

The Cognition Gap: What the Cyber Grand Challenge Revealed About Autonomous Cyber Reasoning

Jason L. Wright, Idaho State University

Accepted September 30, 2026

Abstract

The 2016 DARPA Cyber Grand Challenge fielded autonomous systems for vulnerability discovery, exploitation, and patching of stripped binaries. CGC solved orchestration but exposed a cognition gap along two axes: discovery and exploitation. AIXCC demonstrated that LLMs approach human discovery on source code; exploitation and binary-only reasoning remain open.

Keywords: autonomous cyber reasoning, binary analysis, vulnerability discovery, exploit generation, large language models, Cyber Grand Challenge

1 Introduction: A Decade Later

In 2016, seven teams competed in the final event of the DARPA Cyber Grand Challenge (CGC),¹ fielding fully autonomous Cyber Reasoning Systems capable of discovering, exploiting, and patching software vulnerabilities in real time. The competition grew out of a question familiar to the capture-the-flag community: could machines play the game that human teams had been playing at DEF CON since 1996? Twenty years later, at DEF CON 2016, they got a partial answer: ForAllSecure’s Mayhem competed alongside human teams in the CTF finals, playing on the same DECREE platform built for CGC.

© 2026 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

This is the author’s accepted manuscript. Cite as: J. L. Wright, “The Cognition Gap: What the Cyber Grand Challenge Revealed About Autonomous Cyber Reasoning,” *IEEE Security & Privacy*, accepted for publication, doi: 10.1109/MSEC.2026.3740004. Published version: <https://doi.org/10.1109/MSEC.2026.3740004>

¹Competition rules: https://archive.ll.mit.edu/cybergrandchallenge/docs/CGC_Rules_18_Nov_14_Version_3.pdf

In the years that followed, CGC’s model was replicated internationally. China launched at least a dozen government-backed Robot Hacking Games beginning in 2017 [2]. DARPA itself returned to the format with the AI Cyber Challenge (AIXCC) in 2023 [15]. The orchestration architecture that CGC demonstrated, coordinating vulnerability discovery, exploitation, and patching into a continuous autonomous loop, proved durable and reproducible.

But the systems built on that architecture remained limited. The reasoning techniques available in 2016 (fuzzing, symbolic execution, constraint solving) could explore program behavior mechanically but could not understand it. Removing the human analyst that every tool assumed, as CGC required, exposed a gap that no amount of engineering could close: the systems could be orchestrated, but they could not think.

A decade later, large language models (LLMs) have begun to fill that gap—but only partially. LLM-based systems can reason about code semantically, generalize across unfamiliar software, and engage with the open-ended analytical questions that previously required human expertise. But their strength is greatest when operating on source code. CGC was deliberately designed around binary-only analysis [5], reflecting DARPA’s concern with the cost of analyzing the commercial off-the-shelf software the Department of Defense relies on. That harder problem, reasoning about program behavior from a stripped binary alone, remains largely open.

This article traces that arc. We argue that CGC solved the *orchestration problem* of autonomous cybersecurity but exposed a deeper *cognition problem*, and that this decomposition, visible only in retrospect, explains both why CGC’s immediate impact appeared limited and why its hardest challenge is still unmet.

2 What the Cyber Grand Challenge Actually Built

CGC is often remembered as a hacking competition, but that framing understates what it required. To field a system in the final event, teams had to build, and DARPA had to host, something closer to an autonomous cyber operations platform.

2.1 An End-to-End Autonomous Loop

Each competing Cyber Reasoning System (CRS) had to perform four tasks in a continuous loop, with no human intervention:

- **Vulnerability discovery.** Finding exploitable flaws in challenge binaries.
- **Exploit generation.** Producing working proofs of vulnerability (inputs that demonstrated control over the target’s execution).

- **Patch synthesis.** Generating replacement binaries that fixed discovered vulnerabilities while preserving correct program functionality; patches that broke legitimate behavior were penalized.
- **Continuous adversarial interaction.** Operating all three tasks simultaneously across multiple challenge binaries, under scoring pressure, while opponents attempted the same.

No CRS performed all of these tasks well, but every finalist performed all of them—several well enough, particularly at discovery and patching, to demonstrate that end-to-end autonomy was achievable.

2.2 Infrastructure at Scale

The competition infrastructure was itself a significant engineering achievement, providing:

- **Deterministic execution.** Challenge binaries ran on DECREE (DARPA Experimental Cyber Research Evaluation Environment), a custom operating system interface built specifically for CGC [13]. Where a conventional OS exposes hundreds of system calls, DECREE provided exactly seven (for I/O, memory management, synchronization, and process termination), with no signals, no shared memory, no file system, and no environment variables. Hardware instructions that could leak host state were trapped and replaced with deterministic values, and the sole entropy source was a kernel-controlled seeded PRNG. Identical inputs *guaranteed* identical outputs, making automated analysis tractable and scoring fully verifiable. DECREE ran on 32-bit x86, the architecture for which analysis tooling (fuzzers, symbolic executors, binary lifters) was most mature in the early 2010s. Critically, CGC was binary-only by design [5]: competitors received stripped executables with no source code, no symbols, and no debug information.
- **Isolation and fairness.** Each CRS operated in an isolated environment with identical resources; strict separation ensured that no team could interfere with the execution of another or observe its strategy. To further ensure integrity, DECREE was implemented independently on two operating systems—a Linux shim for qualifying and a FreeBSD-based hypervisor for the final—designed to be indistinguishable to user programs.²

2.3 Scoring and Proof of Vulnerability

To score points, a CRS had to produce a working *proof of vulnerability* (PoV): not merely a crash, but a structured demonstration of what the crash meant. Two exploit types were accepted.

²J. L. Wright, C. Eagle, and T. Vidas, “Operating System Doppelgängers,” *BSD Magazine*, Feb. 2018; https://archive.org/details/BSD_Magazine_102.

A Type 1 PoV demonstrated execution control. The CRS declared which general-purpose register it could control; the scoring infrastructure then issued a challenge: produce an input that crashes the binary with EIP and that register holding specified values at crash time. This was a stronger proof than an uncontrolled crash; it demonstrated that the CRS understood the program well enough to redirect execution to an attacker-chosen address.

A Type 2 PoV demonstrated an information leak. Every DECREE process mapped a *magic page* at a fixed address, pre-filled with per-instance pseudorandom data. To score, the exploit had to leak 4 consecutive bytes from that page in a single interaction; the scoring infrastructure verified the bytes automatically.

Both verification procedures required no human judgment: the scoring system either confirmed the result or it did not. This complete automation was what made the competition among seven autonomous systems operationally feasible in 96 rounds [9].

2.4 The Challenge Corpus

The 82 challenge sets fielded in the final event [9] spanned 29 distinct CWE vulnerability categories. The corpus was dominated by memory corruption: stack-based buffer overflows (CWE-121) alone accounted for roughly a fifth of the challenges, and heap overflows, out-of-bounds reads and writes, and buffer underflows collectively made up over half. But the corpus also included integer overflow and type confusion vulnerabilities, uncontrolled format strings, use-after-free conditions, improper null termination, information disclosure, and error-handling flaws. Challenge binaries were authored by three independent performer groups (Cromulence, Kaprica Security, and Narf Industries), each contributing distinct software designs and vulnerability styles.³ A CRS could not rely on a single analysis strategy; the corpus demanded breadth across vulnerability classes, not just depth in one.

2.5 An Existence Proof

Before CGC, autonomous vulnerability discovery, exploit generation, and patch synthesis existed as separate research problems; no one had composed them into a single system that operated end-to-end, under adversarial conditions, in real time.

CGC provided an existence proof that they could be so composed. The systems were imperfect (they missed vulnerabilities, generated fragile patches, and sometimes degraded the binaries they were trying to protect) but they worked. The loop closed.

³The challenge binary corpus is archived at <https://github.com/CyberGrandChallenge/>.

3 The State of Program Analysis in 2016

To understand why CGC-era systems hit a ceiling, it helps to survey what program analysis could and could not do at the time, independent of what the CGC competitors themselves built.

3.1 Finding Bugs

The dominant technique for automated bug finding was coverage-guided fuzzing, led by AFL (American Fuzzy Lop), released by Michal Zalewski in 2013.⁴ AFL made it possible to find memory corruption bugs in real software with minimal setup, but it was blind to program semantics: mutating inputs and observing coverage scaled to shallow bugs but struggled with deeply guarded paths, checksum validations, and stateful protocols. Structured fuzzers like Peach could target specific protocols, but only after a human wrote the input grammar.

On the formal side, symbolic and concolic execution tools—KLEE [1], S2E [4], and Microsoft Research’s SAGE [7]—could reason about path constraints and generate inputs that reached specific program states. But all of these tools shared a fundamental scaling problem: path explosion. Feasible paths through a nontrivial program grow exponentially, and no amount of heuristic pruning made symbolic execution practical for large, real-world binaries without significant human guidance.

3.2 Understanding Programs

Reverse engineering tools were powerful but fundamentally manual. IDA Pro with the Hex-Rays decompiler dominated; Radare2 provided an open-source alternative; Binary Ninja, launched in July 2016 by Jordan Wiens and Rusty Wagner (both CGC veterans), brought a more modern architecture. All of these were interactive workbenches: they presented information to a human analyst, who supplied the reasoning. No tool could explain what a function did, recognize that a series of operations implemented a known cryptographic algorithm, or identify a use-after-free in a custom allocator’s coalescing logic. Those judgments required a human who understood both the mechanics of the code and the intent behind it.

3.3 Building Exploits

Exploit development in 2016 was a craft. Some subtasks had been partially automated: gadget enumeration tools could find return-oriented sequences, and constraint solvers could sometimes chain them. But the harder parts—understanding heap layout, predicting allocator behavior, identifying useful primitives in application-specific data structures—remained specialist knowledge, passed between practitioners through CTF write-ups and blog posts rather than through tools.

⁴AFL has no formal publication; see <https://lcamtuf.coredump.cx/afl/>.

Defensive mitigations (ASLR, DEP/NX, stack canaries, early forms of control-flow integrity) were standard on major platforms by this point. Bypassing them required per-target manual effort, and their interaction made automated exploit generation an open research problem rather than an engineering task.

3.4 The Gap

The pattern across all of these areas was the same. Automation existed for narrow, well-defined subtasks: mutate inputs and measure coverage; lift a binary to an intermediate representation; enumerate ROP gadgets. But the reasoning that connected these subtasks, deciding *what* to analyze, understanding *why* a crash was exploitable, judging *how* to construct a working exploit, was entirely human. Every tool in the ecosystem assumed a skilled analyst in the loop. The question CGC posed was what happened when you removed that analyst entirely.

4 The Orchestration/Cognition Split

In retrospect, the competing teams solved two fundamentally different kinds of problems when building their Cyber Reasoning Systems, and they solved them to very different degrees.

4.1 Orchestration: The Problem That Yielded

The first problem was orchestration: how to coordinate vulnerability discovery, exploitation, and patching into a continuous autonomous loop. Each CRS had to decide, in real time and without human input, how to allocate its computational resources across multiple challenge binaries. Should it keep fuzzing a binary where it had found one crash, or move on? Should it prioritize attacking opponents or defending itself?

The competing teams solved this problem convincingly. Their systems scheduled analysis tasks, managed parallel workloads, and made strategic decisions under time pressure. The orchestration layer, the part that decided what to do and when, worked.

The competition infrastructure solved a parallel orchestration problem: coordinating challenge distribution, binary replacement, network traffic routing, exploit verification, and scoring across all seven competitors simultaneously. Scoring was fully automated, and final rankings were verified using both the live production system and an independently written offline system to ensure correctness.

Both of these orchestration challenges, competitor-side and infrastructure-side, were hard. But they were *engineering* problems, amenable to careful design, testing, and iteration.

4.2 Cognition: The Problem That Didn't

The second problem was cognition: the ability to form semantic models of program behavior—understanding *what* code does and *why*, not merely *where* execution goes. Where orchestration is a scheduling problem (which tool to run, on which binary, when), cognition is an interpretation problem: recognizing that a buffer is undersized for the data it will receive, that a particular code pattern implements a vulnerable protocol parser, or that a crash implies a controllable write primitive. This is the reasoning a human analyst supplies, and it is what no tool in 2016 could provide. The competing teams hit this limit hard.

The CRS systems combined the best available techniques (fuzzing, symbolic execution, constraint solving, heuristic-driven analysis) into sophisticated pipelines. But these techniques could explore program paths without understanding programs. A fuzzer could discover that a particular input caused a crash, but could not explain why. A symbolic executor could enumerate feasible paths through a function, but could not recognize that the function implemented a vulnerable parser.

The result was systems that were brittle in characteristic ways: strong on vulnerabilities that lay along easily reachable paths with straightforward triggering conditions, weak on complex control flow, deep state dependencies, and vulnerabilities that required semantic understanding of what the program was trying to do. The numbers bear this out: of the 82 challenge sets fielded in the final event, competitors produced working proofs of vulnerability for only 20, less than a quarter of the available targets [9].

The pattern of what was and was not exploited is telling. Our own analysis of the CFE archive,⁵ cross-referencing per-round exploitation outcomes against each challenge binary's CWE classification, reveals a clear skew: the 20 exploited binaries were dominated by classic memory corruption (stack and heap buffer overflows, integer overflows, off-by-one errors), vulnerability classes that coverage-guided fuzzing can reach mechanically. Meanwhile, the corpus included use-after-free conditions, type confusion, and format string vulnerabilities, none of which were successfully exploited. These are vulnerability classes that require understanding program semantics: allocator state and object lifetimes for use-after-free, type hierarchies for type confusion. A fuzzer that discovers a crash has no model of any of these concepts.

The timing was equally revealing. Two-thirds of the exploited binaries (13 of 20) were cracked within five rounds of their deployment; the longest-lived exploited binary held out for thirteen rounds before falling. The 62 binaries that were never exploited remained unexploited across enablement windows that ran fifteen to twenty-five rounds. The difficulty curve was not gradual but bimodal: either a CRS's mechanistic techniques could reach the vulnerability quickly, or no amount of additional computation would find it. That bimodality is the signature of a cognition gap rather than a difficulty gradient: more compute, more time, and better-tuned heuristics never steadily extended a system's reach into

⁵Scripts and per-CB results at <https://github.com/wrigjl/cgc-cfe-reanalysis>, archived at <https://doi.org/10.5281/zenodo.22728886>.

the unsolved binaries. Those targets required a kind of reasoning the available techniques could not produce at any scale.

Cognition is not a single capability: CGC required at least two distinct kinds of reasoning. *Discovery cognition* is the ability to recognize a vulnerability class from program behavior: identifying that an allocator permits a use-after-free, that a parser’s length handling is inconsistent with its declared protocol, or that an integer truncation three calls deep produces a controllable write. *Exploitation cognition* is the ability to turn such a flaw into a working primitive: reasoning about heap layout, predicting allocator state, choosing useful gadgets, and chaining them under live mitigations. CGC’s scoring forced systems to demonstrate both. A Type 1 PoV demanded controlled execution: a specific EIP and register value, not merely a crash. A Type 2 PoV demanded leaking four chosen bytes, not an uncontrolled read. The unexploited 62 binaries were hard on both axes simultaneously: harder to recognize as vulnerable, and harder to exploit once a crash was found. The distinction matters because the two axes are orthogonal. Progress on one does not imply progress on the other, and the security consequences of advances on each are not symmetric. Independent empirical work corroborates this asymmetry. ExploitGym [12], a 2026 benchmark of 898 real-world vulnerabilities spanning userspace, V8, and the Linux kernel, frames exploitation as the under-evaluated cognition axis: even with mitigations disabled, the two strongest LLM-agent configurations produce working exploits for only 157 and 120 of the 898 instances, while every other configuration solves fewer than 15. Exploitation cognition lags discovery even when source is available—the same asymmetry the unexploited CGC binaries showed a decade earlier. Figure 1 illustrates this decomposition.

4.3 Engineering Around the Gap

Both the competitors and the program organizers recognized this limitation and engineered around it.

On the infrastructure side, DECREE’s narrow surface—no nondeterminism, seven syscalls, no environmental variability—reduced the problem space to something the available analysis techniques could handle within the competition’s multi-year cycle.

On the competitor side, teams built heavy scaffolding around their analysis engines: crash triage heuristics, patch templates, exploit construction pipelines. These were sophisticated engineering, but they were compensating for a reasoning capability that did not exist. Each new class of vulnerability or program structure required new heuristics, new templates, new special cases. The variation across teams was stark: the most prolific attacker exploited 15 of 82 original binaries, while one finalist exploited none at all.

The patching side showed the same limitation. Across the competition, teams deployed 442 replacement binaries. Most survived, but primarily because the underlying vulnerability was never found by any attacker. When an attacker *did* hold a working exploit, the patch frequently failed to stop it: of the replacement binaries that were exploited, 72% fell in the very next round. In several

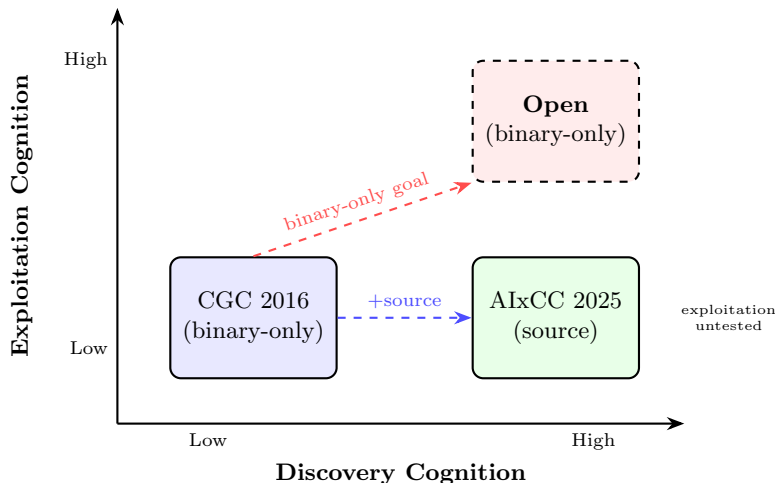


Figure 1: The cognition gap as a two-axis problem. Placements are qualitative, not measured. CGC 2016 systems operated at low capability on both axes. AIxCC 2025 closed the gap along the discovery axis for source-available code, but its scoring did not test exploitation cognition. The binary-only region that motivated CGC’s design remains open along both axes.

cases, teams deployed multiple successive patches for the same binary, each one exploited immediately. The orchestration loop was working (detect that the patch was scored against, generate a new one) but without understanding why the previous patch failed, each replacement was no better than the last.

The orchestration worked. The cognition was a workaround. And the fundamental gap, that no system could reason about what a program *meant*, remained open.

5 After CGC: Replication Without Revolution

If CGC had solved the right problems, one might have expected autonomous cyber reasoning systems to proliferate in the years that followed. In a sense, they did. But only on the orchestration side.

5.1 International Replication

The most direct evidence that CGC’s orchestration model worked came from China. Beginning in 2017, the Chinese government launched a series of Robot Hacking Games (RHG) explicitly modeled on CGC’s format. According to a 2021 Georgetown CSET report [2], China hosted at least a dozen RHG events between 2017 and 2021, with participants spanning universities, major technology companies (Baidu, Tencent, Qihoo 360), and PLA military organizations.

The RHG competitions demonstrated that CGC’s orchestration framework was reproducible: different teams, in a different country, under different institutional structures, stood up the same kind of autonomous attack-and-defend competition. What did not transfer was a solution to the cognition problem. The RHG competitors used the same classes of techniques and hit the same ceilings. The orchestration scaled; the reasoning did not.

5.2 Incremental Progress in Analysis Tools

Outside of competition contexts, the individual components of program analysis continued to advance. Coverage-guided fuzzing matured significantly: AFL spawned a family of successors including libFuzzer, honggfuzz, and structure-aware fuzzers, and Google’s OSS-Fuzz, launched in 2016, brought continuous fuzzing to hundreds of open-source projects and found thousands of bugs. NSA released Ghidra in 2019, giving the reverse engineering community a capable free alternative to IDA Pro.

The CGC competitors themselves seeded much of this progress. ForAllSecure’s Mayhem [3], the competition winner, became a commercial binary analysis product. Shellphish’s entry produced angr [10], an open-source binary analysis framework that became the dominant platform for academic work on automated exploitation. Driller [11] combined angr’s concolic execution with AFL-style fuzzing, using symbolic reasoning to push past coverage plateaus that defeated pure fuzzers. QSYM [14] refined this further, making concolic execution fast enough for practical use. Each advance made a specific component more powerful, but none addressed the integration problem. A fuzzer that could reach deeper code still could not judge whether the crash it found was exploitable, and a concolic executor that could solve path constraints still could not decide which paths were worth exploring. The ceiling was not in the tools but in the absence of a reasoning layer that could use them the way a human would.

5.3 AIxCC: The Successor Competition

In 2023, DARPA launched the AI Cyber Challenge (AIxCC) [15], a competition that was, in many ways, a direct sequel to CGC. The structure was familiar: teams built autonomous Cyber Reasoning Systems to find and patch vulnerabilities, competing at DEF CON in 2024 (semifinals) and 2025 (finals). Forty-two teams entered; seven reached the final. The results were impressive: by the finals, competing systems identified 86% of injected vulnerabilities and correctly patched 68% [15]. All seven finalist systems were released as open source.

But AIxCC changed the problem in a fundamental way (Table 1). Where CGC competitors operated on stripped binaries with no source code, AIxCC gave its competitors access to real open-source C and Java projects: source code, build systems, commit history, and fuzzing harnesses. The competing systems used LLMs to reason over source code directly, with tools like CodeQL and tree-sitter providing structural context. Six of the seven finalists used LLMs to

drive proof-of-vulnerability generation directly, and Theori’s RoboDuck pushed the agentic-LLM approach furthest, building its entire workflow—bug candidate identification, PoV generation, and patching—around autonomous LLM operation. The term “proof of vulnerability” carries different weight between the two competitions. In CGC, a PoV was an exploitation primitive: Type 1 demonstrated attacker-chosen execution and Type 2 demonstrated information disclosure. AIxCC’s PoV served a different purpose: inputs that trigger a sanitizer (ASan, Jazzer) when run [15], proving the existence of a reproducible memory corruption bug, but importantly not demonstrating that the corruption is exploitable (crash, but not necessarily control). This represents a significantly lower bar than that required by CGC. The 86% headline figure measures discovery cognition; AIxCC’s scoring never tested whether competing systems could turn those crashes into working exploits. The exploitation axis remains unmeasured by competition, with ExploitGym’s results (Section 4.2) providing the only direct benchmark.

The architectural continuity with CGC is striking. A systematic analysis of all seven finalist systems [15] found the same orchestration pattern CGC proved in 2016: a coordination layer managing parallel fuzzing and LLM-analysis pipelines, with results cross-feeding between them.

What changed was the reasoning engine *and* the representation it operated on. The improvement over CGC-era results reflects two shifts, not one: LLMs provided a qualitatively more capable reasoning engine, but they also received a qualitatively richer input: source code rather than stripped binaries. Disentangling these two factors is important, because it determines whether the cognition gap CGC exposed has been closed or merely sidestepped—and the answer may differ for the discovery and exploitation axes.

The decade between CGC and AIxCC followed a consistent pattern: the orchestration model spread and matured, the analysis tools improved incrementally, and the cognition ceiling held across all of them. When LLMs began to lower that ceiling, they did so unevenly: powerfully for source-level analysis, tentatively for the binary-only problem that CGC was designed to test.

Table 1: CGC (2016) vs. AIxCC (2024–2025). The competition structure and orchestration architecture carried forward; what changed was the reasoning engine and the representation it operated on.

Dimension	CGC (2016)	AIxCC (2024–2025)
Target format	Stripped DECREE binaries	Real open-source C/Java projects
Source code	None	Full source, build system, commit history
Reasoning engine	Fuzzing + symbolic/concolic	LLMs + fuzzing pipelines
Architecture	Orchestration loop	Orchestration loop (unchanged)
Vuln. detection rate	20 of 82 challenge sets in finals	86% of injected vulnerabilities
Patch success rate	Limited	68% correct patches
Finalist systems	7	7
Open-sourced	Partial	All 7 finalists

6 LLMs as the Missing Reasoning Engine

The cognition gap that limited CGC-era systems was not a matter of engineering effort: teams with deep expertise in program analysis, backed by years of DARPA funding, built the best systems the available techniques allowed. The limitation was in the techniques themselves: symbolic execution, constraint solving, and heuristic analysis are fundamentally incapable of the kind of open-ended reasoning about program behavior that human analysts perform.

Large language models represent a qualitatively different approach to this problem.

6.1 A Different Kind of Reasoning

What distinguishes LLM-based program analysis from CGC-era techniques is not speed or scale but the nature of the reasoning itself.

A symbolic executor asks: what inputs satisfy this path constraint? A fuzzer asks: does this mutated input produce new coverage? These are precise, narrow questions with mechanically verifiable answers. They are powerful within their scope but cannot express, let alone answer, the questions a human analyst asks when studying a program: what is this code trying to do? Is this parsing logic consistent with the protocol specification? Could this integer truncation lead to an exploitable condition three function calls later?

LLMs can engage with questions of this kind. They reason over code semantically, drawing on patterns learned from vast corpora of source code, vulnerability reports, and technical documentation. Their judgments are probabilistic rather than formally verified, but cover a far broader range of program behaviors than any formal technique can reach. These capabilities are strongest when LLMs operate on source code, where the representation retains the semantic cues (names, types, structure) that models were trained on.

Several specific capabilities are new:

- **Semantic understanding without formal models.** LLMs can identify vulnerability patterns in code they have never seen before, across languages and domains, without a formal specification of the program’s behavior.
- **Cross-domain generalization.** A single model can reason about protocol parsers, cryptographic implementations, memory allocators, and kernel interfaces—tasks that previously required separate specialized tools and separate human expertise.
- **Iterative, interactive analysis.** Rather than single-pass pipelines, LLM-based systems can hypothesize, test, and revise their understanding of a program over successive interactions.

6.2 A Familiar Architecture

What is remarkable about the emerging generation of LLM-based cyber reasoning systems is how much their overall architecture resembles what CGC demonstrated in 2016. In the AIxCC finalists, LLMs provided the reasoning that connected discovery, exploitation, and patching into coherent judgments [15]—inside the same orchestration loop. The structure is the same; the reasoning engine has changed.

Figure 2 illustrates this continuity.

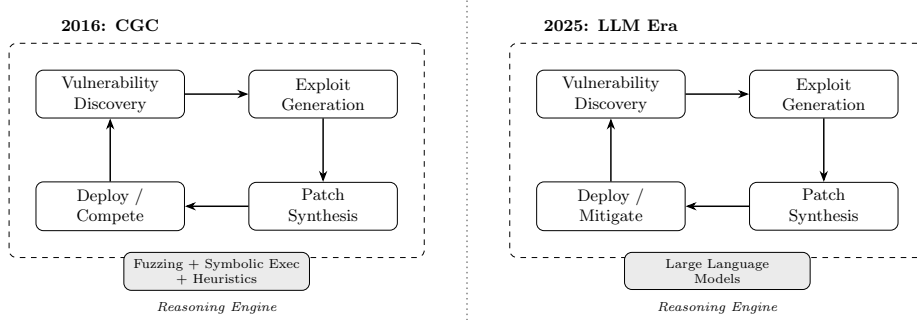


Figure 2: The autonomous cyber reasoning loop, 2016 vs. 2025. The orchestration architecture (the outer loop of discovery, exploitation, patching, and deployment) is unchanged. What has changed is the reasoning engine at the center.

This pattern is appearing outside competitions as well. Google’s Big Sleep,⁶ Microsoft’s MDASH, and startups such as XBOW and ZeroPath all pair LLM-based reasoning with CGC-shaped orchestration applied to real software. MDASH alone disclosed 16 Windows vulnerabilities, including four critical RCEs, in the May 2026 Patch Tuesday;⁷ members of its team came from AIxCC-winning Team Atlanta. The orchestration is CGC’s; the cognition is new.

6.3 Limitations: The Representation Gap

LLMs are not a complete solution: they can miss vulnerabilities, produce incorrect analyses, and generate patches that introduce new flaws. But the deeper limitation is structural: LLM capabilities depend heavily on the *representation* they operate on.

When given source code, with meaningful variable names, type declarations, comments, and structural context, LLMs can reason about program behavior at a level that approaches human analytical judgment. The AIxCC results

⁶Google Project Zero, “From Naptime to Big Sleep,” Nov. 2024; <https://projectzero.google/2024/10/from-naptime-to-big-sleep.html>.

⁷Microsoft Security Blog, “Defense at AI Speed,” May 2026; <https://www.microsoft.com/en-us/security/blog/2026/05/12/defense-at-ai-speed-microsofts-new-multi-model-agentic-security-system-tops-leading-industry-benchmark/>.

demonstrate this convincingly. But CGC was not a source-level competition. Its inputs were stripped binaries, by deliberate design.

When LLMs operate on decompiler output, mechanically recovered pseudo-code with synthesized variable names, lost type information, and no high-level structure, their performance degrades substantially. The degradation is not cosmetic; it stems from specific information that the compile/strip pipeline destroys and that decompilers can only guess back. *Type recovery* is heuristic: a struct pointer rendered as `int *` hides the field-level access patterns that identify a vulnerable parser. *Alias analysis* collapses without symbols and types, so recovered code conflates data flows that source-level analysis would keep distinct. *Control-flow reconstruction* is incomplete in the presence of indirect jumps, tail calls, and non-returning functions, propagating wrong CFGs into wrong logic. And the *abstraction cues* LLMs were trained to rely on (meaningful names, scoped lifetimes, type hierarchies) are simply absent, replaced by synthetic identifiers and flat structure.

These representation losses bear primarily on *discovery* cognition: they impair the ability to recognize what a function does or how data flows through it. The *exploitation* axis faces an additional set of representation challenges that source code does not help with. Heap layout depends on allocator runtime state (chunk boundaries, fastbin/tcache contents) that no static view exposes. Useful ROP and JOP gadgets exist below the instruction abstraction LLMs were trained on; source code does not contain gadgets, and decompiler output erases them. And the viability of a chosen primitive depends on which mitigations (ASLR, CFI, stack canaries, shadow stacks, CET) are active in the deployed system—a runtime question that neither source nor decompiler output answers. The binary-only problem has two cognition axes, and source recovery, even if it succeeded, would only address one of them.

A growing body of work attacks this representation gap directly, from learned binary embeddings [6] to decompiler-aware models that recover variable names from stripped output [8], but results remain far behind source-level analysis. The cognition gap that CGC exposed has narrowed for source-available software; the binary-only problem that motivated CGC’s design remains largely open.

7 The Hard Problem That Remains

CGC’s orchestration/cognition decomposition, confirmed by a decade of replication and by the AIxCC finalist architectures, clarifies what has been solved and what has not.

7.1 Source-Level Cognition: Narrowing the Gap

For source-available software, LLMs have dramatically narrowed the cognition gap. The AIxCC results (86% detection, 68% patching) demonstrate capabilities that would have been unimaginable in 2016. The question for source-level

analysis is shifting from “can we find the vulnerabilities?” to “can we process the results fast enough to act on them?”

But AIXCC, like CGC, used synthetic vulnerabilities in a controlled competition. Translating these results to the full complexity of real-world software, with its scale, its accumulated technical debt, and its absence of ground truth, remains an open engineering challenge.

7.2 Binary-Only Cognition: The Original Hard Problem

The problem DARPA designed CGC to address was not source-level analysis. The program’s chartering announcement framed the operational motivation as the Department of Defense’s reliance on “Commercial Off The Shelf (COTS) operating systems and applications” whose manual analysis “often [requires] skilled analysts to spend weeks or months analyzing a problem,” and defined the program’s evaluation artifact as “compiled test software (‘Challenge Binaries’ or ‘CBs’)” rather than source [5]. Stated plainly, the problem is this. The input is a stripped binary: no source code, no symbols, no debug information. From that input alone, an autonomous system must recover a semantic model of the program rich enough to do three things: identify vulnerability classes (discovery cognition), generate working exploits (exploitation cognition), and synthesize correct patches that preserve functionality. That is the problem CGC posed in 2016, and it is the problem that remains open today.

This is the problem that remains hardest, for the representation reasons detailed above. Early research on LLM-based binary vulnerability detection and automated reverse engineering shows promise, but the results are far behind what the same models achieve on source code. The cognition gap has closed fastest where the input representation is richest, and remains widest where the operational need is greatest.

7.3 Evaluation Beyond Competition

CGC’s answer to evaluating autonomous cyber reasoning—a controlled environment with known ground truth—carried forward into AIXCC and the Robot Hacking Games.

But deployment is not competition. In deployment, the ground truth is unknown; the whole point is to find vulnerabilities that humans have not yet found. Evaluating whether an autonomous system’s output is correct requires the same expertise that the system is meant to augment. The trust problem is compounded for binary-only analysis, where even verifying a system’s reasoning is harder without source-level context; evaluation frameworks for deployed autonomous cyber reasoning remain an open area of work.

8 Conclusion

The Cyber Grand Challenge was built on an implicit bet: that the right architecture (orchestration layer, evaluation framework, competitive structure) would carry the reasoning capabilities along with it. In 2016, it did not. The competing teams closed the autonomous loop, but the reasoning inside that loop was brittle and narrow, and for nearly a decade no incremental advance in fuzzing, symbolic execution, or binary analysis closed the gap.

Large language models have changed this, but unevenly. The cognition gap has two axes (Figure 1): discovery cognition, recognizing vulnerability classes in unfamiliar code, and exploitation cognition, turning a discovered flaw into a working exploit. For source-available software, LLMs have closed the gap along the first axis dramatically: AIXCC’s 86% detection rate shows discovery cognition operating at scale. Exploitation cognition lags far behind. As of this writing, the best published results are from ExploitGym, where frontier LLM agents solve 157 of 898 challenges (16%) *with mitigations disabled*; the gap is more stark with modern defenses active. The problem CGC was actually designed to address—autonomous reasoning about stripped binaries—remains largely unsolved.

The two axes also matter differently. Discovery uplift expands the supply of known flaws; the ecosystem has machinery for absorbing it: patching workflows, coordinated disclosure, continuous fuzzing, mitigation deployment. Exploitation uplift is different in kind: it changes the operational value of every previously known but unexploited flaw. A reasoning engine that makes exploit construction cheap reshapes the security landscape in a way that a reasoning engine that finds more bugs does not. That is the asymmetry the PoV requirement made unavoidable: even in 2016, CGC was not asking only whether a CRS could find a flaw, but whether it could produce a working exploit against it. It is the asymmetry to watch as autonomous cyber reasoning moves out of competition and into deployment.

The same decomposition is showing up well beyond cyber reasoning. The current generation of agentic AI systems (LLM-driven agents that write code, operate tools, and act over long horizons) is increasingly understood not as a story about smarter models but as a story about better *harnesses*: the scaffolding of tool interfaces, context management, and feedback loops that converts a capable model into a useful long-running worker—without which, practitioners observe, even frontier models fall short on extended tasks.⁸ That is the same lesson CGC surfaced a decade earlier in a narrower domain: the orchestration layer is load-bearing, and the intelligence that sits inside it is a separate problem.

CGC’s lasting contribution is the architectural decomposition it revealed: orchestration and cognition are separable problems, and within cognition, discovery and exploitation are separable axes. The orchestration problem yielded to engineering; the cognition gap is closing unevenly, axis by axis and rep-

⁸J. Young, “Effective Harnesses for Long-Running Agents,” Anthropic engineering blog, Nov. 2025; <https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents>.

resentation by representation. Binary-only cognition is the hard problem the security community has yet to crack—and because the two axes carry different consequences, it is not one problem but two.

References

- [1] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.
- [2] Dakota Cary. Robot hacking games: China’s competitions to automate the software vulnerability lifecycle. Technical report, Center for Security and Emerging Technology, Georgetown University, September 2021.
- [3] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P)*, pages 380–394, 2012.
- [4] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–278, 2011.
- [5] Defense Advanced Research Projects Agency. Cyber grand challenge (CGC): Automated cyber reasoning. https://www.esd.whs.mil/Portals/54/Documents/F0ID/Reading/20Room/DARPA/15-F-0059-CYBER_GRAND_CHALLENGE_BAA_14_05.pdf, November 2013. Broad Agency Announcement DARPA-BAA-14-05, Amendment 2, posted Oct. 22, 2013; Amendment 2 dated Nov. 27, 2013.
- [6] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (S&P)*, pages 472–489, 2019.
- [7] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, March 2012.
- [8] Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, Adam Doupé, Chitta Baral, and Ruoyu Wang. “len or index or count, anything but v1”: Predicting variable names in decompilation output with transfer learning. In *Proceedings of the 2024 IEEE Symposium on Security and Privacy (S&P)*, 2024.

- [9] Benjamin Price, Michael Zhivich, Michael Thompson, and Chris Eagle. House rules: Designing the scoring algorithm for Cyber Grand Challenge. *IEEE Security & Privacy*, 16(2):23–31, March 2018.
- [10] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (S&P)*, pages 138–157, 2016.
- [11] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, 2016.
- [12] Zhun Wang, Nico Schiller, Hongwei Li, Srijiith Sessa Narayana, Milad Nasr, Nicholas Carlini, Xiangyu Qi, Eric Wallace, Elie Bursztein, Luca Invernizzi, Kurt Thomas, Yan Shoshitaishvili, Wenbo Guo, Jingxuan He, Thorsten Holz, and Dawn Song. ExploitGym: Can AI agents turn security vulnerabilities into real attacks? *arXiv preprint arXiv:2605.11086*, May 2026.
- [13] Lok Yan, Benjamin Price, Michael Zhivich, Brian Caswell, Christopher Eagle, Michael Frantzen, Holt Sorenson, Michael Thompson, Timothy Vidas, Jason Wright, Vernon Rivet, Samuel Colt VanWinkle, and Clark Wood. DECREE: A platform and benchmark corpus for repeatable and reproducible security experiments. In *Proceedings of the 2018 IEEE Secure Development Conference (SecDev)*, 2018.
- [14] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symposium*, 2018.
- [15] 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, and Taesoo Kim. SoK: DARPA’s AI cyber challenge (AIXCC): Competition design, architectures, and lessons learned. In *Proceedings of the 35th USENIX Security Symposium*, August 2026.

Author Biography

Jason L. Wright is a clinical professor at Idaho State University in Pocatello, Idaho, USA. He was a member of the CGC infrastructure team, where he helped build the DECREE operating environment and built the independent scoring

system used to verify the final results of the competition. His research focuses on operating system security, deterministic execution, and network security. Mr. Wright received his MS in Computer Science from the University of Idaho. He earned a DEF CON CTF black badge in 2010 as a member of team ACME Pharm, and is a senior member of IEEE. Contact him at jlwright@ieee.org.