WOOT '26 Artifact Appendix: 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*

A Artifact Appendix

A.1 Abstract

The artifact contains OSS-CRS, an open framework for developing, running, and composing cyber reasoning systems (CRSs). It includes the OSS-CRS codebase, configurations for running the Atlantis CRS, and PoCs used to demonstrate OSS-CRS's bug-finding and patching capabilities. The artifact supports a lightweight functionality test that runs without LLM credentials, as well as the full evaluation workflow for reproducing the paper's CRS results with appropriate LLM API keys.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

OSS-CRS runs natively on the evaluator's local machine and launches CRSs in *privileged* Docker containers. Evaluators should therefore run the artifact on a dedicated machine or virtual machine that does not contain sensitive data.

The full evaluation requires evaluators to use dedicated LLM API keys. The LLM keys supplied by the evaluator are provided to the OSS-CRS LiteLLM proxy, which is launched as part of the OSS-CRS infrastructure. Keys are exposed to the LiteLLM proxy, but never directly to the CRSs.

The vulnerability reproductions included in the artifact exercise bugs that have already been disclosed and fixed.

A.2.2 How to access

The artifact is archived on Zenodo at https://doi.org/10.5281/zenodo.20072038.

A.2.3 Hardware dependencies

x86_64 desktop or server with minimum 8 CPU cores and 16 GB memory.

A.2.4 Software dependencies

Python ($\geq$ 3.10), Docker, `git`, and the `uv` Python package manager.

LLM API keys (i.e., OpenAI, Anthropic, and Google) is required for full reproduction of the evaluation, since the CRSs invoke LLMs.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Download `oss-crs.zip`, `oss-fuzz.zip`, and `oss-crs-pocs.zip` from Zenodo. Then extract the archives in the same directory.

```
unzip oss-crs.zip
unzip oss-fuzz.zip
unzip oss-crs-pocs.zip
cd oss-crs
```

A.3.2 Basic Test

The basic test runs libfuzzer through OSS-CRS, which does not require LLM tokens.

```
COMPOSE=example/crs-libfuzzer/compose.yaml
PROJ=../oss-fuzz/projects/jq

uv run oss-crs prepare --compose-file $COMPOSE
uv run oss-crs build-target \
  --compose-file $COMPOSE --fuzz-proj-path $PROJ
uv run oss-crs run \
  --compose-file $COMPOSE --fuzz-proj-path $PROJ \
  --target-harness jq_fuzz_execute --timeout 60
```

The third command (`run`) fuzzes jq for one minute and prints how many seeds were produced in the short campaign.

If `build-target` or `run` fails due to upstream OSS-Fuzz breakage, you can run the test against our in-tree mock-c project fixture:

```
FIXT=oss_crs/tests/integration/fixtures
PROJ=$FIXT/mock-c-project

uv run oss-crs build-target \
  --compose-file $COMPOSE --fuzz-proj-path $PROJ
uv run oss-crs run \
  --compose-file $COMPOSE --fuzz-proj-path $PROJ \
  --target-harness fuzz_parse_buffer --timeout 30
```

To adjust LLM settings for LLM-based CRSs, edit `example/CRS/litellm-config.yaml`; this file follows the LiteLLM proxy configuration format documented at https://docs.litellm.ai/docs/proxy/config_settings.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): OSS-CRS discovers vulnerabilities beyond classic memory corruption, including undefined behavior (null-pointer arithmetic, numeric-conversion overflow) and logic bugs (schema-validation cycles). This is Finding 1 of the paper (§ Zero-day Results), supported by the zero-day vulnerability table.

(C2): Patches generated by OSS-CRS 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. This is Finding 2 of the paper (§ Zero-day Results), illustrated by the patch listings cited there.

A.4.2 Experiments

(E1): Bug finding with OSS-CRS [30 human-minutes + 5 compute-hour + $40 LLM API]: run three OSS-CRS bug-finding campaigns to validate that the framework can reproduce the paper's vulnerability-discovery workflow.

Preparation: Prepare the Atlantis CRS image and build the jq and libyang targets.

```
# LLM keys
export ANTHROPIC_API_KEY=sk-...
export GEMINI_API_KEY=sk-...
export OPENAI_API_KEY=sk-...

CF=example/atlantis-multilang-wo-concolic/compose.yaml
PROJ_BASE=../oss-fuzz/projects

uv run oss-crs prepare --compose-file $CF
uv run oss-crs build-target \
  --sanitizer undefined \
  --compose-file $CF \
  --fuzz-proj-path $PROJ_BASE/jq
uv run oss-crs build-target \
  --compose-file $CF \
  --fuzz-proj-path $PROJ_BASE/libyang
```

Execution: Two of the bug-finding campaigns are against the jq harnesses. If needed, increase the timeout for the Docker image building overhead.

```
uv run oss-crs run \
  --sanitizer undefined \
  --compose-file $CF \
  --fuzz-proj-path $PROJ_BASE/jq \
  --timeout 600 \
  --target-harness jq_fuzz_execute

uv run oss-crs run \
  --sanitizer undefined \
  --compose-file $CF \
  --fuzz-proj-path $PROJ_BASE/jq \
  --timeout 600 \
  --target-harness jq_fuzz_arith
```

The third campaign runs against libyang. Since this bug is not shallow, it requires a long campaign which may not deterministically succeed.

```
# Full campaign
uv run oss-crs run \
  --compose-file $CF \
  --fuzz-proj-path $PROJ_BASE/libyang \
  --timeout 14400 \
  --target-harness lys_parse_mem
```

Results: The `oss-crs run` output reports generated PoVs and campaign artifacts. In case the full campaign of libyang does not find a PoV, the seeds produced still demonstrate the LLM capabilities of creating structured formats to aid fuzzing. These results should align with C1 by showing CRS-discovered undefined-behavior and logic bugs in the evaluated targets.

(E2): Patching JQ [30 compute-minutes + $10 LLM API]: run patching CRSs on two jq PoCs to validate that OSS-CRS can drive CRS-generated repairs that align with C2.

Preparation: Prepare the Codex-based patching CRS and build the jq target with the undefined-behavior sanitizer.

```
# LLM keys
export OPENAI_API_KEY=sk-...

COMPOSE=example/crs-codex/compose.yaml
PROJ=../oss-fuzz/projects/jq
POC_DIR=../oss-crs-pocs

uv run oss-crs prepare --compose-file $COMPOSE
uv run oss-crs build-target \
  --sanitizer undefined \
  --compose-file $COMPOSE \
  --fuzz-proj-path $PROJ
```

Execution: Run the patching CRS on the two jq PoCs.

```
# Patch the null pointer poc
uv run oss-crs run \
  --sanitizer undefined \
  --compose-file $COMPOSE \
  --fuzz-proj-path $PROJ \
  --target-harness jq_fuzz_execute \
  --pov $POC_DIR/jq/execute_null_pointer

# Patch the dtoi overflow
uv run oss-crs run \
  --sanitizer undefined \
```

```
--compose-file $COMPOSE \
--fuzz-proj-path $PROJ \
--target-harness jq_fuzz_arith \
--pov $POC_DIR/jq/arith_mod_overflow
```

Results: The `oss-crs run` output reports where each generated patch diff is stored. Inspect those diffs and confirm that they match C2: the jq repairs are minimal, single-location patches that address the corresponding root causes.

A.5 Notes on Reusability

OSS-CRS is reusable beyond the CRSs evaluated in the paper: it provides a standard framework for developing, running, and composing new CRSs. Developers and researchers can use this standard to port an existing CRS or build a new one that targets OSS-Fuzz-style projects. For details on CRS development, see the CRS development guide included in the Zenodo archive at `oss-crs/docs/crs-development-guide.md`.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/woot.