AgentFlow: Finding Security Holes Automatically

How AI Teams Hunt Zero-Day Vulnerabilities in Real Software

Simplified version of: Synthesizing Multi-Agent Harnesses for Vulnerability Discovery (arXiv:2604.20801, Apr 2026)

By Hanzhi Liu, Chaofan Shou, Xiaonan Liu, Hongbo Wen, Yanju Chen, Ryan Jingyang Fang, Yu Feng — UC Santa Barbara, UC San Diego, Fuzzland, World Liberty Financial

This paper solves a practical problem: how to automatically organize teams of AI agents to find real security vulnerabilities in software, without humans manually designing the search strategy each time. The system discovered 10 previously unknown vulnerabilities in Google Chrome, including 2 Critical sandbox-escape bugs that could let attackers take over your computer from a web page.

The Problem: Why Finding Bugs is Hard

Security researchers hunt for vulnerabilities — flaws in software that hackers could exploit. Traditionally, this requires highly skilled humans spending weeks or months reading code, crafting test inputs, and analyzing crashes. Recently, AI agents (large language models given access to tools like compilers and debuggers) have started finding real bugs that humans missed for decades.

Google's Big Sleep agent found a zero-day memory-corruption bug in SQLite. Anthropic's Claude Mythos found a 27-year-old denial-of-service bug in OpenBSD. But these successes hide a deeper problem: the way you wire AI agents together — called the harness — matters enormously, and most harnesses are designed by hand.

What is a Harness?

Think of a harness as the organizational chart for a security team. It decides: who does what (roles), who talks to whom (communication), what information gets passed along (messages), what tools each person can use, and when the team retries after failure. The same AI model, organized differently, can perform 4x better or worse depending on the harness.

On the TerminalBench-2 leaderboard, three systems all use the same Claude Opus 4.6 model, but their pass rates range from 20% to 80%. The only difference is the harness design. This means harness engineering is the bottleneck, not the AI model itself.

Two Key Limitations of Existing Approaches

1. Narrow Scope: Each existing optimizer only tunes a small slice of the harness. Meta-Harness rewrites prompts for a single agent. ADAS generates new agent code but fixes the communication graph. AFlow searches over a fixed library of workflow operators. None can make cross-component changes — like adding a new agent AND rewiring the communication graph AND adjusting its prompt in one step.

2. Coarse Feedback: Current systems use binary pass/fail signals. A harness that never reached the vulnerable code gets the same 'fail' signal as one that reached it but missed the error-handling branch. Without knowing WHY something failed, optimization becomes a random walk.

The Solution: AgentFlow

AgentFlow addresses both limitations with two innovations: a typed graph DSL (domain-specific language) that represents the entire harness design space, and a feedback-driven optimization loop that reads detailed runtime signals from the target program.

The Typed Graph DSL

AgentFlow represents every harness as a program in a typed graph language. The five components of any harness map directly into this language:

- Agent Set (A): Each agent has a role label (analyst, explorer, verifier), a prompt template, an LLM model, and a set of tools it can use.
- Communication Topology (G): A directed graph showing which agent's output feeds into which other agent.
- Message Schemas (Σ): Templates defining what information each edge passes. Templates can reference feedback channels like coverage data or sanitizer reports.
- Tool Bindings (Φ): Which tools each agent can invoke. An analyst might only read files, while an explorer can compile and run the target.
- Coordination Protocol (Ψ): How agents are composed — sequential chains, parallel fan-out, retry loops.

A single optimization step can add a new agent, rewire the communication graph, rewrite a prompt, and change tool bindings — all as one local edit to the same program.

The Type System

The type system ensures every proposed harness is well-formed before spending expensive LLM inference. Three rules: (1) every prompt variable must resolve to an upstream output or feedback channel, (2) every edge must connect agents that actually use the data, (3) the graph must be connected. About 20% of proposer outputs fail this check and are rejected before any computation, saving significant budget.

Runtime Feedback Channels

Instead of just pass/fail, AgentFlow reads four types of feedback from the target program:

Feedback Type	What It Tells You	Availability
Test Verdict	Whether the target's test suite passed or failed	Always
Program stdout/stderr	Raw error messages and diagnostic output	Always
Line/Branch Coverage	Which source-code lines were actually executed	Needs instrumentation
Sanitizer Reports	Memory-safety violations even when no visible crash	Needs instrumentation

Why This Matters: A Concrete Example

The paper walks through finding CVE-2020-23109, a heap-buffer-overflow in libheif (an image format library):

> Iteration 1: A single agent generates a fake HEIF file. The binary rejects it before the bug ever runs. stderr says 'format constraint not satisfied.' The harness knows the file was rejected but not why.
>
> Iteration 2: AgentFlow splits work into two agents — an analyst reads the source code to understand what a valid HEIF file looks like, and a crafter generates the file. Now the file passes format checks and the parser runs, but the bug's branch is never visited. Coverage data shows exactly which lines ran.
>
> Iteration 3: A verifier agent is added that starts from a real HEIF file and tweaks just the alpha-bit-depth field. With sanitizer feedback, it finds the exact value that triggers the heap-buffer-overflow. Three iterations, three different harness changes, all guided by runtime feedback.

The Optimization Loop

AgentFlow's optimization has four phases per iteration:

Phase	What Happens	Key Detail
Propose	LLM reads diagnosis + archive, emits a new AgentFlow program	Single call, ~30 seconds
Execute & Observe	Harness runs on every task, collects per-agent traces and feedback	Structured feedback per task
Score	Domain-specific function reduces feedback to a single number	Pass rate or unique crash count
Diagnose	LLM reads full feedback bundle, identifies bottleneck agent and root cause	Structured 4-field diagnosis

The Diagnose step is the key innovation. Its structured output has four fields:

Bottleneck agent: which agent most directly caused the failure.

Intended behavior: what that agent tried to do.

Actual execution: what the target program actually did, based on feedback channels.

Corrective edit: a description of what harness change would fix the problem.

This diagnosis feeds back to the Propose step, creating a closed loop where each iteration learns from the detailed reasons behind failures, not just the binary outcome.

Results: TerminalBench-2 Leaderboard

AgentFlow was evaluated on TerminalBench-2, a public leaderboard of 89 long-horizon terminal tasks including code translation, ML pipeline reconstruction, cryptanalysis, and vulnerability remediation.

System	Score (%)	Date
AgentFlow (this work)	84.3	2026-04-17
ForgeCode	81.4	2026-03-12
Capy	77.7	2026-03-12
Terminus-KIRA	77.3	2026-02-22
Meta-Harness	76.4	2026-03-30
TongAgents	74.6	2026-02-22
Droid	72.4	2026-02-05
Mux	69.0	2026-02-13
Crux	66.9	2026-02-23
Terminus 2	65.6	2026-02-06
Claude Code (bare)	60.9	2026-02-07

The synthesis trajectory climbed from 35.2% to 84.3% through three phases:

Infrastructure (Steps 1-5, +28.8 pp): Missing environment guards, tool bindings, and coordination-protocol fixes.

Specialization (Steps 6-9, +15.8 pp): Adding specialist sub-agents, retry edges, and tool refinements.

Ensemble (Step 12, +4.5 pp): Rewriting the topology into a fan-out/merge ensemble that runs independent attempts in parallel.

Ablation Study: What Matters Most

To understand which parts of AgentFlow contribute the most, the authors ran ablation experiments where they disabled different types of edits:

Variant	Score (%)	Drop
Full AgentFlow	84.3	—
No Structure Search (fixed topology)	76.4	-7.9
No Prompt Search (fixed prompts)	51.8	-32.5
No Tool Search (fixed tools)	71.9	-12.4

Key Findings:

> Prompt edits are the single largest contributor (-32.5 pp when disabled). The diagnoser's ability to identify which agent's prompt needs rewriting, and the proposer's ability to craft better prompts, drives most of the optimization.
>
> Tool edits contribute -12.4 pp. Restricting which tools each agent can use prevents agents from taking shortcuts that bypass the intended workflow.
>
> Structure edits contribute -7.9 pp. Adding specialist agents and rewiring the topology provides the scaffold on which better prompts operate.
>
> The three families are complementary, not redundant. Each enables the others to work better.

Real-World Impact: 10 Chrome Zero-Days

The most impressive result: AgentFlow was run on Google Chrome — one of the world's largest and most extensively audited open-source codebases (35+ million lines of C/C++) — and discovered 10 previously unknown zero-day vulnerabilities.

Chrome Component	Vuln Type	Severity	CVE
WebCodecs	Use-after-free	Critical	CVE-2026-5280
Proxy	Use-after-free	Critical	CVE-2026-6297
Network	Use-after-free	High	CVE-2026-4454
Codecs	Integer overflow	High	CVE-2026-5274
Rendering	Heap Buffer Overflow	High	CVE-2026-4462
Rendering	Use-after-free	High	494352590
Rendering	Heap Buffer Overflow	High	493534964
WebRTC	Heap Buffer Overflow	High	488803429
WebCodecs	Heap Buffer Overflow	High	488585490
WebGL	Inappropriate impl.	Medium	CVE-2026-5291

All 10 were accepted by the Chrome Vulnerability Reward Program and confirmed by Google. The two Critical bugs (CVE-2026-5280 and CVE-2026-6297) are sandbox-escape vulnerabilities — an attacker-controlled web page could execute code on the user's computer.

The Chrome campaign used Kimi K2.5 (an open-weight model, much cheaper than Claude Opus 4.6) and ran for 7 days on 24 cloud nodes with 192 H100 GPUs. The final harness had 18 agent roles with 192 parallel explorers across 7 Chrome subsystems.

The Chrome Harness Architecture

The synthesized Chrome harness is far more complex than the TerminalBench-2 harness. It has five phases:

Phase	Agents	What It Does
1. Analysis	Attack Surface Mapper, 7 subsystem analysts	Maps the attack surface and reads source code for each Chrome subsystem
2. Planning	Strategy Planner	Decides which subsystems to target and how
3. Exploration	192 parallel explorers (24-48 per subsystem)	Crafts and tests exploit inputs in parallel
4. Triage	Crash Filter, Root Cause Analyzer, PoC Minimizer	Filters crashes, analyzes root causes, minimizes proof-of-concept inputs
5. Validation	Repro Checker, Exploit Validator, Report Writer	Confirms reproducibility, validates exploits, writes reports

Six feedback loops drive iterative PoC generation. When an explorer fails, the feedback channels tell the diagnoser exactly why — whether the input never reached the vulnerable code, whether the crash was a real vulnerability or a false positive, and what to try differently.

This architecture was not designed by humans — it was synthesized by AgentFlow's optimization loop. The system discovered that the Chrome codebase needs subsystem-specific specialists, parallel exploration across attack surfaces, and multi-stage crash triage to handle the complexity of 35+ million lines of code.

Comparison with Prior Work

AgentFlow is the first system to search across all five harness dimensions simultaneously. Here's how it compares:

System	Agents	Topology	Prompts	Tools	Coordination
Meta-Harness	Fixed (1)	Fixed	Searched	Searched	Fixed
ADAS	Searched	Fixed	Searched	Searched	Fixed
AFlow	Fixed pool	Searched	Fixed	Fixed	Partially
MaAS	Searched	Fixed	Fixed	Partially	Fixed
OPRO	Fixed	Fixed	Searched	—	—
DSPy	Fixed	Fixed	Searched	—	—
AgentFlow	Searched	Searched	Searched	Searched	Searched

Multi-agent frameworks like AutoGen, MetaGPT, CAMEL, and ChatDev provide expressive runtimes but ship the topology as a constant — users must manually design the agent graph. AgentFlow treats the topology as a first-class search variable.

Coverage-guided fuzzers like AFL++ use coverage to steer input mutation for a single fuzzer. AgentFlow reuses the same instrumentation as one of several feedback channels, but the key difference is that AgentFlow searches the harness itself, not just the inputs.

Technical Details: The DSL

The AgentFlow DSL is a typed, graph-structured language. A program P = (N, E) is a labeled directed graph where N is the node set (agents, plus fanout nodes for parallelism) and E is the edge set.

Each agent node has four components: a role label (human-readable tag), a Jinja template string with free variables from upstream outputs and feedback channels, a model identifier, and a tool set.

The fanout operator fanout(n, k) clones a node into k independent copies. When a downstream agent references the family's output, it gets a JSON list of all k outputs. The join is implicit — whichever downstream agent references the family's outputs serves as the join point.

Edges can be unconditional ($n1 \to n2$) or guarded ($n1 \to_g n2$, where $g \in \{ok, fail\}$). A guarded edge fires only when the source agent's outcome matches the guard. This enables retry loops: if validation fails, the flow goes back to the analyst.

Well-Formedness Rules

Three rules ensure structural correctness:

- T-Agent: Every template variable must resolve to an upstream output or feedback channel.
- T-Edge: Every edge must connect to an agent that actually references the source's output.
- T-Conn: The graph must be connected — every node must be reachable from some source.

These checks run in linear time (single graph traversal) and reject about 20% of proposer outputs before any LLM inference is spent.

Practical Implications

What does this mean for bug bounty hunters and security researchers?

1. Harness Engineering is the Bottleneck

The same AI model can perform 4x better or worse depending on how you organize the team. AgentFlow automates this organization, achieving the highest score on the public TerminalBench-2 leaderboard with Claude Opus 4.6.

2. Rich Feedback Beats Binary Outcomes

Pass/fail signals are not enough. Coverage data, sanitizer reports, and program stderr give the diagnoser the information it needs to make targeted improvements. If you're running your own security campaign, invest in instrumentation.

3. Cross-Component Edits Matter

The best harnesses require changes across multiple dimensions — adding an agent while rewiring the graph and adjusting its prompt. Existing optimizers that fix one dimension miss these cross-cutting improvements.

4. Open-Weight Models Can Find Real Bugs

The Chrome campaign used Kimi K2.5 (open-weight, 1T parameters) — not the most expensive frontier model. AgentFlow's framework drove a mid-tier model to discover 10 real zero-day vulnerabilities in production code.

5. Scalability to Large Codebases

Chrome has 35+ million lines of C/C++. No single agent can hold this in context. AgentFlow's multi-agent approach with subsystem-specific specialists scales to codebases that exceed individual model context windows by orders of magnitude.

6. Budget Efficiency

The well-formedness type check rejects ~20% of proposer outputs before any LLM inference is spent. This is crucial because LLM inference is the dominant cost. A single proposer call takes 5-30 seconds, but evaluating a malformed harness on 89 tasks would waste orders of magnitude more compute.

Limitations and Future Work

Static Topology: AgentFlow searches static topologies — no runtime agent spawning or within-execution topology changes. Dynamic topologies that adapt during execution could potentially handle even more complex scenarios.

Source Code Required: The system requires access to source code and build infrastructure. It cannot hunt for vulnerabilities in binary-only targets.

Compute Cost: The Chrome campaign ran for 7 days on 192 H100 GPUs. While cheaper than manual security audits at scale, the compute cost is still significant.

Single-Subject Training: Like many AI systems, AgentFlow's effectiveness depends on the quality of the LLM models it orchestrates. As models improve, the harnesses they're organized into will likely improve too.

Future Directions

Extending to binary-only analysis, reducing compute requirements, exploring dynamic topologies, and applying the framework to other domains beyond security (e.g., automated testing, code review, performance optimization).

Ethical Considerations

All previously unknown vulnerabilities reported in this paper were disclosed to the affected vendor before paper submission. The ten Chrome vulnerabilities were filed with Google through the Chrome Vulnerability Reward Program in Q1 2026 and all ten were accepted by the vendor. Entries with public patch metadata currently span releases dated 2026-03-18, 2026-03-31, and 2026-04-15.

The released artifact is the harness optimizer source only. It does not include per-target proof-of-concept inputs, exploit primitives, crashing inputs, or trigger conditions for any Chrome vulnerabilities. This responsible disclosure approach ensures the security community benefits from the research while minimizing risk to users.

Quick Reference

Key Numbers to Remember:

Metric	Value
TerminalBench-2 Score	84.3% (highest on leaderboard)
Chrome Zero-Days Found	10 (including 2 Critical sandbox escapes)
CVEs Assigned	CVE-2026-5280, CVE-2026-6297, and 4 more
Improvement over Claude Code (bare)	+23.4 pp (60.9% to 84.3%)
Improvement over Meta-Harness	+7.9 pp (76.4% to 84.3%)
Largest ablation drop	-32.5 pp (disabling prompt search)
Chrome campaign compute	7 days, 192 H100 GPUs
Chrome codebase size	35+ million lines of C/C++
Open-source optimizer	github.com/berabuddies/agentflow

One-Line Summary:

AgentFlow automates the design of multi-agent security teams, achieving the highest score on a public hacking benchmark and discovering 10 real zero-day vulnerabilities in Google Chrome — all by treating the team organization itself as the thing to optimize.

Conclusion

AgentFlow demonstrates that automated harness synthesis can match or exceed hand-engineered security systems. By treating the entire harness design space as a single searchable program and using rich runtime feedback to guide optimization, it achieves state-of-the-art results on both a public benchmark and real-world production code. The 10 Chrome zero-days — including 2 Critical sandbox escapes — show that this approach works at scale on the most scrutinized codebases in the world.