AIxCC ASC Technical Report

Prepared by team Theori

Theori's CRS follows the design described in our registration paper. The design is "LLM first", with most of the decisions being driven by LLMs, but with guardrails and access to tools which are specifically set up to help the LLMs accomplish their goals.

Section A. CRS Design Strategy

A.1 Vulnerability Discovery Strategy

Our team's vulnerability discovery strategy was primarily based on having LLMs construct POVs. This was performed using LLM agents with the ability to read and modify the source code, and run attempted POVs. All POVs generated directly by LLMs are written in Python and constructed in two parts: an agent produces a "buffer generating function" for the harness to produce a higher level of abstraction than simply "opaque buffer of bytes", then another agent attempts to create a POV that produces inputs for that function.

Additionally, our CRS used fuzzing when possible. The harnesses were compiled based on hardcoded rules to attempt to produce interfaces with which our CRS could interface. If the CRS was able to produce valid harnesses, these would run in the background continuously. Seeds were explicitly produced by LLM agents dedicated to producing testcases that would be "near" each commit with the intention of improving coverage over the areas of interest. Additionally, any POV attempt generated by the CRS which failed to trigger a sanitizer was also added to the seed pool for further fuzzing.

Each POV which was confirmed to trigger a sanitizer was tested to determine which commit introduced the bug, ensuring the information that would be submitted to the scoring API was confirmed locally.

We are unable to evaluate a breakdown of which components produced POVs in the ASC without access to the logs produced by our CRS.

A.2 Vulnerability Patching Strategy

Vulnerability patching was again driven by LLM agents. This task only ran after a valid POV was produced and some time had passed for any additional bugs to be produced by the CRS to minimize the likelihood that a GP would be submitted that did not adequately secure the code. The patching agents were given information about the commit in which the bug occurred that the vulnerability finding agents had produced, as well as the actual vulnerability which triggered the bug (either as Python source or as a blob produced by a fuzzer). The patcher was given the ability to read, write, compile, and test patches, with tests failing due to functionality test failures or failure to prevent a sanitizer triggering when run with the produced POV.

A.3 LLM Usage Strategy

LLMs were heavily used in our CRS. We believe they present the best option for making non-trivial patches to source code, and represent a robust strategy for POV production across languages and codebases which cannot easily be built and instrumented. The highest success in our testing came from using Anthropic's Claude 3.5 Sonnet, which we then preferred for patching and POV production. Due to rate limit constraints, we also used OpenAI's GPT 4o in a few circumstances: when performing the initial fuzzer seeding, as this was done in parallel with other tasks, and we did not wish to hit the rate limit; if there were more than 24 commits, as we then analyzed commits in parallel rather than serially; if we needed to compress LLM context in order to lessen our token rate usage; and randomly to re-run failed POV production jobs.
Our CRS also used Anthropic's Claude 3 Haiku for very simple workloads of summarizing large amounts of text (for example, summarizing the errors produced by a compilation failure).

The Theori CRS has a number of techniques to keep LLM agents on track. Breaking down problems into specific tasks for agents leads to fewer opportunities for LLMs to go astray. This is not only a matter of giving each agent a specific goal, but also giving each agent only the set of tools it should need. For example, the POV producer tool does not have the ability to read source code, but it does have the ability to ask questions about source code through the "Source analyzer" agent. This intentionally directs the agent to make "high level" decisions and keeps the context window for the agent cleaner.

The tools given to LLM agents are also carefully written to be robust to mis-usage and to produce errors as helpful as possible, with the hopes that agents can use this and adjust to errors they encounter. For example, when Python code for POVs is evaluated, it is expected to produce a single file named "pov.bin". If the file is not found, an error is returned specifically mentioning that "pov.bin" was not produced and should have been. However, if a single file with an alternate name is produced, the system assumes it was the "intended" output, and uses it regardless of the file name.

Because LLM Agents run in a loop, they may run for a while without making progress. There are a few strategies to handle this: specific pattern matching for exit conditions (such as "POV successfully triggered a sanitizer"), and hard limits on the number of iterations of conversation that an agent can have. If an agent is stopped due to hard limits, it will be prompted to report what it has done so far and errors it encountered to return to its caller, which may choose to invoke it again. In some cases such as POV production, when an agent fails, a new one takes its place in case the issue was transitory. Our CRS contains a few interventions to help when agents meet certain heuristics for getting "stuck". Largely these follow a similar pattern: detect that an action has repeatedly failed, and if so, inject a text prompt in the agent conversation to attempt to bring the LLM back on track. For example when attempting to generate patches, if patches repeatedly fail to compile, the LLM is recommended to double check the original source, break the patch into smaller pieces, re-evaluate its entire approach, or even attempt to make a less useful patch that is simpler to implement.

We believe our CRS would benefit greatly from the ability to fine-tune models, and/or from the ability to access open-weight models (such as LLaMa). A large amount of development time and effort was spent in having LLMs produce valid patches for code. Our approach uses fuzzy matching to handle the numerous mistakes that LLMs produce, but this approach still has many issues, and we believe that fine-tuning would greatly improve the success rate. Open-weight models could also help as they permit deeper introspection for more analytical prompt engineering.

A.4 LLM Feedback

We plan to continue to use LLMs for the final event, and increase our usage of LLM agents. We intend to further research how to limit both false positives and false negatives in vulnerability detection through LLM interactions. Additionally, we expect to make use of text embeddings in the future for RAG based tools for our CRS agents. While we believe fine-tuning of models would improve their performance, we do not have concrete ideas for how to train and incorporate entirely custom models into our CRS.

A.4 Strategic Reflections

While we have changes we wish to implement for the final event, given the same timeframe (May to July) we do not feel there are substantial changes we would make to our CRS.

Section B. Technical Details

B.1 CRS Codebase

The CRS is an agent based architecture with several distinct agents and tools created to perform the tasks that security professionals take when performing audits. The code is written entirely in Python and is approximately 8k lines of code, including comments and prompting strings.

All of the tools and agents range from incredibly simple to quite complex. This architecture blends the flexibility of LLMs being able to select from a number of strategies and choices for how to generate POVs and GPs, with the rigidity of following specific workflows that minimize the risk of the LLMs getting “stuck” or “confused” about what to work on.

Agents

The agents are generally “conversation style” agents with tool use. That is: they retain their interactions through several iterations of LLM queries, with the LLMs calling tools at each step to progress towards their goal. These agents lose their “memory” and have a fresh context window when they are initialized. This keeps the context windows small, as the agents get work that is hopefully possible to be accomplished with few enough interactions and tool calls that it fits inside a single context window. The callers of the agents only get the significant output results in the form of relatively short answers to queries, keeping the caller's context relatively clear.

Beyond the hard limits on context size, keeping the agent context as small as possible also reduces query cost and latency. To this end, our agent logic contains a context compression strategy. Once the context size exceeds some threshold, our CRS will query an LLM to compress the current transcript while preserving all critical information relevant to the task.

By default, the agents use Claude 3.5 Sonnet. Most of the development of these agents was done with ChatGPT 4o, and the system can work with other models swapped in. However, testing showed Claude 3.5 Sonnet was more capable of accurately finding and generating inputs for bugs. As it is still incredibly powerful, but also faster and with greater token rate limits in the competition, ChatGPT 4o is still used for tasks which require more throughput.

Commit analyzer: The commit analyzer is responsible for reading a single commit and deciding whether the code changes it contains may represent a security risk. As commits by themselves contain little context, the agent is enriched with the deterministic "Definition finding/reading" and the "Source reading" tools. These empower the agent to search through the code as it sees fit to determine the safety of the commit. If it believes it finds a plausible bug, it is responsible for returning information about that bug, including not just what the bug is, but why it believes this represents a vulnerability in-scope for the contest and reachable by the provided harnesses.

Debug print adder: This agent is responsible for adding print statements to the given source code, solely for the purpose of determining what is happening inside the program. This mimics how many programmers use "debug prints", which are inelegant, but work across almost any target. The agent is given a description of the information that should be printed, and it performs the edits asked, and returns a description of them. Some language-specific instructions were given to ensure the LLM generates debug prints that will pass log level filtering.

Dynamic debugger: Our CRS does not have the ability to use debuggers such as gdb. This unfortunately is a severe limitation when attempting to understand why a PoV does not work, or other similar information that may require dynamic analysis. We instead rely on print debugging by calling "Debug print adder", running the actual input, and extracting the relevant details from the output of the program.

Patcher: The patcher is responsible for fixing found vulnerabilities. It is given the original vulnerability description (generated by the "POV producer" or basic info from a fuzzed finding), commit information, and the input itself that caused the vulnerability. It has the ability to read source code, search for definitions, perform modifications to source code, and run the provided tests for the challenge. When it has a candidate patch, it is tested against the functionality tests and the known POVs that trigger a sanitizer on that commit.

Buffer generating function builder: This agent is responsible for building a “buffer generating function”. This function is a small piece of Python code that generates a suitable byte-stream for the provided harnesses, but potentially takes as input a higher-level set of parameters. Things like encoded lengths or checksums, or fields which are concatenated with special bytes between them are ideally handled by the buffer generating function. Once the Python code is produced, this agent is further instructed to document the Python code with any information about the harness’s behavior that isn’t obvious from the code alone. This assists the POV producer in focusing on the vulnerability rather than the details of the provided harnesses.

POV producer: The POV producer directly attempts to produce a POV given a description of the potential vulnerability (generated from the “Commit analyzer”). It is also given “reachability” and “ease of POV development” scores which are used for prioritization in the face of several POV production jobs. It produces Python code rather than a byte stream directly, which makes it easier to generate complex outputs. Generally the POV produced uses the buffer generating function from the previous agent, and adds in extra steps to find good parameters to that function. Python code for POV production is run in a minimal NSJail sandbox to safely evaluate the output.

Reachability: This is similar to the “Dynamic debugger” tool, but augmented with extra information about caller/callee relations, and includes some caching to more quickly answer results. This agent takes in an input and a target function, and attempts to determine the “closest” function to the target that was hit. This may use the “Debug print adder” if deterministic reachability through instrumentation is not available.

Seed producer: This agent is similar to the POV producer in that its job is to generate valid inputs for the target program. However, the outputs from the seed producer are intended not to crash the program, but only to reach as close as possible in the program to commits. This data is then fed to fuzzers, hopefully bootstrapping their search efforts. Unlike the POV producer, this is given minimal tools as its failure is not detrimental.

Source analyzer: This agent is responsible for answering static questions about the source repository. This allows this agent to potentially spend its own context reading many files and poking around source code to answer questions, rather than having the querying agent take the context window penalty from somewhat open-ended exploration.

AI orchestrator: This agent is designed to consider the aspects of POV development. Generally the workflow expected from this tool is that it will read commits, decide which are vulnerable, and queue them to have POVs produced. When they are queued, they are also given an estimated score for “reachability” and “ease of POV development”.

Tools

The tools for our CRS are “deterministic” code, distinguishing them from the agents which heavily use LLMs. Any task that can be performed in a generic and robust way without requiring an LLM is done with a tool The tools are designed to be as robust as possible to bad arguments, and to give errors as detailed and helpful as possible, as LLMs occasionally make mistakes when calling tools, but can adjust to helpful error messages.

Fuzzing: The CRS attempts to build the provided harnesses as fuzzers and quickly sanity check if that worked. If so, fuzzers are launched against each harness provided. Seeds are added to the corpus of each fuzzer from the “Seed producer” agents, as well as from any POV tests which the CRS decides to run.

Jazzer and libFuzzer were usable with minimal changes to the CP build process. However, fuzzing the Linux kernel challenge required writing a custom harness wrapper that used AFL as the engine with KCOV for coverage feedback. This approach was based on the article published on the Google Security Blog.

Deterministic Reachability: Using the instrumentation provided by the fuzzers, this tool can answer queries about which functions are reached when a harness is run with a given input. This tool was made available to the POV Producer and Dynamic Debugger to aid in developing and debugging POV candidates. In particular, using coverage instrumentation for reachability queries is significantly faster than debug-print based approaches, so using this tool was prioritized when possible.

Source reading: A simple tool which reads a given line of a source file, along with some context lines before and after it. This is an attempt to balance “reading enough code to contextualize it” with “too much code can overwhelm the model”.

Fuzzy patch reader: A significant amount of development effort was put towards creating a patch reader robust to many of the common errors LLMs make when generating unified diff patches. We use dynamic programming sequence alignment techniques to figure out the area where the LLM was trying to modify, allowing for fuzzy matching for minor errors. We observed several common errors, with the most difficult one being LLMs implicitly changing lines in patch files (that is: hallucinating changes in the lines surrounding the actual patch they submitted). For some specific cases (such as hallucinating changes in lines which cause specific semantic changes) this tool has special error handling to encourage a new patch to be attempted.
This tool is used not only for generating patches to submit as GPs, but also for internal patching for debugging purposes.

Definition finding/reading: Rather than using full static analysis, our CRS only has the ability to search for symbol references and definitions. This is done using the relatively common off-the-shelf tools ctags and gtags. These support a variety of languages, ensuring the CRS can be generalized, and they require no specific compilation instructions. Their analysis is based on string parsing which can have issues with non-trivial code, but in practice they perform well.
This tool outputs not only the location(s) where a definition or reference was found, but also the content of where it was found. This enables the CRS to potentially skip an extra tool call to read the source line where the definition occurred.

Compilation: Compilation calls the provided run.sh commands to build the code. Rather than build with a patch file passed as an argument, our code always applies the patch to the source repository and then builds the entire repository. A large issue is what to do when the build process fails: errors may be printed that are needed to fix attempted patches, but the entire output from the build may be thousands of lines long. Our CRS uses a cheap LLM model to attempt to extract the relevant lines from the output and return the errors to the caller in the case of a compilation failure.

Bisector: The bisector takes potential vulnerabilities found by the CRS and does a git bisect equivalent to determine which commit was responsible, and sanity checks that the necessary conditions for submission are met. To make this faster, builds are performed and cached for each of the commits. For vulnerabilities discovered by the LLM agents, the CRS reports a commit for the vulnerability. If that is correct, the more expensive bisect process is bypassed. After confirming a POV meets the eligibility criteria, a separate thread in the bisector also tests the POV on each commit (whereas the normal bisect did only a binary search). This is an attempt to find regressions which may lead to a POV being “valid” against multiple commits. The bisector writes any valid POVs to a database to be submitted.

Problem information and interaction: Basic information about the problem such as commit information, sanitizer, problem name, problem language, and so on comes from this tool. For Java programs, the descriptions are also enriched with information about how the specific sanitizers operate, to ensure that an LLM can understand how to trigger the sanitizers when they look for special strings. Other information may also be injected based on things such as the binary architecture, whether or not the problem is the Linux Kernel, and so on.

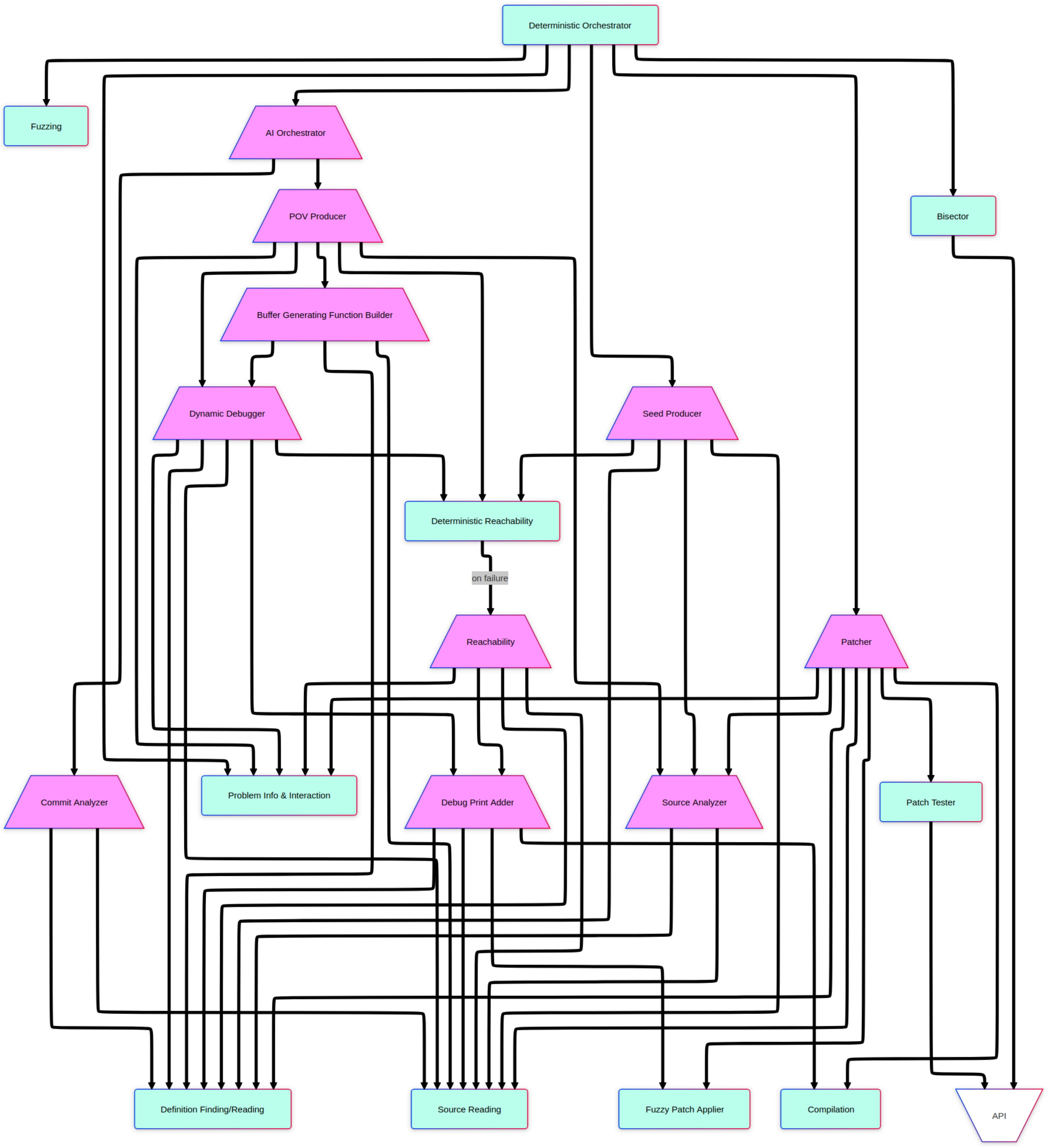
Deterministic orchestrator: This is the overall controller for the CRS. It launches the AI orchestrator and fuzzers at startup. If the number of commits is more than 24, different orchestrators are launched in parallel on the commits to improve throughput. Once suspected vulnerabilities have been queued for examination, they are scheduled for running. If a POV is found, after a fixed delay (15 more minutes, to allow fuzzers to find similar crashes), patching jobs are also queued for production.
Due to engineering constraints, we primarily run one LLM agent at a time, minimizing contention on possible shared resources. If there were 10 plausible vulnerabilities scheduled to be analyzed, we need to schedule them to ensure each is examined, as attempting to produce a POV for a non-existent bug can run indefinitely. The scheduling uses a priority score for each schedulable job. These scores are initially set when the job is created, and decay over time based on hard-coded heuristics such as failed tool calls.

B.2 Architecture Integration

Our CRS did not make strong use of the Kubernetes environment, due to engineering constraints. Our deployment put the Docker pod on its own node in an attempt to improve performance of Docker commands (including fuzzing), our CRS on its own node to improve its resources, and all remaining components on the final node.

B.3 Architecture Diagram

As our CRS is only a single Kubernetes node, the best way to show its design is by diagramming the relationship between the agents and tools used. In the following diagram, light teal boxes represent deterministic tools, the pink trapezoids represent AI agents, and the white trapezoid represents the AIxCC API.



Section C. Additional Feedback

C.1 Resource Constraints

Our CRS design was greatly impacted by the LLM API limits. The rate limits caused us to limit most of our LLM usage to a single dispatcher which would schedule individual agents to run in serial, and to make other concessions to prevent too much parallelism.

For our CRS, increased rate limits would be very helpful. We believe increased rate limits would more accurately reflect both the usage of LLMs in practice today (our Anthropic account was upgraded to a 1M token/minute rate limit for all models at no cost and with no questions, compared to limits as low as 40K token/minute limits in the competition), as well as better reflects where LLM usage will be in the near future as costs continue to decrease.

The spend budget of $100 seemed the appropriate order of magnitude, but with more spending available, we believe some additional LLM agent based strategies such as multiple agents debating over findings or several paths explored in parallel would open up. Given that LLM API costs will continue to decrease after the competition ends, we believe the AIxCC results would stand up better over time if the budget was increased to around $500 per problem.

Our CRS also was impacted by other competition restrictions from the Kubernetes environment. Originally our CRS was designed to “fork” CPs easily using OverlayFS. This allowed the system to prevent accidental code modifications by creating a clean, modifiable instance for workers. Switching to a network file system which does not permit OverlayFS presented huge performance losses over this approach: instantiating new copies of a CP went from milliseconds to minutes, disk usage went up dramatically, and build times increased as there was less efficient reuse of existing build artifacts.

We did not find the time constraints of 4 hours nor the resource constraints of the machines themselves to be limiting.

C.2 Real-World Use of CRS

Theori is very interested in real world usage of the CRS we are producing. Our primary existing commercial interest in this area is to produce agents to assist with web application security, potentially without access to source code. While this scenario is quite different from the AIxCC environment in which we have full code access, we believe many of the lessons and designs around LLM based agents will carry over. Future commercialization strategies may more closely follow AIxCC’s pattern of working on repositories and integrating with CI/CD systems.

Some necessary concessions made in AIxCC, such as the existence of a standardized interface to use for evaluation of patches and POVs, may not adequately reflect how commercial customers would present code, and represent additional research and engineering challenges.

We would be amenable to discussing these topics further.

C.3 Final Competition

Our primary recommendation would be to increase the number of distinct codebases on which the CRSs are run. We expect that many common failure scenarios for a CRS would apply to an entire codebase due to, for example: language issues, unique directory structure, instrumentation challenges, and so on. This means a larger number of codebases are needed to better evaluate a CRS's performance. Additionally, with many challenges released beforehand for the ASC, teams are more able to add special case handling for those codebases, which goes against the spirit of an automated system which would work on new projects.

Increased transparency would also help not just competitors to trust the competition is run fairly, but also to increase the likelihood that competitor systems align with the goals of AIxCC. Decisions like the extension of the submission deadline without explanation may be done for valid reasons, but without transparency on how the decision was made, are indistinguishable to competitors from extending the deadline to accommodate a favored team, which would present huge fairness issues. These issues also impact successful teams: good performance is diminished by potential appearance of fairness issues. Additionally, decisions to not release logs, playbooks, or any verifiable information about how the ASC was run are indistinguishable to competitors from several challenges being run, and only those with results favoring certain teams being chosen. While not perfect, the Cyber Grand Challenge used cryptographic precommitments to ensure the organizers could not adjust competition parameters to favor teams while not revealing the details until after the competition was run.

Transparency can also help improve the performance of all teams: if decisions are made with certain intentions (for example if limiting API spend is done to reflect constraints for certain real-world scenarios, as opposed to concerns of cost for running the event), that information can help teams better understand the goals of the contest and build their systems accordingly. Information about how many harnesses and commits are in each CP (just a binary order-of-magnitude range like 8-16 harnesses and 64-128 commits) would be useful for competitors to properly account for how the competition will be run. There is no good reason to hide this information, and providing it can help competitors make systems that more closely meet the goals of the competition.