analysis. By default, CRSs analyze the entire target codebase. For focused analysis (checking whether a recent commit introduces vulnerabilities or fixing a specific reported bug), users can provide targeting metadata that constrains the analysis scope. OSS-CRS accepts targeted inputs through standard channels. A code diff specifies changed code regions, enabling delta analysis between versions; directed fuzzers can focus on changed functions, and patch generators can scope fixes to the modified code. Bug-candidate reports identify specific issues for CRSs to address; OSS-CRS accepts SARIF reports [38] to align with the standard format

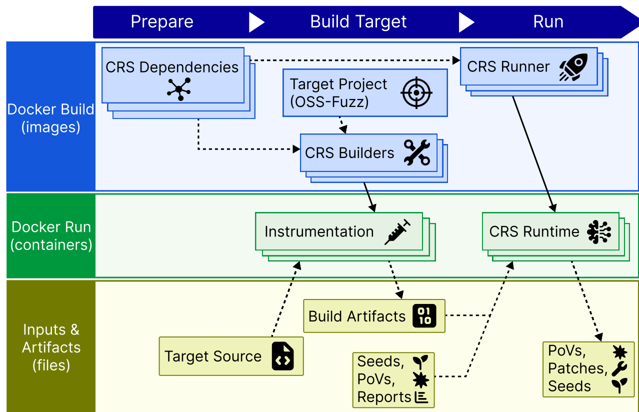

Figure 2: Docker workflow across the three operational phases, showing image construction (top), container execution (middle), and file artifact flow (bottom).

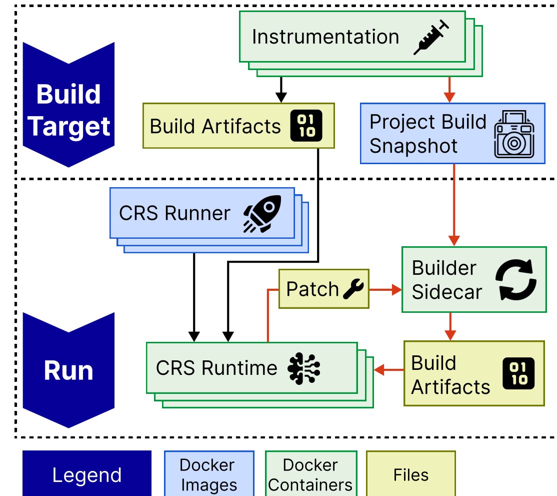

Figure 3: Builder sidecar workflow. The sidecar restores a snapshot of the compiled target, applies the patch diff, and performs an incremental rebuild.

Table 2: Representative `libCRS` commands used by CRS developers.

Category	Command	Purpose
Build outputs	`submit-build-output`	Publish build artifacts
Runtime setup	`register-shared-dir`	Share files within CRS
Artifact exchange	`register-submit-dir`	Register for background submission
Artifact exchange	`register-fetch-dir`	Register for background fetching
Artifact exchange	`submit`	Submit artifact (PoV, seed, patch)
Artifact exchange	`fetch`	One-shot fetch of artifacts
Patch validation	`apply-patch-build`	Apply patch and rebuild
Patch validation	`run-pov`	Run PoV against patched build
Patch validation	`run-test`	Run regression tests

of existing static analysis tools. Targeted analysis supports CI/CD integration, where CRSs check pull requests rather than the entire codebase.

3.3 `libCRS`

`libCRS` is a Python library automatically injected into every CRS container, providing the interface between a CRS's analysis logic and the OSS-CRS infrastructure features. A CRS uses the same `libCRS` interface regardless of deployment environment. Table 2 lists the core commands across four categories: *build outputs*; *runtime setup*; *artifact exchange*; and *patch validation*.

Build outputs. The *build-target* and *run* phases execute in isolated containers, so build artifacts must be handed off across the phase boundary. Build outputs such as instrumented binaries, source snapshots, and coverage metadata, are published via `submit-build-output` during compilation and retrieved at run time.

Runtime setup. A CRS may comprise multiple containers: a fuzzer generating inputs, an analyzer triaging crashes, a patcher synthesizing fixes. The `register-shared-dir` command provides intra-CRS file sharing for corpora, coverage data, and intermediate results, without passing through the inter-CRS exchange.

Artifact exchange. CRSs receive external data via `register-fetch-dir` and `fetch`: initial inputs supplied by the operator (seed corpora, reference diffs, bug-candidate reports) and artifacts submitted by other CRSs during ensemble execution. Conversely, CRSs upload their findings (PoVs, seeds, patches) via `register-submit-dir` and `submit`, which flow to other CRSs' fetch directories via the exchange sidecar.

Patch validation. Bug-fixing CRSs require a tight edit-compile-test loop to iterate on patch candidates. `libCRS` provides `apply-patch-build`, `run-pov`, and `run-test` for incremental rebuilding and testing through the builder sidecar.

Figure 3 shows the builder sidecar workflow. During the *build-target* phase, OSS-CRS captures a Docker image snapshot of the fully compiled target along with its build artifacts. In the *run* phase, CRS runtime containers receive the build artifacts and begin analysis. When a CRS generates a candidate patch, it invokes `apply-patch-build`, which sends the diff to the builder sidecar; the sidecar restores the snapshot, applies the patch, and performs an incremental recompilation. The CRS then calls `run-pov` to re-execute the crash-triggering input against the patched binary, and `run-test` to run the project's regression tests. Unlike shared infrastructure services, the builder sidecar runs within the CRS's resource allocation (`cpuset` and `memory_limit`), ensuring that rebuild costs are accounted to the CRS, not the platform.

3.4 Resource Management

Running multiple CRSs requires allocating heterogeneous resources: CPU cores, memory, and LLM API budgets for AI-powered reasoning.

Compute and memory isolation. Each CRS declares a `cpuset` string specifying its assigned CPU cores and a `memory` limit enforced via Docker cgroups (`cpuset`, `mem_limit`). Pinning CPUs prevents CRSs from contending for the same cores, while hard memory caps prevent a single CRS from destabilizing others.

LLM budget as a first-class resource. Beyond compute resources, OSS-CRS manages LLM API costs as a first-class resource. Each CRS may declare an `llm_budget` in US dollars; the LLM proxy tracks usage against this limit and rejects requests that would exceed it. This prevents cost overruns during long-running campaigns and enables fair comparison between CRSs with different cost profiles: a CRS that achieves the same results within a $50 budget is more efficient than one requiring $500.

Network isolation. A shared network introduces potential resource-management issues such as port conflicts between CRS containers and uncontrolled bandwidth consumption that could interfere with per-CRS resource guarantees. OSS-CRS mitigates this by creating separate Docker networks per CRS and restricting all inter-CRS data exchange to the filesystem-based artifact mechanism. This also simplifies access control for shared services like the LLM proxy and ensures fair comparison when benchmarking multiple CRSs side by side.

Flat Docker architecture. One approach to porting existing CRSs is to wrap them in Docker-in-Docker (DinD), running the entire system inside a single outer container. However, DinD complicates build performance, resource control, and debugging: Docker layer caches are harder to preserve across builds, host-level cgroup enforcement for individual inner containers is less direct, and debugging nested containers is substantially harder. OSS-CRS instead provides interfaces that let CRS developers build for a flat Docker architecture, where all containers are managed by the host Docker daemon. This enables straightforward per-container resource enforcement, preserves Docker layer caches, and simplifies debugging.

3.5 LLM Services

CRSs rely on LLMs for code understanding, patch generation, and vulnerability reasoning. However, different CRSs may prefer different providers and models. OSS-CRS handles these differences with an LLM proxy layer based on LiteLLM [5], an open-source proxy that natively supports multiple provider APIs and model aliasing.

Unified endpoint. All CRSs make standard OpenAI-compatible API calls to a single proxy that routes requests to the appropriate backend such as OpenAI, Anthropic, Google, or self-hosted inference servers. A CRS only needs to implement one API client, regardless of which provider the user configures. The CRS is not limited to only OpenAI API calls, as the endpoint accepts a variety of model provider APIs.

Model aliasing. CRSs reference models by logical names (*e.g.*, `claude-sonnet` or `gpt-4o`) rather than provider-specific identifiers. The proxy maps these aliases to concrete provider endpoints, so swapping providers requires only a configuration change, not CRS code modifications. Operators define model mappings in a YAML configuration file. Each entry specifies a logical `model_name` (what CRSs request) and provider-specific parameters including the actual model identifier, API credentials, and optional custom endpoints for self-hosted or Azure deployments. At validation time, OSS-CRS checks that each CRS's `required_llms` are available in the operator's configured model list.

Per-CRS budget enforcement. OSS-CRS generates a unique API key per CRS at campaign startup, each associated with its budget. The proxy tracks cumulative costs and rejects requests when the budget is exhausted. This per-CRS keying also enables fine-grained usage tracking such as which CRS made each request, what model was used, and how many tokens were consumed. When a CRS exhausts its budget, subsequent requests are rejected. The LiteLLM proxy is deployed as a service in the OSS-CRS infrastructure, and its lifetime is coupled with the run phase.

Deployment modes. OSS-CRS supports three deployment modes. In *internal mode*, OSS-CRS manages LiteLLM, and a key-generation sidecar; this is the default for standalone deployments. In *external mode*, operators provide an existing LLM proxy endpoint and key, useful when an organization already runs centralized LLM infrastructure; per-CRS budget enforcement depends on the external proxy's capabilities. In *disabled mode*, OSS-CRS performs no LLM setup, for CRSs that do not use LLMs.

3.6 Artifact Exchange

Running multiple CRSs simultaneously enables complementary workflows; a fuzzer can discover crashes while a separate patcher generates fixes. Our evaluation demonstrates that cross-CRS artifact flow works in practice (§4); quantifying performance gains over single-CRS runs remains future work. OSS-CRS enables this *ensemble execution* through a filesystem-based exchange mechanism that requires no direct communication between CRSs.

Exchange model. All inter-CRS coordination flows through a shared *exchange directory*. Each CRS writes artifacts to its private submit directory, and reads from a shared fetch directory that mirrors the exchange. The exchange sidecar synchronizes per-CRS submit directories into a shared exchange exposed through each CRS's fetch directory. The exchange organizes artifacts by type: `seeds` for fuzzing inputs, `povs` for crash-triggering inputs, `patches` for proposed fixes, and `bug-candidates` for triage reports. CRS-internal data (*e.g.*, coverage maps, model weights, intermediate analysis state) remains private to each CRS and is not shared

Table 3: CRSs integrated into OSS-CRS. The Components column counts separately integrated subcomponents (Table 6).

CRS	Type	# Components
Vanilla fuzzer	Bug-finding	2
ATLANTIS	Bug-finding	5
BUTTERCUP	Bug-finding	1
ROBODUCK	Bug-finding	1
FUZZINGBRAIN	Bug-finding	1
ARTIPHISHELL	Bug-finding	7
BUGBUSTER	Bug-finding	2
LACROSSE	Bug-finding	1
AI coding agent	Bug-finding	5
ATLANTIS	Bug-fixing	3
AI coding agent	Bug-fixing	5

through the exchange. Artifacts are stored under content-hash filenames, ensuring that duplicate discoveries from multiple fuzzers appear exactly once. This hash-based deduplication is the baseline strategy. Because the exchange sidecar is a shared OSS-CRS service rather than library code launched in a CRS container, we can extend it with more advanced logic without modifying individual CRSs: coverage-based deduplication that keeps only seeds increasing overall coverage, or stack-trace-based deduplication that groups PoVs by crash signature to reduce redundant triage.

Coordination without direct communication. CRSs coordinate implicitly through the artifacts they exchange, without direct messaging or task assignment. Each CRS polls for new artifacts at its own pace and decides independently what work to perform. This design provides fault isolation. If one CRS crashes, others continue operating on previously shared artifacts. Resource exhaustion in one CRS does not cascade to others, since each runs in an isolated container with independent resource allocations.

4 CRS Integration

The goal of OSS-CRS is to provide a modular framework where researchers can study, compare, and compose the distinctive techniques each team developed. As a first phase of validation, we integrated CRSs from all seven AIxCC finalist teams, spanning the spectrum from a minimal libFuzzer baseline to LLM-powered multi-language fuzzers and patch-generation agents (Table 3). These comprise bug-finding CRSs from every finalist team, ATLANTIS's bug-fixing agents, and a family of general-purpose AI coding agents (*e.g.*, CLAUDECODE) wrapped as both bug-finding and bug-fixing CRSs. The framework is open to the community to port additional systems. Our primary validation focuses on ATLANTIS for two reasons: ❶ it is the best-performing AIxCC finalist, and ❷ it provides no support for local execution outside the competition's cloud infrastructure. The remaining CRSs validate interface usability and ensemble mechanics.

Table 4: Environment variables injected into CRS containers.

Variable	Purpose
`OSS_CRS_TARGET`	Target project name
`OSS_CRS_TARGET_HARNESS`	Harness binary name
`OSS_CRS_NAME`	CRS name (for service discovery)
`OSS_CRS_CPUSET`	Allocated CPU cores (*e.g.*, `4-7`)
`OSS_CRS_MEMORY_LIMIT`	Memory limit (*e.g.*, `16G`)
`OSS_CRS_LLM_API_URL`	LLM proxy endpoint
`OSS_CRS_LLM_API_KEY`	Per-CRS API key for budget enforcement

4.1 Integration Requirements

CRS developers integrate their system into OSS-CRS via a `crs.yaml` manifest consisting of three phase-aligned sections.

`prepare_phase` builds the CRS's own container images (*e.g.*, fuzzers, analyzers, LLM agents) that are target-independent and reusable across projects.

`target_build_phase` declares build steps that require knowledge of the target: custom instrumentation passes, sanitizer-specific compilations, or snapshot captures for incremental patch validation. In the AIxCC competition, all seven teams independently implemented their own build and patch-compilation pipelines: ATLANTIS built a `cp_manager` with Docker-in-Docker, BUTTERCUP used a dedicated `build-bot` service, and ARTIPHISHELL deployed separate `patchery` and `patch-validation-testing` components [6, 8, 25]. With OSS-CRS, CRS developers extract these custom build passes into `target_build_phase` declarations.

`crs_run_phase` defines the runtime modules as a set of named containers, each with its own Dockerfile. A multi-component CRS (*e.g.*, a fuzzer, an analyzer, and a builder sidecar) declares one entry per component, and OSS-CRS launches them together with shared networking and resource limits.

OSS-CRS also injects environment variables into every CRS container at runtime (Table 4). These expose the target name, resource allocation, and LLM proxy credentials, so CRSs can adapt without hard-coded assumptions. For instance, a fuzzer can spawn one worker per core.

4.2 Vanilla Fuzzer

The simplest integrated CRSs are thin wrappers around the stock fuzzing engines used by OSS-Fuzz: CRS-LIBFUZZER wraps libFuzzer [41] for C/C++ targets, and `crs-jazzer` wraps Jazzer [24] for Java. They serve as a baseline: they require no LLM, use a single container, and exercise the minimal OSS-CRS interface, *i.e.*, prepare, build-target, and run, with nothing beyond standard fuzzing. The CRS-LIBFUZZER build script (Figure 4) is two lines: `compile` the target, then

```bash
#!/bin/bash
set -e
compile
libCRS submit-build-output $OUT build
```

Figure 4: Build script for CRS-LIBFUZZER.

hand over artifacts to the run phase. This two-line pattern is the minimal build contract; CRSs can extend it freely by adding custom compiler passes, instrumenting with different sanitizers, or running static analyzers, while OSS-CRS handles container orchestration and artifact transfer.

4.3 Porting ATLANTIS

ATLANTIS, the first-place AIxCC system [25], is among the most complex CRSs: it comprises multiple sub-systems (three independent bug-finding systems), uses multiple instrumentation passes (coverage, AddressSanitizer [45], and uninstrumented builds), and combines diverse fuzzing engines with multiple LLM agents.

Porting process. Converting ATLANTIS to the OSS-CRS interface required four categories of changes. ❶ *Manifest creation.* We wrote `crs.yaml` manifests for each sub-system, declaring its supported languages, required LLM models, capabilities, and prepare/build/run phase definitions. ❷ *Artifact submission.* Competition-specific API usage (*e.g.*, sending PoVs) was replaced with `libCRS` commands (`register-submit-dir`, `submit`). ❸ *Project building.* Each ATLANTIS sub-system had its own way of adding CRS tooling to the target project image: ATLANTIS-MULTILANG rebuilt the project image with a custom base image and ATLANTIS-C mounted tooling into the project container and overwrote the compiler environment variable. Under OSS-CRS, all sub-systems follow the same standard: providing a `builder.Dockerfile` that receives the target project base image as a build argument, unifying the build workflow. ❹ *Image caching via prepare phase.* In the competition environment, CRS images are prebuilt into registries. However, for rapid local CRS development, OSS-CRS supports rebuilding CRS images through its infrastructure. As such, each ATLANTIS sub-system needed to optimize its build process by partitioning some images into the prepare phase.

Eight components ported. We ported ATLANTIS's bug-finding sub-systems as five separate components and ATLANTIS's bug-fixing sub-systems as three separate components, each integrated as an independent CRS (Table 3). For bug-finding, ATLANTIS-MULTILANG and ATLANTIS-C each bundle two fuzzer variants, while ATLANTIS-JAVA contributes one (Table 6). ATLANTIS-MULTILANG handles C/C++/Java targets using microservices-based fuzzing driven by directed fuzzing and LLM agents. ATLANTIS-JAVA targets Java projects with Jazzer [24] sinkpoint-based fuzzing, ATLANTIS-C targets C/C++ projects with custom libAFL [18] and AFL++ [17] fuzzers, and both ATLANTIS-JAVA and ATLANTIS-C use a high-throughput agentic seed generator. Each sub-system has its own `crs.yaml` manifest, its own target build process, and runs in its own containers. The ATLANTIS bug-fixing CRSs are also integrated. We ported the three highest-accuracy agents (ATLANTIS-PRISM, ATLANTIS-MULTI-RETRIEVAL, and ATLANTIS-VINCENT) each as an independent patching CRS (Table 3), selecting those whose per-vulnerability success rate exceeds 90% on PatchIsland's AIxCC patch experiment [26]. ATLANTIS-MULTI-RETRIEVAL gathers code context around the crash, then iteratively generates and refines patches, querying for additional context as needed; ATLANTIS-VINCENT runs a three-stage pipeline of root-cause analysis, project-specific property inference, and patch generation; and ATLANTIS-PRISM cycles between analysis, patching, and evaluation agents, iteratively refining its understanding of the bug. PatchIsland's agents already build, apply, and validate candidate patches through its own `Crete` framework, so integration required switching the validation to use `libCRS`'s patch-validation pipeline (§4.5).

What changed, what was preserved. The original ATLANTIS requires over 20 Azure VMs across multiple Kubernetes node pools, three separate LiteLLM proxy instances, and dynamic node scaling via Azure SDK calls (§2.4). Porting replaced all Kubernetes orchestration with a flat Docker architecture, collapsed the three LiteLLM proxies into the single OSS-CRS proxy, and substituted the competition scoring API with `libCRS`'s artifact submission. Each sub-system also shed auxiliary components: ATLANTIS-MULTILANG and ATLANTIS-JAVA dropped their `concolic` hybrid-fuzzing engines, and ATLANTIS-C dropped its backup fuzzing engines and harness scheduler. We excluded `concolic` because its instrumentation introduced compatibility issues across diverse target projects; in the original competition it contributed only 1.7% of ATLANTIS's results [25], so we expect minimal impact on bug-finding capability. On the patching side, we likewise excluded PatchIsland's CLAUDELIKE and MARTIAN agents, which scored below 90% on its patch benchmark [26]. All primary CRS logic, including fuzzers, LLM agents, and analysis pipelines, was preserved with minimal modification, confirming that the core analysis techniques are independent of the deployment infrastructure.

4.4 Porting Other AIxCC Finalist CRSs

We port the remaining six finalist teams following the bug-finding taxonomy from the AIxCC SoK [58], which identifies proof-of-vulnerability (PoV) generation and seed generation as the recurring LLM-driven components shared across teams. We therefore focus on extracting each team's agentic workflows. When such an agent interacts with a stock, unmodified fuzzer, we port only the agent and leave the fuzzer out: OSS-

CRS's ensemble mechanism lets the seed generator feed any other fuzzing CRS at runtime. When a team instead incorporates a customized fuzzer, we integrate it as a separate component whenever it can be decoupled from the rest of the CRS.

Under this strategy, BUTTERCUP [8], ROBODUCK [4], FUZZINGBRAIN [46], and LACROSSE [30] are ported as LLM agents only. ATLANTIS, BUGBUSTER [7], and ARTIPHISHELL [6] implement their own custom fuzzers, which are integrated into OSS-CRS separately.

Across all of these ports, we also strip out each CRS's harness-orchestration logic. OSS-CRS runs one campaign per fuzz harness, so the competition-level scheduling that divides effort across a target's many harnesses is not applicable under our model. We thus omit components such as ATLANTIS-C's harness-deprioritization scheduler [25] and BUGBUSTER's BandFuzz collaborative-fuzzing coordinator [47].

4.5 AI Coding Agents

A growing class of CRSs wraps a general-purpose AI coding agent around the OSS-CRS interface. We integrated five such agents (Claude Code, Codex, Copilot CLI, Gemini CLI, and OpenCode), each in both a bug-finding and a bug-fixing configuration. As a worked example we describe CLAUDECODE, a CRS that uses an LLM agent to analyze crash traces and source code for vulnerabilities discovered by other CRSs in order to generate patches.

Builder sidecar in action. CLAUDECODE validates each candidate through `libCRS`'s three-step patch validation pipeline: `apply-patch-build` sends the unified diff to a builder sidecar, which applies the patch and performs an incremental rebuild from a pre-captured build snapshot; `run-pov` re-executes the crash-triggering input against the patched binary to confirm the vulnerability is resolved; `run-test` runs the project's regression tests to check for functional regressions. Because the builder sidecar handles all build-system complexity (such as Makefiles), CLAUDECODE's implementation contains no build logic. The CRS focuses entirely on LLM-driven patch synthesis, delegating compilation and testing to the framework.

5 Zero-day Results

To evaluate whether OSS-CRS enables practical, large-scale vulnerability discovery and fixing on real-world software, we ran the ported ATLANTIS CRS against open-source projects.

Target selection. We selected 8 OSS-Fuzz projects spanning C/C++ and Java, chosen to cover a range of project sizes (3 kLoC to 1,960 kLoC), application domains (databases, parsers, network servers, cryptographic libraries), and existing fuzzing maturity (from newly onboarded to heavily fuzzed for years). We did not cherry-pick projects based on expected vulnerability yield.

Running setup. All experiments ran on a single machine with 32 CPU cores and 128 GB RAM, running Ubuntu 22.04 with Docker 27. Each campaign allocated 16 cores and 64 GB RAM to the bug-finding CRS, with a 24-hour timeout per target project. LLM budgets were set to $50 per campaign, using a mix of Claude and GPT-4o through the OSS-CRS LLM proxy.

5.1 Zero-day Bugs Found

Table 5 summarizes the 7 zero-day vulnerabilities discovered by running ATLANTIS on OSS-CRS across 6 OSS-Fuzz projects. We found sanitizer crashes in two selected projects, Mosquitto and WAMR, but could not verify them as security bugs. Using the NIST CVSS v3.1 calculator [37], we self-assign severity scores, classifying one as high severity (CVSS ≥ 7.0), two as medium, and four as low. At the time of writing, six have been fixed by upstream maintainers, and one remains unpatched. All vulnerabilities are memory-safety bugs in C, but the set also includes logic bugs and undefined-behavior flaws (CWE-476, CWE-674, CWE-681), demonstrating that the CRS finds null-pointer, schema-validation, and numeric-conversion issues, not only fuzzer-class crashes. We present three cases in detail from two projects, all fixed and merged by upstream maintainers, to illustrate the range of bugs OSS-CRS finds and the quality of evidence it produces.

Case study 1: jq (C, undefined behavior). We found two undefined-behavior vulnerabilities in jq, a widely used command-line JSON processor (34k GitHub stars). ❶ Null pointer arithmetic in `found_string()` (`jv_parse.c`:449, CWE-476, CVSS 2.5): when parsing an empty string `""`, the parser never calls `tokenadd()`, leaving `tokenbuf` as `NULL` and `tokenpos` as zero. The expression `tokenbuf + tokenpos` evaluates `NULL + 0`, which is undefined behavior per the C standard. ❷ Float-to-integer overflow in the `dtoi` clamping macro (`builtin.c`:374, CWE-681, CVSS 3.3): the modulo operator `%` triggers an out-of-range cast when either operand is $\geq 2^{63}$. The macro's upper-bound guard `-(n) < INTMAX_MIN` uses strict less-than, so the boundary value 2^{63} (whose negation equals `INTMAX_MIN` exactly) falls through to an unclamped `(intmax_t)(n)` cast. Both bugs were detected by UBSan instrumentation.

Case study 2: libyang (C, logic/recursion). We found an infinite-recursion vulnerability (CWE-674, CVSS 7.5) in libyang's union/leafref type resolution. A crafted YANG module with a circular `leafref` inside a `union` type causes unbounded recursion during `lys_parse_mem()`, leading to stack overflow and denial of service. The call chain cycles through `lyplg_type_store_union()` → `union_find_type()` → `lyplg_type_store_leafref()` → resolution back to the union type → re-entry into `lyplg_type_store_union()`.

Table 5: Zero-day vulnerabilities (7) discovered using OSS-CRS with ATLANTIS across 6 open-source projects. All findings were reported to upstream maintainers and DARPA. The Reference column cites the public issue, fix PR, or advisory.

#	Project	Lang	kLoC	★	Vulnerability	CWE	Location	Harness	CVSS	Status	Reference
1	libxml2	C	130	723	OOM Error Misreport	CWE-252	`xmlIO.c:1332`	Existing	2.5	Fixed	#1067*
2	libyang	C	115	410	Infinite Recursion	CWE-674	`union.c:490`	Existing	7.5	Fixed	GHSA-mq87-gwqp-w3v8
3	FRRouting	C	627	4.0k	NULL Pointer Deref	CWE-476	`nhrp_packet.c:158`	Custom	6.5	Fixed	PR #20932
4	memcached	C	32	14.1k	Heap Buffer Underflow	CWE-125	`mcmc.c:140`	Existing	3.9	Fixed	#1268
5	jq	C	30	33.8k	Float-to-int Overflow	CWE-681	`builtin.c:374`	Custom	3.3	Fixed	#3482
6					Null Pointer Arithmetic	CWE-476	`jv_parse.c:449`	Existing	2.5	Fixed	#3483
7	arp-scan	C	3	1.2k	*Redacted for responsible disclosure*				5.4	Pending	

*Fixed in the maintainer's `libxml2-ee` fork; the corresponding upstream GitLab issue was closed.

```
1  module leafref-unions {
2      namespace "http://example.com/leafref-unions";
3      prefix "lu";
4
5      container config {
6          leaf circular-ref-1 {
7              type union {
8                  type leafref {
9                      path "../circular-ref-1";
10                 }
11             }
12             default "circular-value";
13         }
14     }
15 }
```

Listing 1: YANG module triggering infinite recursion in libyang's union/leafref resolution: the `leafref` path points back to the enclosing leaf, causing unbounded re-entry during default-value compilation.

Listing 1 shows the 15-line YANG module that triggers the bug: a single leaf whose type is a union containing a `leafref` that points back to itself. When libyang compiles the module's default value, the circular reference causes unbounded recursion with no cycle detection.

It demonstrates that OSS-CRS finds logic bugs beyond classic memory corruption, such as schema-validation flaws that require reasoning about type-resolution semantics, not just memory access patterns.

> **Finding 1:** OSS-CRS discovers vulnerabilities beyond classic memory corruption, including undefined behavior (null-pointer arithmetic, numeric-conversion overflow) and logic bugs (schema-validation cycles).

5.2 Patches and Disclosure

Patch generation and validation. For each vulnerability, OSS-CRS feeds the crash trace, root-cause analysis, and surrounding source code to ATLANTIS, which uses an LLM to generate a minimal candidate patch as a unified diff. Each candidate is then validated automatically through the `libCRS` infrastructure (§3.3): the patch is applied and the project rebuilt, the original PoV is re-executed to confirm the crash no longer triggers, and the project's test suite is run to check

```
1    // jv_parse.c: found_string()
2    static pfunc found_string(struct jv_parser* p) {
3  +   if (p->tokenpos == 0) {
4  +     TRY(value(p, jv_string("")));
5  +     return 0;
6  +   }
7      char* in = p->tokenbuf;
8      char* out = p->tokenbuf;
9      char* end = p->tokenbuf + p->tokenpos;
```

Listing 2: Generated patch for jq's null-pointer arithmetic (CWE-476). An early return for empty strings avoids `NULL + 0` undefined behavior.

```
1   // builtin.c: dtoi macro
2  -#define dtoi(n) ((n) < INTMAX_MIN  \
3  -  ? INTMAX_MIN : -(n) < INTMAX_MIN \
4  -  ? INTMAX_MAX : (intmax_t)(n))
5  +#define dtoi(n) ((n) < INTMAX_MIN   \
6  +  ? INTMAX_MIN : -(n) <= INTMAX_MIN \
7  +  ? INTMAX_MAX : (intmax_t)(n))
```

Listing 3: Generated patch for jq's `dtoi` macro (CWE-681). Changing < to <= clamps the boundary value 2^{63} correctly.

for regressions. We then manually reviewed all patches before submitting them to upstream maintainers. The following three examples illustrate the kinds of patches OSS-CRS produces.

Patch: jq empty-string guard and dtoi fix. Listing 2 shows our generated patch for the null-pointer arithmetic bug. It short-circuits `found_string()` when `tokenpos` is zero, returning an empty `jv_string` before any pointer arithmetic. This avoids the `NULL + 0` expression entirely. Listing 3 shows the one-character fix for the `dtoi` overflow: changing < to <= in the upper-bound guard so that the boundary value 2^{63} is correctly clamped to `INTMAX_MAX` instead of falling through to the undefined cast. The jq maintainer merged the `dtoi` fix within one day of our report (PR #3486); the null-pointer fix (PR #3485) was also merged.

Patch: libyang cycle detection. No CRS-generated patch file is available for the libyang infinite-recursion bug; the upstream maintainer implemented the fix directly (commit `b5f56b3`), adding cycle detection during leafref resolution so that circular type references are rejected at schema-compile time rather than causing unbounded recursion. The bug was

published as advisory GHSA-mq87-gwqp-w3v8 and fixed in libyang v5.0.6.

Finding 2: Generated patches are targeted and minimal: the jq fixes are a four-line early return and a one-character operator change, each addressing the precise root cause.

Disclosure outcomes. Of the 7 findings, six have been fixed by upstream maintainers: jq ($\times 2$), libyang, FRRouting, memcached, and libxml2. The remaining one finding is still pending maintainer response. We reported all vulnerabilities following each project's preferred disclosure process, including GitHub Security Advisories and direct email.

6 Discussion

6.1 Challenges and Lessons

Docker-in-Docker vs. flat Docker. Our initial attempt at local portability wrapped each CRS in a Docker-in-Docker (DinD) container to support container operations in the CRS. As discussed in §3.4, DinD complicated resource control, build caching, and debugging, so OSS-CRS adopted a flat Docker architecture instead. The broader lesson is that the framework should absorb the container-level operations that CRSs commonly need (such as building the project and validating patches), and expose them as first-class infrastructure helpers. A hands-off framework that instead leaves each CRS to reinvent these operations pushes that complexity into every CRS (DinD being the prime example) and complicates both the CRS and the framework. We therefore designed OSS-CRS to proactively identify and provide such helpers, rather than leaving them to individual CRSs.

Build-system diversity. OSS-Fuzz projects use heterogeneous build systems (Make, CMake, Autoconf, Bazel, Meson, and more). During CRS integration, we found that build assumptions baked into the competition environment (pre-installed tool versions, fixed filesystem layouts) frequently broke on different target projects. OSS-CRS mitigates this by building targets through OSS-Fuzz's official build flows, inheriting the build environment that each project's maintainers already support.

Porting effort. Adapting ATLANTIS to the OSS-CRS interface required understanding the system's internal architecture and artifact flow without changing the core logic of the modules we ported. The changes fell into four categories: configuration (writing `crs.yaml` manifests), I/O adaptation (replacing competition-specific API calls with `libCRS` commands), build integration (replacing containerized patch compilation with `libCRS`'s builder sidecar, which handles the apply-patch, rebuild, and test cycle), and image optimizations using the prepare phase. For the ATLANTIS-MULTILANG port, this integration work required approximately 3 person-days. For ATLANTIS-C, the port took over 4 person-days, as its tighter coupling to competition-specific build pipelines and container orchestration demanded more extensive I/O and build-integration changes. For ATLANTIS-JAVA and CLAUDECODE, integration required approximately 1 person-day each. All three of ATLANTIS's patching CRSs (ATLANTIS-PRISM, ATLANTIS-MULTI-RETRIEVAL, and ATLANTIS-VINCENT) together required only 2 person-days: they share the `Crete` framework, which already builds and validates candidate patches, so the port only had to redirect patch validation to `libCRS`. LACROSSE and FUZZINGBRAIN, being LLM-only, took roughly 1 person-day each. ROBODUCK is also LLM-only but took 3 person-days, since separating its agents from the rest of the CRS was complicated by tight coupling to the architecture. BUTTERCUP took 4 person-days because its C and Java targets relied on different coverage mechanisms; BUGBUSTER took 7 person-days due to the instrumentation toolchain environment for the custom fuzzer; and ARTIPHISHELL took 8 person-days, due to the large number of components and the tight coupling to the `pydatatask` [16] plumbing. In general, porting effort depends on the gap between a CRS's original design and the four categories above: systems with hard-coded competition APIs, refactoring containerized build flows, non-standard configuration formats, or tightly coupled image assumptions require proportionally more adaptation work.

6.2 Limitations and Threats to Validity

Our results demonstrate feasibility: CRS logic can be decoupled from competition infrastructure and applied to real OSS targets with actionable outputs. We note several limitations of OSS-CRS in its current state.

Single AIxCC patching system ported. For bug fixing, OSS-CRS integrates only ATLANTIS's patch agents, alongside general-purpose AI coding agents (§4.5). While ATLANTIS is the best-performing finalist, purpose-built patchers from other teams may exercise interface assumptions or repair strategies that a single team's system does not.

Target format assumptions. OSS-CRS currently targets projects compatible with OSS-Fuzz, which require a standardized build script, a containerized environment, language restrictions, and at least one fuzz harness. Projects not yet onboarded to OSS-Fuzz require harness development, which is outside the scope of the framework.

Target selection. Our 8 target projects were selected from OSS-Fuzz's project corpus based on adoption and security relevance.

LLM non-determinism. LLM-based CRS components produce non-deterministic outputs, affecting both vulnerability discovery and patch generation. Our results reflect specific model versions and prompt configurations; different models or API versions may yield different findings.

6.3 Community and Future Work

OSS-CRS is developed as a sandbox project in the OpenSSF to support the open-source security community. Our roadmap focuses on three directions:

- *Cross-CRS technique analysis.* With ensemble execution and cross-CRS artifact exchange, we aim to answer two questions: which individual techniques are most effective, and which combinations yield the best results?
- *Additional CRS types.* Beyond bug-finding and bug-fixing, we are extending OSS-CRS to harness-generation and triage CRSs, broadening the range of techniques the framework can host and compose.
- *Broader target coverage.* We plan to apply integrated CRSs to a wider range of open-source projects to discover vulnerabilities at scale.

7 Related Work

Autonomous cyber reasoning. The Cyber Grand Challenge (CGC) [12, 52] introduced autonomous systems that find and patch vulnerabilities in custom binaries on the DECREE OS. AIxCC [14] extended this to real-world open-source software, producing seven finalist CRSs whose techniques are analyzed by Zhang *et al*. [58]. OSS-CRS differs from both competition platforms: CGC provided a fixed binary format and a scoring API; AIxCC provided Azure infrastructure with competition-specific endpoints. OSS-CRS provides a reusable framework that persists beyond any single competition, targeting OSS-Fuzz's corpus of open-source projects. On top of this framework, CRSBench [51] is a benchmark suite that lets security practitioners and researchers evaluate and compare CRSs under controlled conditions.

Fuzzing infrastructure. OSS-Fuzz [2] provides continuous fuzzing for open-source projects but supports only single-container, single-fuzzer execution without bug-fixing or LLM integration (§2.1). FuzzBench [36] evaluates fuzzer performance on standardized benchmarks but does not support multi-component CRS workloads. ARVO [34] reproduces historical OSS-Fuzz vulnerabilities for research but provides no execution framework. Magma [22] and UniFuzz [32] offer ground-truth benchmarks for evaluating fuzzers but focus on single-fuzzer comparison, not multi-technique composition. OSS-CRS complements these tools: it uses OSS-Fuzz project definitions as targets and could integrate with FuzzBench for CRS-level comparison.

Ensemble and collaborative fuzzing. Collaborative fuzzing [11, 20, 21, 60] demonstrated that given fixed resources, distributing effort across multiple fuzzers outperforms focusing on any single fuzzer, motivating ensemble approaches. OSS-CRS generalizes this insight from fuzzer ensembles to CRS ensembles that combine heterogeneous techniques, including fuzzing, static analysis, LLM reasoning, and autonomous patching. It coordinates them via unified resource and LLM budget allocation and a filesystem-based exchange mechanism that requires no modification to individual CRSs.

LLM-powered fuzzing. LLMs are increasingly used to augment fuzzing. TitanFuzz [15] and Fuzz4All [54] use LLMs to generate test inputs, ChatAFL [35] applies them to protocol fuzzing, and ELFuzz [10] synthesizes entire fuzzers via LLM-driven evolution. HLPFuzz [55] and G^{2}FUZZ [59] leverage LLMs for constraint solving and input generator synthesis, respectively. In a complementary direction, OSS-Fuzz-Gen [33] and SHERPA [29] use LLMs to synthesize fuzz harnesses, automating a key bottleneck in onboarding new targets. These works improve individual fuzzing components; OSS-CRS provides the orchestration layer to compose such LLM-augmented tools into complete CRS pipelines.

LLM-based vulnerability repair. Several benchmarks evaluate LLM agents on security tasks: SEC-bench [31], Cybench [57], AutoPatchBench [9], and CVE-Bench [53]. On the technique side, San2Patch [27] automates vulnerability repair from sanitizer logs via tree-of-thought LLM reasoning, and PatchAgent [56] mimics human debugging expertise for practical program repair. The AIxCC competition demonstrated that integrating LLMs into full CRS pipelines, combining code reasoning with fuzzing and program analysis, yields stronger results than applying them in isolation [58]. OSS-CRS provides the infrastructure to study this integration: its LLM proxy tracks per-CRS costs, enabling comparison of LLM-augmented versus traditional techniques under fixed budgets.

8 Conclusion

DARPA's AI Cyber Challenge produced seven autonomous CRSs for finding and fixing vulnerabilities, but left them entangled with competition-specific infrastructure. We analyzed all seven finalist CRSs and identified three categories of barriers (infrastructure duplication, cloud lock-in, and monolithic design) that prevent real-world deployment. OSS-CRS addresses these barriers with a standardized CRS interface, resource isolation with LLM budget management, and cross-CRS artifact exchange for composing complementary techniques. By porting the AIxCC champion system and discovering 7 previously unknown bugs across 6 open-source projects, we provide feasibility evidence that competition CRS logic can be decoupled from its original infrastructure and applied to real-world OSS projects. OSS-CRS is available as open source, a first step toward broader cross-CRS accessibility and evaluation. We invite the community, including researchers, CRS developers, and security practitioners, to actively develop, compose, and evaluate CRS techniques and help mature OSS-CRS into shared infrastructure for open-source software security.

Ethical Considerations

OSS-CRS lowers the barrier to automated vulnerability discovery and patching in open-source software. This automation carries inherent dual-use risks, as the same techniques that help defenders find and fix bugs could also help attackers identify exploitable weaknesses. However, we believe the benefits outweigh these risks: our system produces not only proofs of vulnerability but also validated patches, shifting the balance toward defense.

We ran OSS-CRS entirely in isolated Docker containers on local machines without transmitting any data to external services. Also, because OSS-CRS executes all analysis inside containerized environments and operates on fuzzing harnesses originally written for security testing, it poses no risk to running production systems.

All vulnerabilities discovered in this work were reported to upstream maintainers following each project's preferred disclosure process, including GitHub Security Advisories, SourceForge, and direct email. We withheld public disclosure of vulnerability details until maintainers acknowledged the reports. The proof-of-vulnerability inputs and patches are shared with maintainers to facilitate timely remediation. We are actively communicating with project maintainers and responsible parties to ensure all reported issues are addressed.

Open Science

OSS-CRS, including the framework, all integrated CRSs, and evaluation artifacts, is publicly available. We archive a snapshot at `https://zenodo.org/records/20072038`, and the actively maintained repository lives at `https://github.com/ossf/oss-crs`. The repository includes the CRS interface specification, `libCRS` library, configuration templates, and documentation for porting new CRSs.

Acknowledgments

We thank Younggi Park for the initial design and implementation of the bug-fixing infrastructure in OSS-CRS. We thank Sin Liang Lee, Isaac Hung, Joshua Wang, and Jiho Kim for their contributions to the bug-finding campaign. We also thank Jeff Diecks and the Open Source Security Foundation (OpenSSF) for supporting OSS-CRS as an OpenSSF sandbox project and promoting community outreach.

References

[1] Apache Software Foundation. Kafka, 2026. https://github.com/apache/kafka.

[2] A. Arya, O. Chang, J. Metzman, K. Serebryany, and D. Liu. OSS-Fuzz: continuous fuzzing for open source software, 2016. https://github.com/google/oss-fuzz.

[3] D. Bass. One of the Best Hackers in the Country Is an AI Bot. Bloomberg, 2025. https://www.bloomberg.com/news/articles/2025-06-24/one-of-the-best-hackers-in-the-country-is-an-ai-bot.

[4] T. Becker, R. Goulden, Y. Kim, J. Kwon, S. Myung, T. Nighswander, and S. Seo. aixcc-afc-archive: Public source code release of theori's aixcc afc submission, Aug. 2025. URL https://github.com/theori-io/aixcc-afc-archive/.

[5] BerriAI. LiteLLM: Call 100+ LLM APIs in OpenAI format, 2026. https://github.com/BerriAI/litellm.

[6] A. Bianchi, K. Borgolte, J. Corbetta, F. Disperati, A. Dutcher, J. Grosen, P. Grosen, A. Machiry, C. Salls, Y. Shoshitaishvili, N. Stephens, G. Vigna, and R. Wang. Cyber grand shellphish. *Phrack*, 16 (70), June 2017. URL https://sites.cs.ucsb.edu/~vigna/publications/2017_Phrack_CyberGrandShellphish.pdf.

[7] BugBuster. 42-b3yond-6ug-crs: Bugbuster, our team's submission to the ai cyber challenge final competition, Aug. 2025. URL https://github.com/42-b3yond-6ug/42-b3yond-6ug-crs.

[8] Buttercup. Buttercup: finds and patches software vulnerabilities, Dec. 2024. URL https://github.com/trailofbits/buttercup.

[9] T. Byun, C. Aschermann, K. Y. Thng, W. Zhou, Y. Yang, L. Deason, and J. Saxe. Introducing AutoPatchBench: A Benchmark for AI-Powered Security Fixes, Apr. 2025. URL https://engineering.fb.com/2025/04/29/ai-research/autopatchbench-benchmark-ai-powered-security-fixes/.

[10] C. Chen, B. Dolan-Gavitt, and Z. Lin. ELFuzz: Efficient Input Generation via LLM-driven Synthesis Over Fuzzer Space. In *Proceedings of the 34th USENIX Security Symposium (Security)*, Seattle, WA, Aug. 2025.

[11] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[12] DARPA. Cyber Grand Challenge, 2016. https://www.darpa.mil/research/programs/cyber-grand-challenge.

[13] DARPA. AIxCC Archive, 2025. https://archive.aicyberchallenge.com/.

[14] DARPA. AI Cyber Challenge (AIxCC), 2025. URL https://aicyberchallenge.com/. Accessed: 2025-09-09.

[15] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, pages 423–435. Association for Computing Machinery, 2023. doi: 10.1145/3597926.3598067.

[16] A. Dutcher. pydatatask: Library for constructing data-centric processing pipelines , 2022. https://github.com/rhelmot/pydatatask.

[17] 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.

[18] 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)*, CCS '22. ACM, November 2022.

[19] L. Franceschi-Bicchierai. Google says its AI-based bug hunter found 20 security vulnerabilities. TechCrunch, 2025. https://techcrunch.com/2025/08/04/google-says-its-ai-based-bug-hunter-found-20-security-vulnerabilities/.

[20] Y.-F. Fu, J. Lee, and T. Kim. autofz: Automated Fuzzer Composition at Runtime. In *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[21] E. Güler, P. Görz, E. Geretto, A. Jemmett, S. Österlund, H. Bos, C. Giuffrida, and T. Holz. Cupid : Automatic fuzzer selection for collaborative fuzzing. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2020.

[22] A. Hazimeh, A. Herrera, and M. Payer. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3): 49:1–49:29, Nov. 2020. doi: 10.1145/3428334. URL https://dl.acm.org/doi/10.1145/3428334.

[23] Helm. Helm: The Kubernetes Package Manager, 2026. https://github.com/helm/helm.

[24] C. Intelligence. Jazzer: Coverage-guided, in-process fuzzing for the jvm, 2025. https://github.com/CodeIntelligenceTesting/jazzer.

[25] T. Kim, H. Han, S. Park, D. R. Jeong, D. Kim, D. Kim, E. Kim, J. Kim, J. Wang, K. Kim, S. Ji, W. Song, H. Zhao, A. Chin, G. Lee, K. Stevens, M. Alharthi, Y. Zhai, C. Zhang, J. Jang, Y. Jang, A. Askar, D. Kim, F. Fleischer, J. Cho, J. Kim, K. Ko, I. Yun, S. Park, D. Baik, H. Lee, H. Heo, M. Gwon, M. Lee, M. Baek, S. Min, W. Kim, Y. Jin, Y. Park, Y. Choi, J. Jung, G. Lee, J. Jang, K. Kim, Y. Cha, and Y. Kim. ATLANTIS: AI-driven Threat Localization, Analysis, and Triage Intelligence System. *arXiv*, Sept. 2025. doi: 10.48550/arXiv.2509.14589. URL https://arxiv.org/abs/2509.14589. Cross-listed in cs.AI.

[26] W. Kim, S. Min, M. Gwon, D. Baik, H. Lee, H. Heo, M. Lee, M. W. Baek, Y. Jin, Y. Park, Y. Choi, T. Kim, S. Park, and I. Yun. Patchisland: Orchestration of llm agents for continuous vulnerability repair, 2026. URL https://arxiv.org/abs/2601.17471.

[27] Y. Kim, S. Shin, H. Kim, and J. Yoon. Logs In, Patches Out: Automated Vulnerability Repair via Tree-of-Thought LLM Analysis. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 4401–4419, 2025.

[28] kubernetes. kubernetes: Production-Grade Container Scheduling and Management, 2026. https://github.com/kubernetes/kubernetes.

[29] Kudu Dynamics. SHERPA: Security Harness Engineering for Robust Program Analysis, 2025. URL https://github.com/AIxCyberChallenge/sherpa.git. Developed as part of DARPA's AI Cyber Challenge (AIxCC).

[30] Lacrosse. afc-crs-lacrosse: Aixcc finals 2025 crs submission, Aug. 2025. URL https://github.com/siftech/afc-crs-lacrosse.

[31] H. Lee, Z. Zhang, H. Lu, and L. Zhang. SEC-bench: Automated benchmarking of LLM agents on real-world software security tasks. In *The thirty-ninth annual conference on neural information processing systems*, 2025. URL https://openreview.net/forum?id=QQhQIqons0.

[32] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, et al. UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *Proceedings of the USENIX Security Symposium (Security)*, pages 2777–2794. USENIX Association, Aug. 2021.

[33] D. Liu, O. Chang, J. Metzman, M. Sablotny, and M. Maruseac. OSS-Fuzz-Gen: LLM powered fuzzing via OSS-Fuzz, 2024. https://github.com/google/oss-fuzz-gen.

[34] X. Mei, P. S. Singaria, J. D. Castillo, H. Xi, Abdelouahab, Benchikh, T. Bao, R. Wang, Y. Shoshitaishvili, A. Doupé, H. Pearce, and B. Dolan-Gavitt. ARVO: Atlas of Reproducible Vulnerabilities for Open Source Software, Aug. 2024. URL http://arxiv.org/abs/2408.02153. arXiv:2408.02153 [cs].

[35] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury. Large Language Model Guided Protocol Fuzzing. In *Proceedings of the 2024 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2024.

[36] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya. FuzzBench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, pages 1393–1403, New York, NY, USA, Aug. 2021. Association for Computing Machinery. ISBN 978-1-4503-8562-6. doi: 10.1145/3468264.3473932. URL https://dl.acm.org/doi/10.1145/3468264.3473932.

[37] NIST. NVD — CVSS v3.1 Calculator, n.d. URL https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator. Accessed: 2026-02-28.

[38] OASIS. Static Analysis Results Interchange Format (SARIF) Version 2.1.0, 2023. https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html.

[39] OpenAI. Introducing Aardvark: OpenAI's Agentic Security Researcher. OpenAI, 2025. https://openai.com/index/introducing-aardvark/.

[40] PostgreSQL. Postgresql, 2026. https://github.com/postgres/postgres.

[41] T. L. Project. libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2025. Accessed: 2025-12-09.

[42] RabbitMQ. Rabbitmq-server: core server and tier 1 (built-in) plugins, 2026. https://github.com/rabbitmq/rabbitmq-server.

[43] Redis. Redis: preferred, fastest, and most feature-rich cache, data structure server, and document and vector query engine, 2026. https://github.com/redis/redis.

[44] S. Rudra. FFmpeg Calls Google's AI Bug Reports "CVE Slop". It's FOSS, 2025. https://itsfoss.com/news/ffmpeg-google-fiasco/.

[45] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[46] Z. Sheng, Q. Xu, J. Huang, M. Woodcock, H. Huang, A. F. Donaldson, G. Gu, and J. Huang. All You Need Is A Fuzzing Brain: An LLM-Powered System for Automated Vulnerability Detection and Patching, Sept. 2025. URL http://arxiv.org/abs/2509.07225. arXiv:2509.07225 [cs].

[47] W. Shi, H. Li, J. Yu, W. Guo, and X. Xing. Bandfuzz: A practical framework for collaborative fuzzing with reinforcement learning. In *Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing*, SBFT '24, page 55–56, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705625. doi: 10.1145/3643659.3648563. URL https://doi.org/10.1145/3643659.3648563.

[48] D. Stenberg. AI slop security reports submitted to curl. GitHub Gist, 2025. https://gist.github.com/bagder/07f7581f6e3d78ef37dfbfc81fd1d1cd.

[49] D. Stenberg. The end of the curl bug-bounty. Daniel Stenberg Blog, 2026. https://daniel.haxx.se/blog/2026/01/26/the-end-of-the-curl-bug-bounty/.

[50] T. B. S. team. From Naptime to Big Sleep: Using Large Language Models To Catch Vulnerabilities In Real-World Code. Project Zero, 2024. https://projectzero.google/2024/10/from-naptime-to-big-sleep.html.

[51] Team Atlanta. CRSBench: Cyber Reasoning System Benchmark Suite, 2026. https://github.com/sslab-gatech/crsbench.

[52] M. Walker. Machine vs. Machine: Lessons from the First Year of Cyber Grand Challenge. In *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C., Aug. 2015. USENIX Association.

[53] P. Wang, X. Liu, and C. Xiao. CVE-Bench: Benchmarking LLM-based Software Engineering Agent's Ability to Repair Real-World CVE Vulnerabilities. In L. Chiruzzo, A. Ritter, and L. Wang, editors, *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 4207–4224, Albuquerque, New Mexico, Apr. 2025. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.212. URL https://aclanthology.org/2025.naacl-long.212/.

[54] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang. Fuzz4ALL: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, pages 126:1–126:13. Association for Computing Machinery, 2024.

[55] Y. Yang, S. Yao, J. Chen, and W. Lee. Hybrid Language Processor Fuzzing via LLM-Based Constraint Solving. In *Proceedings of the 34th USENIX Security Symposium (Security)*, Seattle, WA, Aug. 2025.

[56] Z. Yu, Z. Guo, Y. Wu, J. Yu, M. Xu, D. Mu, Y. Chen, and X. Xing. PatchAgent: A Practical Program Repair Agent Mimicking Human Expertise. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 4381–4400, 2025.

[57] A. K. Zhang, N. Perry, R. Dulepet, J. Ji, C. Menders, J. W. Lin, E. Jones, G. Hussein, S. Liu, D. J. Jasper, P. Peetathawatchai, A. Glenn, V. Sivashankar, D. Zamoshchin, L. Glikbarg, D. Askaryar, H. Yang, A. Zhang, R. Alluri, N. Tran, R. Sangpisit, K. O. Oseleononmen, D. Boneh, D. E. Ho, and P. Liang. Cybench: A framework for evaluating cybersecurity capabilities and risks of language models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=tc90LV0yRL.

[58] C. Zhang, Y. Park, F. Fleischer, Y.-F. Fu, J. Kim, D. Kim, Y. Kim, Q. Xu, A. Chin, Z. Sheng, H. Zhao, B. J. Lee, J. Wang, M. Pelican, D. J. Musliner, J. Huang, J. Silliman, M. Mcdaniel, J. Casavant, I. Goldthwaite, N. Vidovich, M. Lehman, and T. Kim. SoK: DARPA's AI Cyber Challenge (AIxCC): Competition Design, Architectures, and Lessons Learned. *arXiv*, 2026. doi: 10.48550/arXiv.2602.07666. URL https://arxiv.org/abs/2602.07666. Cross-listed in cs.AI.

[59] K. Zhang, Z. Li, D. Wu, S. Wang, and X. Xia. Low-Cost and Comprehensive Non-textual Input Fuzzing with LLM-Synthesized Input Generators. In *Proceedings of the 34th USENIX Security Symposium (Security)*, Seattle, WA, Aug. 2025.

[60] S. Österlund, E. Geretto, A. Jemmett, E. Güler, P. Görz, T. Holz, C. Giuffrida, and H. Bos. Collabfuzz: A framework for collaborative fuzzing. In *Proceedings of the 14th European Workshop on Systems Security*, EuroSec '21, pages 1–7. Association for Computing Machinery, 2021. doi: 10.1145/3447852.3458720. URL https://doi.org/10.1145/3447852.3458720.

A Integrated CRS Components

Table 6: Each integrated CRS (Table 3), grouped by AIxCC team. Names match the entries in the OSS-CRS registry.

Team	Component	Capability	LLM	Language	Description
Vanilla fuzzer	`crs-libfuzzer`	**Finding**	×	C/C++	libFuzzer engine; minimal baseline CRS
	`crs-jazzer`	**Finding**	×	Java	Jazzer JVM fuzzer; minimal baseline CRS
ATLANTIS	`atlantis-multilang-given_fuzzer`	**Finding**	×	C/C++/Java	Base UniAFL fuzzer
	`atlantis-multilang-wo-concolic`	**Finding**	✓	C/C++/Java	UniAFL fuzzer with LLM mutator microservices
	`atlantis-c-deepgen`	**Finding**	✓	C/C++	LibAFL fuzzer with LLM seed generation
	`atlantis-c-bullseye`	**Finding**	✓	C/C++	AFL++ directed fuzzer
	`atlantis-java-main`	**Finding**	✓	Java	Jazzer sinkpoint-directed fuzzing
	`crs-prism`	**Fixing**	✓	C/C++/Java	Cyclic analyze–patch–evaluate agent
	`crs-vincent`	**Fixing**	✓	C/C++/Java	Root-cause and property-analysis patch agent
	`crs-multi-retrieval`	**Fixing**	✓	C/C++/Java	Two-step retrieval-based patch agent
BUGBUSTER	`42-directed`	**Finding**	×	C/C++	AFL++ directed fuzzer
	`42-seedgen`	**Finding**	✓	C/C++/Java	LLM seed-generation agent
BUTTERCUP	`buttercup-seed-gen`	**Finding**	✓	C/C++/Java	LLM seed-generation agent
ARTIPHISHELL	`crs-shellphish-c-fuzzers-libfuzzer`	**Finding**	×	C/C++	libFuzzer fuzzers
	`crs-shellphish-c-fuzzers-aflpp`	**Finding**	×	C/C++	AFL++ fuzzers
	`crs-shellphish-jvm-fuzzers`	**Finding**	×	Java	Jazzer fuzzer with LOSAN sanitizer
	`crs-shellphish-aijon`	**Finding**	✓	C/C++	LLM IJON annotation generation and fuzzing
	`crs-shellphish-discoveryguy`	**Finding**	✓	C/C++	LLM vulnerability discovery
	`crs-shellphish-grammar`	**Finding**	✓	C/C++/Java	Grammar-based fuzzer
	`crs-shellphish-quickseed`	**Finding**	✓	C/C++/Java	LLM seed-generation agent
ROBODUCK	`roboduck`	**Finding**	✓	C/C++/Java	LLM seed- and PoV-generation agent
FUZZINGBRAIN	`fuzzing-brain`	**Finding**	✓	C/C++/Java	LLM PoV-generation agent
LACROSSE	`lacrosse-seed-gen`	**Finding**	✓	C/C++/Java	LLM seed-generation agent
AI coding agent	`crs-bug-finding-claude-code`	**Finding**	✓	C/C++/Java	Claude Code coding agent for bug finding
	`crs-bug-finding-codex`	**Finding**	✓	C/C++/Java	Codex CLI coding agent for bug finding
	`crs-bug-finding-copilot-cli`	**Finding**	✓	C/C++/Java	Copilot CLI coding agent for bug finding
	`crs-bug-finding-gemini-cli`	**Finding**	✓	C/C++/Java	Gemini CLI coding agent for bug finding
	`crs-bug-finding-opencode`	**Finding**	✓	C/C++/Java	OpenCode coding agent for bug finding
	`crs-claude-code`	**Fixing**	✓	C/C++/Java	Claude Code coding agent for patch generation
	`crs-codex`	**Fixing**	✓	C/C++/Java	Codex CLI coding agent for patch generation
	`crs-copilot-cli`	**Fixing**	✓	C/C++/Java	Copilot CLI coding agent for patch generation
	`crs-gemini-cli`	**Fixing**	✓	C/C++/Java	Gemini CLI coding agent for patch generation
	`crs-opencode`	**Fixing**	✓	C/C++/Java	OpenCode coding agent for patch generation

Each entry in Table 3 is a team-level summary; in practice every finalist CRS is integrated as several independent subcomponents. Table 6 itemizes the bug-finding and bug-fixing components, their registry names, supported languages, and whether each uses LLMs.