SoK: DARPA's AI Cyber Challenge (AIxCC):
Competition Design, Architectures, and Lessons Learned

Cen Zhang[†*§] *Younggi Park*[♮] *Fabian Fleischer*[†] *Yu-Fu Fu*[†] *Jiho Kim*[†] *Dongkwan Kim*[*§]
Youngjoon Kim[†] *Qingxiao Xu*[‡] *Andrew Chin*[†] *Ze Sheng*[‡] *Hanqing Zhao*[†]
Michael Pelican[¶] *David J. Musliner*[¶] *Jeff Huang*[‡] *Jon Silliman*[‖] *Mikel Mcdaniel*[‖]
Jefferson Casavant[‖] *Isaac Goldthwaite*[‖] *Nicholas Vidovich*[‖] *Matthew Lehman*[‖] *Taesoo Kim*[†*]

[†] Georgia Institute of Technology, [‡] Texas A&M University, [♮] Independent Researcher,
[¶] Smart Information Flow Technologies (SIFT), [‖] Kudu Dynamics, [*] Microsoft

Usenix Security '26

Background: AIxCC

- **DARPA/ARPA-H competition**, 2023–2025
  - Goal: fully autonomous **Cyber Reasoning Systems (CRSs)** that find and fix bugs
  - AI partners: Anthropic, Google, Microsoft, OpenAI
- **Final at DEF CON 2025**: 7 finalists out of 42 entrants
  - ~143 hours of fully autonomous operation
  - 53 challenge projects from real-world C and Java OSS
- **Budget per team**: $85K Azure compute + $50K LLM API

Background: AIxCC



Competition Design

- Challenges mirror a **real GitHub workflow** (release, PR, alert)

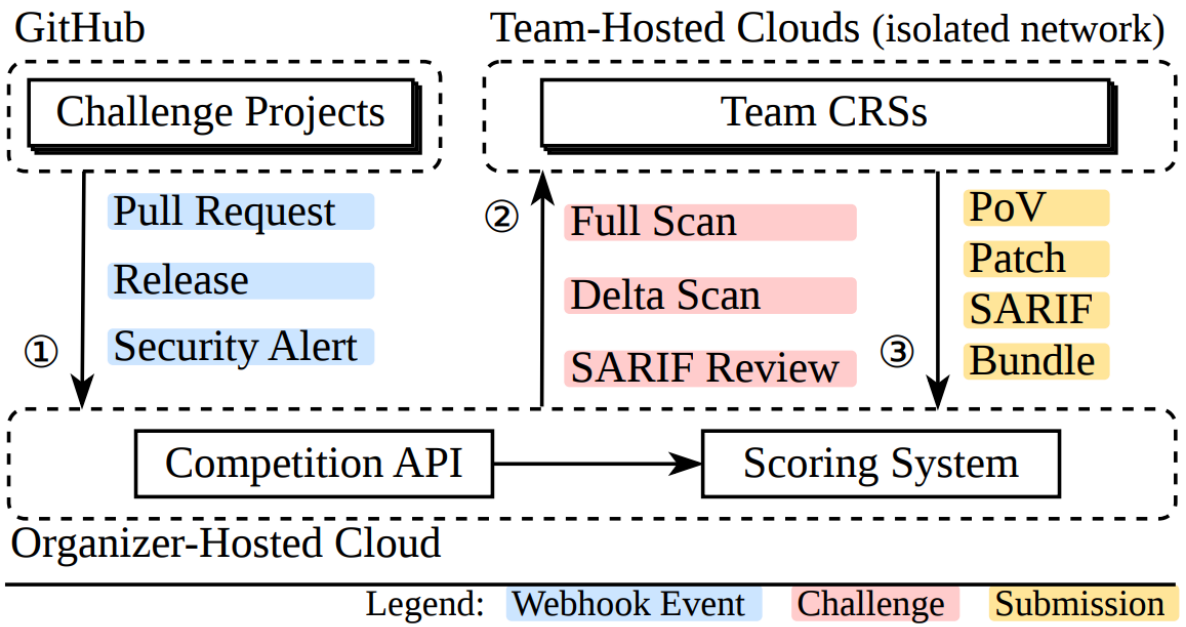


Figure 1: Final competition workflow: ① triggered by webhook, ② challenge dispatched, ③ result submitted.

Given Challenges

- **Full Scan** (new release or tag)
  - Find and patch bugs anywhere in the codebase
- **Delta Scan** (pull request)
  - Analyze only the incoming code change
- **SARIF Review** (security alert)
  - Judge each static-analysis alert as valid or invalid

What is SARIF?

- **S**tatic **A**nalysis **R**esults **I**nterchange **F**ormat
- JSON standard for static-analyzer output
  - Rule, location, and message for each finding
- In AIxCC: organizers send SARIF alerts; the CRS judges whether each one is a real bug

```
{
  "version": "2.1.0",
  "$schema": "https://json.schemastore.org/sarif-2.1.0.json",
  "runs": [
    {
      "tool": {
        "driver": {
          "name": "ExampleScanner",
          "informationUri": "https://example.com/scanner",
          "rules": [
            {
              "id": "EX1001",
              "shortDescription": { "text": "Unused local variable." }
            }
          ]
        }
      },
      "results": [
        {
          "ruleId": "EX1001",
          "level": "warning",
          "message": { "text": "The local variable 'count' is never read." },
          "locations": [
            {
              "physicalLocation": {
                "artifactLocation": { "uri": "src/parser.c" },
                "region": { "startLine": 42, "startColumn": 7 }
              }
            }
          ]
        }
      ]
    }
  ]
}
```

CRS Submissions

Submission	Points	What it is
Proof of Vulnerability (PoV)	1–2	Input that triggers the bug
Patch	3–6	Fix that removes the bug and keeps functionality
SARIF assessment	0.5–1	Verdict on whether a static-analysis alert is valid
Bundle	−7 to +7	Links PoV, patch, and SARIF for the same bug

Scoring (What is a “Good CRS”?)

- **Fixing beats finding**: Patch > PoV > SARIF
- **Bundles cut both ways** (−7 to +7)
  - Correct links save triage effort; wrong links waste it
- **Time decay**: full points if immediate, half at the last minute
- **Accuracy multiplier**: score scaled by submission accuracy
  - 90% accurate → negligible penalty
  - 50% → −6%; 40% → −13% (steep)

Challenge Projects

- **24 OSS repos** (14 C, 10 Java)
  - 16K to 4.9M lines of code
- Most bugs (CPVs) are **hand-injected**
  - Avoids LLM training-data contamination
- A few **real 0-days** surfaced while building the challenges

Table 1: Overview of open-source repositories and challenge projects (CPs) in the final. □: full-mode; ▲: delta-mode. *5 unharnessed CPs are excluded: freertos-kernel, jt808, lwip, openssl, sms4j. †Averaged across CPs.

Lang.	Project	Abbr.	# CPs		# CPVs	# Harn.	SLOC†
C	curl	cu		5 ▲	6	17	240K
	dav1d	da	1 □		1	1	261K
	freerdp	fp		3 ▲	2	7	457K
	little-cms	cm	1 □	1 ▲	2	15	87K
	libavif	av		2 ▲	1	8	44K
	libexif	ex		2 ▲	2	2	16K
	libxml2	lx		1 ▲	1	11	201K
	mongoose	mg	1 □	3 ▲	3	1	364K
	ndpi	nd	1 □		0	55	136K
	openssl	os	1 □		0	30	909K
	shadowsocks-libev	ss	1 □		5	1	19K
	systemd	sd	1 □		4	47	740K
	wireshark	ws	1 □	6 ▲	12	47	4901K
	xz	xz	1 □		1	4	41K
Java	commons-compress	cc		5 ▲	5	16	76K
	dcm4che	dc	1 □		0	1	105K
	dicoogle	dg	1 □		0	1	21K
	healthcare-data-harmonization	hc	1 □		0	1	53K
	hertzbeat	hb	1 □		0	1	78K
	jsoup	js	1 □		0	2	36K
	logging-log4j2	lj		1 ▲	1	1	54K
	pdfbox	pb	1 □	1 ▲	9	6	167K
	poi	po	1 □	1 ▲	7	17	433K
	tika	tk		1 ▲	1	9	188K
Total	**24***		**16*** □	**32** ▲	**63**	**301**	

CRS Teams

Table 2: CRS Teams, ordered top-to-bottom by final score (descending; see Table 7). †: Full name "All You Need IS A Fuzzing Brain". ‡: Orchestration code only. §: Uses LiteLLM [10] for multi-provider routing.

ID	Team	CRS	Bg	Lang‡	LLM Lib
AT	Team Atlanta	ATLANTIS	Mixed	Py,Rust	LangGraph [33]§
TB	Trail of Bits	BUTTERCUP	Industry	Py	LangGraph
TI	Theori	ROBODUCK	Industry	Py,Rust	Self-built§
FB	Fuzzing Brain†	FUZZINGBRAIN	Academic	Py,Go	× §
SP	Shellphish	ARTIPHISHELL	Academic	Py	Self-built§
42	42-b3yond-6ug	BUGBUSTER	Academic	Py,Go	LangChain [13]§
LC	Lacrosse	LACROSSE	Industry	Py,Lisp	DSPy

Competition Results

ID	Team	CRS
AT	Team Atlanta	ATLANTIS
TB	Trail of Bits	BUTTERCUP
TI	Theori	ROBODUCK
FB	Fuzzing Brain†	FUZZINGBRAIN
SP	Shellphish	ARTIPHISHELL
42	42-b3yond-6ug	BUGBUSTER
LC	Lacrosse	LACROSSE

Table 7: AFC Score Breakdown. Columns are ordered left-to-right by final score (descending). Pen. stands for penalties.

		AT	TB	TI	FB	SP	42	LC
PoV	C	**52.6**	31.0	22.7	22.6	31.4	49.1	1.5
	Java	27.0	21.3	**31.6**	29.7	16.5	21.1	0.0
	Sum	**79.6**	52.4	54.3	52.3	47.8	70.1	1.5
Patch	C	**113.5**	74.2	51.8	45.3	40.5	9.7	4.9
	Java	**57.5**	26.7	49.8	23.5	13.8	4.5	0.0
	Sum	**171.0**	100.9	101.6	68.8	54.3	14.2	4.9
SARIF	C	5.0	1.0	3.9	4.7	7.5	**8.7**	0.0
	Java	1.0	0.0	1.0	**1.5**	1.0	1.0	0.0
	Sum	6.0	1.0	4.9	6.2	8.5	**9.7**	0.0
Bundle	C	**99.6**	49.4	26.0	27.6	18.3	7.8	3.2
	Java	**36.6**	15.7	23.8	-1.1	7.0	3.2	0.0
	Sum	**136.2**	65.1	49.8	26.4	25.3	11.0	3.2
Total	C	**270.6**	155.6	104.4	100.2	97.6	75.4	9.6
	Java	**122.1**	63.8	106.3	53.5	38.3	29.7	0.0
	Pen.	-0.4	-0.6	-16.3	-13.5	**-0.1**	-0.3	-1.1
	Final	**392.8**	219.4	210.7	153.7	135.9	105.0	9.6

PoV Generation Strategy

Two pipelines: LLM-assisted fuzzing, plus LLM agents that find bugs and write PoVs directly

Table 3: PoV Generation Techniques Beyond OSS-Fuzz Defaults. ○: non-LLM; ●: LLM-enhanced; ✓: present; blank: absent. Extended details: Table 11.

		AT	TB	TI	FB	SP	42	LC
Fuzzing Pipeline	Pre-Comp Corpus	●		○		●	●	○
	Seed Gen Agent	●	●	●		●	●	●
	├ Bootstrap	●	●			●	●	●
	├ Solve cov blocker	●	●	●		●		
	├ Mutator/generator	●						
	└ Grammar-aware	●		●		●		
	Engine Refinement	●				●	○	○
	├ Semantic feedback					●		
	├ Improved sanitizer	○				○		
	├ Dict Gen	●				○	○	○
	└ Directed fuzzing	○					○	
	Concolic Fuzzing	○						
	Parallel Fuzzing	○	○	○	○	○	○	○
	├ Corpus sync	○	○	○		○	○	○
	├ Added C fuzzers	○				○	○	○
	└ Added JVM fuzzers	○						
LLM-Based PoV Gen Pipeline	Bug Cand. I.D.	●		●	●	●		●
	├ Candidate filter	●		●	●	○		●
	└ Non-PoV Gen usage	✓		✓		✓		✓
	PoV Gen Agent	●	●	●	●	●		
	├ With CWE guidance	●	●		●	●		
	└ Reach-then-exploit	●		●		●		
Pipeline Co-op	LLM PoV Gen → Fuzz	✓	✓	✓	✓	✓		
	Fuzz → LLM PoV Gen	✓		✓		✓		
PoV Sub.	Deduplication	○	○	●	●	○	○	○
	ASAP Submission	✓	✓	✓	✓	✓	✓	✓

Patch Generation

Shared loop: [RCA] → Generate → Validate → Dedup → Submit. Teams differ in agent design and validation

Table 4: Patch Generation Techniques. ●: present; blank: no custom implementation; –: not applicable; 1/N/∗: single/multiple/all PoVs; †all teams use sanitizer/crash reports and failed patch feedback. Extended details: Table 12.

		AT	TB	TI	FB	SP	42	LC
Agent Arch.	Multi-Arch	●				●		
	Multi-Agent		●	●				
	Single-Agent				●		●	●
RCA	Standalone RCA	●	●	●		●		
	├ Multi-PoV RCA		●	●		●		
	└ Non-LLM RCA					●		
Generation	Contextualization†	●	●	●	●	●	●	●
	├ Code Indexer	●	●	●		●	●	
	├ SAST	●		●	●	●		
	├ CWE Guidance				●		●	
	├ Fine-tuned LLM	●						
	├ Agentic Code Search	●	●	●	●	●	●	●
	├ Dynamic Info	●		●	●	●		
	└ PoV Bytes	●		●				●
	LLM Reflection	●	●	●		●		●
	No-PoV Patch Generation			●	●			●
Validation	Basic Checks	●	●	●	●	●	●	●
	├ Build	●	●	●	●	●	●	●
	├ PoV Test (Gen)	1	N	∗	1	N	∗	1
	└ Proj. Tests	●	●	●	●	●		●
	PoV Test (Submit)	∗	N	∗	N	∗	∗	1
	LLM as Judge	●			●	●		
	Post-patch Fuzz				●	●		
	Rebuild Optimization	●				●		
Dedup. & Sub.	Min. Patch Set Calc.	●	●			●	●	
	No-PoV Delayed Sub.	–	–	●	●	–	–	●

SARIF Submission

Three kinds of evidence: match a PoV, match a bug candidate, or ask an LLM judge

Table 5: SARIF Submission Strategies. ✓/×: submit *Correct*/*Incorrect*. Full details at [9].

CRS	Category	Submission Strategy Overview
AT, TB	PoV-centric	N: Match Any PoV —Y→ ✓
FB	PoV-centric	Match Any PoV —Y→ ✓; N: LLM As Judge —Y→ ✓
TI	Bug-cand-centric	Match Bug Cand. —Y/N→ ✓/× (1st); N: Match Bug Cand. —Y→ ✓ (2nd)
SP, LC	LLM-judge-centric	LLM As Judge —Y/N→ ✓/×
42	LLM-judge-centric	LLM As Judge —Y/N→ ✓/×; N: Match Any PoV —Y→ ✓ (UNK)

Bundling Strategies

Table 6: Bundling Pairing Strategies. ●: used; blank: not used.

Pairing	Source	AT	TB	TI	FB	SP	42	LC
PoV-Patch	PoV-based Patch	●	●	●	●	●	●	●
	Match No-PoV Patch			●				
PoV-SARIF	SARIF Validation	●	●	●	●			
	SARIF-guided PoV				●	●		
Patch-SARIF	Bug Candidate DB			●				
	SARIF-guided Patch				●			

Competition Results: Stability Issues

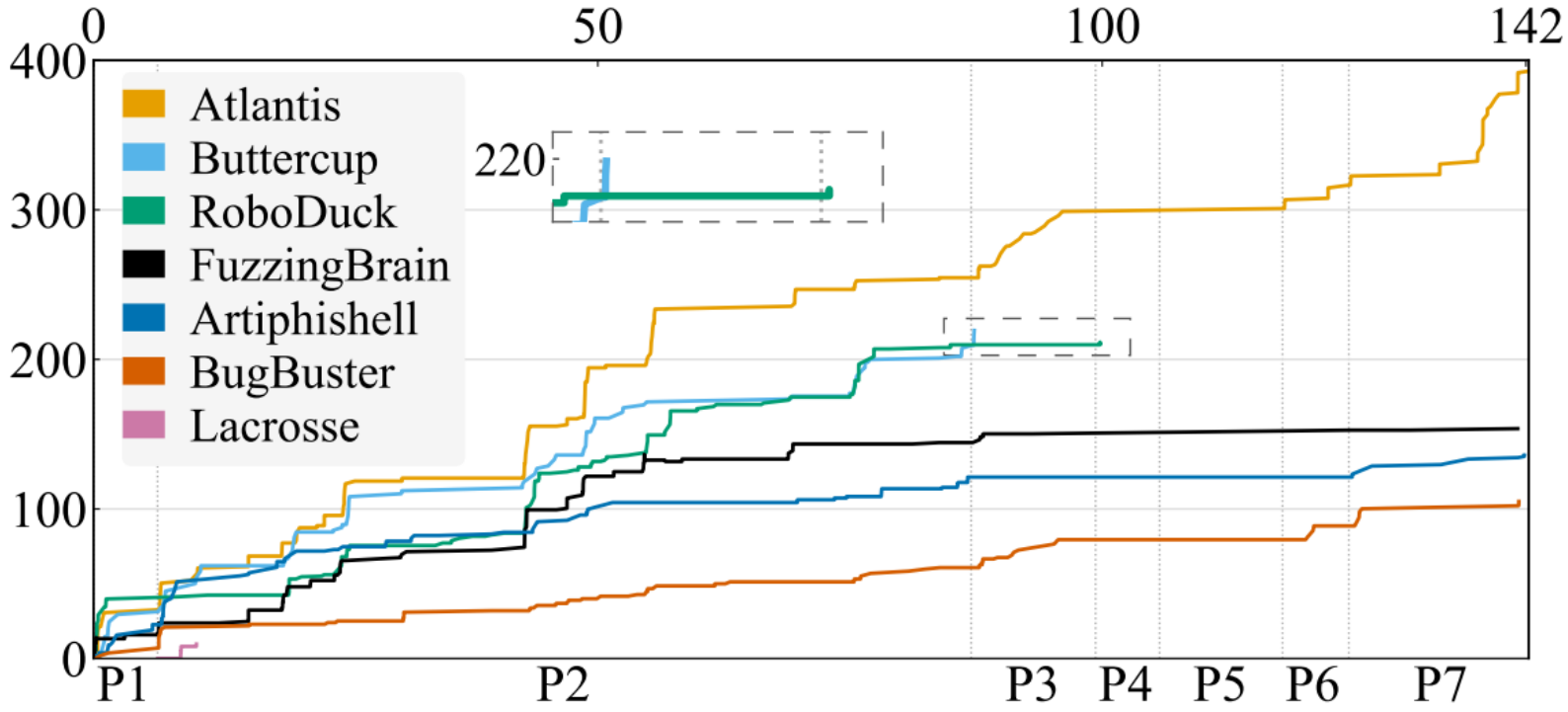


Figure 2: Score per time (top) and phase (bottom) axes.

Why did some teams fail?

- **TB, TI**: plateaued after P3, likely on the wireshark CP
  - ~1 TB of build artifacts exhausts disk on CRSs without disk management
- **LC**: unrecoverable OOM crash on its master scheduler
- **SP**: several system issues, e.g., wrong LLM budget config
- **AT**: silently failed on all poi (Java) challenges

Winning needs research, engineering, and strategy. Stability mattered most, and many teams lacked it.

Measuring CRS Capability

- **Scores alone can't measure capability**
  - Outages and strategy choices also drive the score
- Idea: annotate which CPVs are **solvable** with simple baselines
  - **Parallel Fuzzing (PF)**: non-LLM bug finding
  - **MultiRetrieval (MR)**: minimal agentic code-search-and-patch loop
  - **Claude Code (CC)**: general-purpose coding agent
- Compare CRS results against these baselines, per CPV

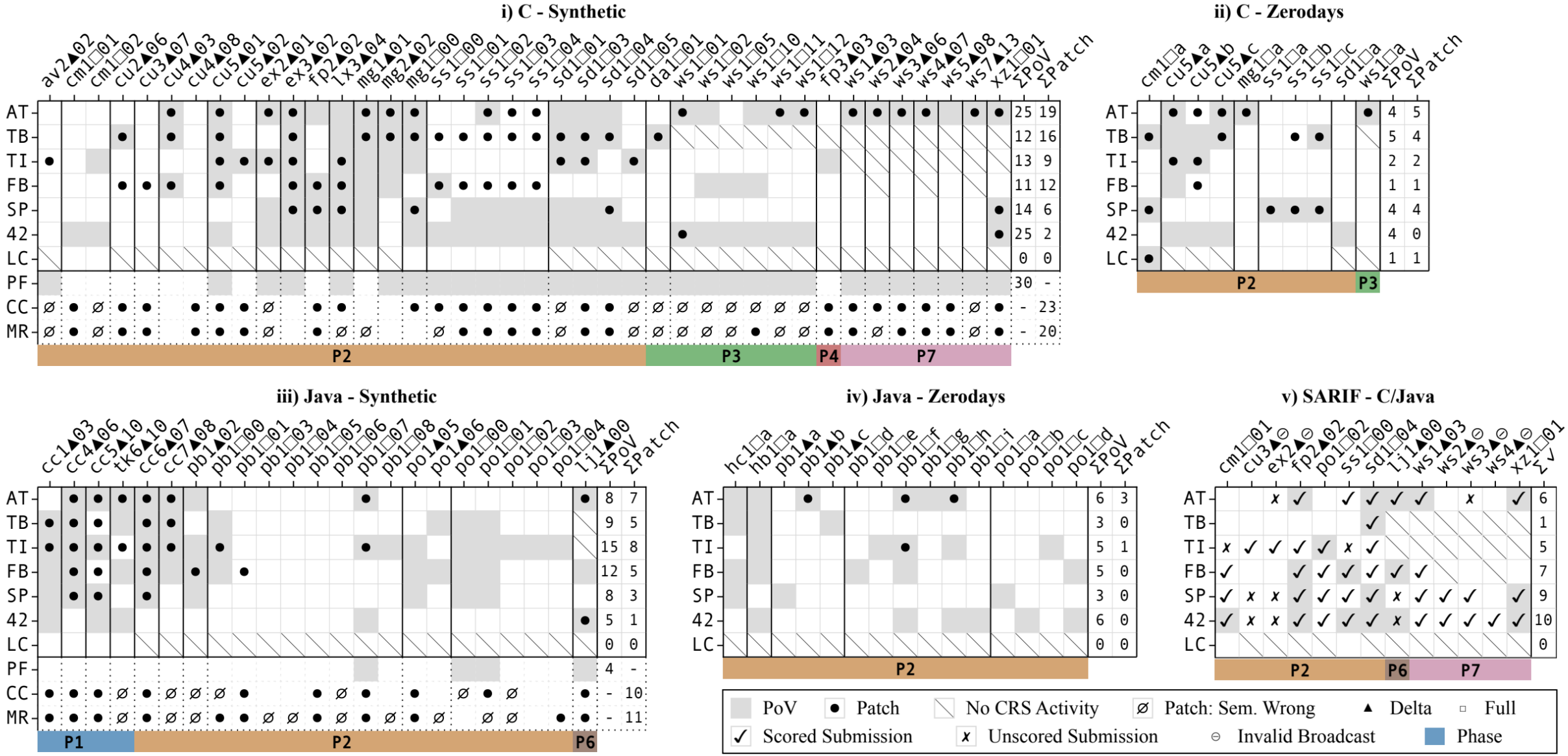


Figure 3: Team performance per CPV. Matrices i)–iv): detected/patched CPVs and 0-days; v): SARIF assessment. Diagonal: no CRS log messages. PF/CC/MR rows are annotations of 3-run union results. In patch: ● valid; ∅/blank: failed manual/automated validation. SARIF: invalid (⊖) expects *Incorrect*; ✓/✗ = CRS judgment *Incorrect*/*Correct*; CPV abbreviations in Appendix B.

PoV Generation: CRS > PF

- Six CRSs each found **7–16 CPVs** that parallel fuzzing missed
- **Targeted detection**
  - LLMs read the diff and craft triggering inputs directly
- **Complex input grammars**
  - LLMs produce inputs in structured formats (e.g., PDF)
- **Logical constraints**
  - LLMs solve path conditions that block fuzzer feedback

PoV Generation: PF > CRS

- Misses came from **pipeline gaps**, not detection ability
- **Broken dependencies**: non-standard CP setups break CRS init
- **Heavy builds**: wireshark needs ~1 TB of disk
- **Reproduction mismatch**: stateful bugs need repeated runs
- **Wrong sanitizer**: e.g., the bug needs UBSan
- **Coarse dedup**: bugs differing only by line number merged

Patch Generation: CRS > MR/CC

- CRS patches are more often **semantically correct**
  - High-accuracy CRSs: **16–21%** incorrect vs. MR/CC: **38–46%**
  - CRSs fixed 16 CPVs that MR/CC could not

Table 10: Submission statistics.

	Counted Submissions				Accuracy (%)				AM
Team	**PoV**	**Patch**	**SARIF**	**Bndl**	**PoV**	**Patch**	**SARIF**	**Bndl**	**Penalty**
AT	43	37	8	36	100	83.8	75.0	97.2	0.1%
TB	31	24	1	19	90.3	79.2	100	100	0.3%
TI	55	63	7	19	61.8	31.7	62.5	89.5	7.2%
FB	35	60	7	13	80.0	23.3	87.5	61.5	8.1%
SP	31	11	12	7	90.3	100	69.2	100	0.1%
42	45	4	13	4	91.1	75.0	76.9	100	0.3%
LC	1	3	0	1	100	33.3	—	100	10.7%

Patch Generation: MR/CC > CRS

- CRSs missed **9 CPVs** that MR/CC could patch
- **Causes are operational, not capability gaps**
  - Missing PoVs never triggered the patcher
  - Scheduling pressure and system instability
- **Configuration trade-offs**
  - AT's 30-min per-CPV timeout was used up by the wireshark build

Limitations

- **Taxonomy interpretation**
  - Two experts cross-checked code and interviewed finalists; misreadings may remain
- **Annotation accuracy**
  - PF/MR/CC labels only approximate which CPVs are solvable
- **No ablation**
  - Comparing techniques fairly needs them ported to one platform

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

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

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

https://www.usenix.org/conference/woot26/presentation/chin

This paper is included in the Proceedings of the 20th USENIX WOOT Conference on Offensive Technologies.

August 10–11, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-57-1

Open access to the Proceedings of the 20th USENIX WOOT Conference on Offensive Technologies is sponsored by USENIX.

OSS-CRS: Open Source Cyber Reasoning System Framework

OSS-CRS is a standard orchestration framework for building and running LLM-based autonomous bug-finding and bug-fixing systems (Cyber Reasoning Systems).

Why OSS-CRS?

- **Standard CRS Interface** — OSS-CRS defines a unified interface for CRS development. Build your CRS once following the development guide, and run it across different environments (local, Azure, ...) **without any modification**.
- **Effortless Targeting** — Run any CRS against projects in OSS-Fuzz format. If your project is compatible with OSS-Fuzz, OSS-CRS can orchestrate CRSs against it out of the box.
- **Ensemble Multiple CRSs** — Compose and run multiple CRSs together in a single campaign to combine their strengths and maximize bug-finding and bug-fixing coverage.
- **Resource Control** — Manage CPU limits and LLM budgets per CRS to keep costs and resources in check.
- **Multi-Environment Support** — Run locally today; deploy to Azure (coming soon) with zero changes to your CRS.