FuzzingBrain V2

A Multi-Agent LLM System for Automated Vulnerability Discovery and Reproduction

Ze Sheng, Zhicheng Chen, Qingxiao Xu, Kewen Zhu, Jeff Huang
Texas A&M University

What if AI could automatically find security holes in software before hackers do?

This paper shows how a team of AI agents working together can find real bugs in real software.

Abstract

Software bugs that hackers can exploit (called "vulnerabilities") are a huge problem. In 2025, nearly 50,000 new security holes were reported - a record high. While AI language models (like ChatGPT) can help find these bugs, they have three big problems:

* They often cry wolf - reporting bugs that are not real (false positives)
* They cannot prove a bug is real by showing how to exploit it
* They struggle to understand complex bugs that span multiple parts of a program

FuzzingBrain V2 fixes these problems by creating a team of specialized AI agents that work together to find, verify, and prove real security vulnerabilities. The system discovered 41 previously unknown bugs in popular open-source software projects.

1. Introduction: Why Software Security Matters

Every piece of software you use - from your phone apps to your car's computer - has bugs. Some bugs are just annoying. Others let hackers steal your data, take over your computer, or crash important systems. These dangerous bugs are called "vulnerabilities."

The Growing Threat

In 2024, over 40,000 new vulnerabilities were reported. In 2025, that number jumped to nearly 50,000. That means every day, hackers and security researchers find more than 130 new ways to break into software.

Real-world example: In December 2025, a bug called "MongoBleed" (CVE-2025-14847) exposed over 87,000 MongoDB databases to anyone on the internet - no password needed. This is why finding and fixing bugs quickly matters.

Why AI Coding Makes It Worse

Here's the scary part: studies show that developers using AI coding assistants (like GitHub Copilot) tend to write LESS secure code while feeling MORE confident about it. AI helps people write code faster, but that code often has more security holes.

1. Introduction: How We Currently Find Bugs

Security researchers have two main tools to find vulnerabilities:

Static Analysis (Reading the Code)

This is like proofreading a book for spelling errors. The computer reads the source code and looks for patterns that might be dangerous. The problem? It finds tons of false alarms. Imagine a spellchecker that flags 100 words, but only 5 are actually misspelled. Security teams waste time checking all those false alarms.

Dynamic Analysis (Running the Code)

This is called "fuzzing" - you give the program random garbage as input and see if it crashes. Tools like AFL and libFuzzer do this millions of times per second. The problem? Fuzzing is blind. It doesn't understand what the code is supposed to do, so it misses bugs that require specific, clever inputs to trigger.

Key insight: What if we could combine AI's understanding of code with fuzzing's ability to prove bugs are real? That's exactly what FuzzingBrain V2 does.

1. Introduction: Three Problems With Current AI Approaches

Recent research shows that using AI language models (LLMs) to find bugs has three fundamental problems:

Gap 1: No Proof

When an AI says "this might be a bug," there's no way to verify it. It's like a doctor saying "you might be sick" without any test results. Security teams need proof - a specific input that triggers the bug - to take action.

Gap 2: Wrong Zoom Level

Looking at an entire function (hundreds of lines) is too much - the AI gets confused. Looking at a single line is too little - the AI lacks context. We need a "just right" level of detail that shows the vulnerable code with enough context to understand it.

Gap 3: Complex Connections

Many bugs involve multiple functions working together. The bug might start in Function A, travel through Function B, and crash in Function C. Current AI struggles to follow these complex chains of events across different parts of a program.

1. Introduction: What is FuzzingBrain V2?

FuzzingBrain V2 is a system that uses multiple AI agents working together to automatically find, verify, and prove security vulnerabilities in software. Think of it as a team of expert security researchers that never gets tired.

How It Works - The Big Picture

1. The system takes any open-source software project as input
2. AI agents analyze the code to understand what it does
3. Agents identify suspicious code patterns that might be bugs
4. Other agents try to prove the bug is real by creating an exploit
5. If successful, the system generates a report with proof

Real results: FuzzingBrain V2 found 41 previously unknown vulnerabilities in popular software, including ImageMagick, MongoDB, and BusyBox. 23 of these have already been fixed by the developers.

2. Background: How Fuzzing Works

Fuzzing is like testing a lock by trying millions of random keys. You give a program weird inputs and watch what happens. If the program crashes, you've found a potential security hole.

Popular Fuzzing Tools

* AFL (American Fuzzy Lop): The most famous fuzzer, uses genetic algorithms to evolve test inputs
* libFuzzer: Google's fuzzer, works well with modern C/C++ code
* Sanitizers: Special tools that detect memory errors (AddressSanitizer, MemorySanitizer)

OSS-Fuzz: Google's Fuzzing Service

Google runs a service called OSS-Fuzz that continuously tests over 1,000 open-source projects. FuzzingBrain V2 builds on top of OSS-Fuzz, which means it can immediately test any of those 1,000+ projects without extra setup.

Key advantage: By using OSS-Fuzz, FuzzingBrain V2 can prove every bug it finds is real by showing the exact input that triggers it.

2. Background: AI Agents and MCP

An AI agent is like a smart assistant that can use tools to get things done. Unlike a simple chatbot that just answers questions, an AI agent can:

* Read and analyze code files
* Run programs and see what happens
* Search for patterns in code
* Make plans and execute them step by step

Multi-Agent Systems

FuzzingBrain V2 uses multiple specialized agents, each with a specific job:

* Direction Agent: Figures out what the software does and divides it into sections
* SP Generator: Looks for suspicious code that might be a bug
* SP Verifier: Double-checks if the suspicious code is actually dangerous
* PoC Generator: Creates an input that triggers the bug to prove it's real
* Report Agent: Writes up the findings in a clear report

MCP: The Glue That Holds It Together

MCP (Model Context Protocol) is a standard way for AI to talk to tools. Think of it like USB for AI - any tool that speaks MCP can work with any AI that speaks MCP. This makes the system modular and easy to extend.

3. System Design: How FuzzingBrain V2 Works

The system works in three main stages:

Stage 1: Static Analysis (Understanding the Code)

First, the system reads all the code and creates a map of how functions call each other. This "call graph" helps the AI understand which parts of the code are reachable and how data flows through the program.

Stage 2: Agent Pipeline (Finding Suspicious Code)

Multiple AI agents work together to find potentially vulnerable code. They look for patterns like: copying data without checking sizes, using memory after freeing it, or trusting user input without validation.

Stage 3: PoC Generation (Proving the Bug)

For each suspicious finding, the system tries to create a specific input that triggers the bug. If it succeeds, the bug is confirmed as real. If not, the finding is discarded as a false alarm.

Two scan modes: Full-Scan analyzes the entire codebase. Delta-Scan focuses only on recent code changes - perfect for checking if a new update introduced any bugs.

3. System Design: The "Suspicious Point" Concept

The key innovation in FuzzingBrain V2 is something called a "Suspicious Point" (SP). This is a structured description of a potential vulnerability that includes:

* Which function contains the bug
* What kind of vulnerability it is (buffer overflow, use-after-free, etc.)
* A description of why it's suspicious
* How confident we are that it's real (score)
* Whether it's been verified and proven

Why "Suspicious Points" Matter

Instead of asking the AI to analyze an entire file (too much) or a single line (too little), SPs capture just the right amount of context. Each SP describes:

- The vulnerable function
- The dangerous code pattern
- How input reaches the dangerous code
- What would need to happen to trigger the bug

Example SP: "In the PNG image reader, when processing large images, the code copies pixel data without checking if the destination buffer is big enough. This could let an attacker crash the program or run their own code."

3. System Design: Parallel Workers

FuzzingBrain V2 is designed to run many analyses in parallel. Each "worker" handles one specific combination of:

* A fuzzer (which part of the code to test)
* A sanitizer (what type of bug to look for)

For example, if a project has 5 fuzzers and 3 sanitizers, the system can run up to 15 workers simultaneously. This means it can check for many different types of bugs at the same time.

Why This Matters

Different fuzzers reach different parts of the code. Different sanitizers detect different types of bugs. By running all combinations in parallel, FuzzingBrain V2 can find more bugs faster than systems that only use one fuzzer.

3. System Design: Logic-Driven Search

Instead of looking for specific bug patterns (like "buffer overflow"), FuzzingBrain V2 first understands what the software is supposed to do. This "logic-driven" approach has a big advantage.

Directions: Grouping by Feature

The system divides the code into "directions" based on business features. For example, in an image library, directions might be:

* PNG chunk parsing
* Color space conversion
* Image resizing

This is better than grouping by bug type because:

* Fuzzers are designed to test features, not find specific bugs
* Related code is analyzed together, giving better context
* High-risk features (like parsing untrusted input) can be prioritized

Key insight: By understanding what the code SHOULD do, the AI can spot when it does something it SHOULDN'T - which is often a bug.

3. System Design: Finding and Verifying Bugs

Step 1: SP Generation (Cast a Wide Net)

The SP Generator looks at each function and asks: "Could this be dangerous?" It intentionally casts a wide net - better to report 10 suspicious things and have 3 be real bugs than to report 2 and miss 1 real bug.

Step 2: SP Deduplication (Remove Duplicates)

Since multiple agents might find the same bug, a deduplication agent merges duplicate findings and keeps the most detailed description.

Step 3: SP Verification (Filter False Alarms)

The SP Verifier is the quality checker. It examines each suspicious point and asks:

* Can the fuzzer actually reach this code?
* Are there safety checks that prevent the bug?
* Is the description accurate?

Important: The verifier is conservative. When in doubt, it lets the finding through. It only rejects something if it's 100% certain it's not a bug. Missing a real bug is worse than checking a false alarm.

3. System Design: Proving Bugs Are Real

The final step is creating a Proof of Concept (PoC) - a specific input that triggers the bug. This is what separates real vulnerabilities from false alarms.

How PoC Generation Works

1. The PoC Generator reads the vulnerable function and understands the data flow
2. It designs an input that will reach the vulnerable code
3. It creates 3 different variants of the input
4. Each variant is tested - does it cause a crash?
5. If yes, the bug is confirmed. If no, try again with adjustments

Dual-Layer Fuzzing

The system runs two types of fuzzing in parallel:

* Global Fuzzer: Continuously runs in the background, exploring broadly
* SP Fuzzer: Focuses specifically on each suspicious point

This means while the AI is thinking about how to trigger a specific bug, the fuzzer is busy mutating inputs in the background. The fuzzer mutates while the AI thinks.

4. Implementation: Building the System

FuzzingBrain V2 is built in Python with these key components:

Technology Stack

* Python: Main programming language
* Redis: Message queue for distributing work to parallel workers
* MongoDB: Database for storing findings and reports
* libFuzzer: The fuzzing engine
* MCP (Model Context Protocol): How AI agents talk to tools

AI Model Tiers

The system uses different AI models for different tasks, balancing cost and capability:

* Tier 1 (Reasoning): Claude Opus 4.5, GPT-5.2-Pro - for complex analysis
* Tier 2 (Main): Claude Sonnet 4.5, GPT-5.2 - for code analysis
* Tier 3 (Utils): Claude Haiku 4.5, GPT-5-Mini - for simple tasks like deduplication

Smart cost savings: Using cheaper models for simple tasks and expensive models only for complex reasoning keeps costs down while maintaining quality.

5. Evaluation: How We Tested It

To prove FuzzingBrain V2 works, the team tested it on:

Test 1: AIxCC Competition Dataset

The DARPA AI Cyber Challenge (AIxCC) is a competition where teams build AI systems to find vulnerabilities. The dataset has 40 known vulnerabilities in popular software like curl, Wireshark, systemd, and libxml2.

Test 2: Real-World Open Source

The system was also deployed on real open-source projects through OSS-Fuzz to find previously unknown (zero-day) vulnerabilities.

Comparison

FuzzingBrain V2 was compared against:

* 6 teams from the AIxCC competition (including the winner)
* Claude Code (a general-purpose AI coding assistant)

Fair testing: Each test had time limits (2-4 hours) and cost limits ($150-400) to simulate real-world constraints.

5. Evaluation: The Results

Competition Results (AIxCC Dataset)

Out of 40 vulnerabilities, FuzzingBrain V2 found 36 - a 90% detection rate. This beat the AIxCC competition winner (Team Atlanta, 29/40) by 7 vulnerabilities. It also beat the previous version (FuzzingBrain V1, 14/40) by 22 vulnerabilities - a 157% improvement!

Hard Challenges

The researchers identified 12 "hard" challenges - bugs that most teams couldn't find. FuzzingBrain V2 solved 9 out of 12 (75%), significantly better than the competition winner (5/12).

Efficiency

Average time to find a bug:

* Delta-scan (checking code changes): 12 minutes, $19.40
* Full-scan (analyzing entire codebase): 18 minutes, $35.20

Total cost: $1,785.60 for all 40 challenges. That's about $45 per vulnerability found - much cheaper than hiring human experts.

5. Evaluation: Real-World Zero-Day Discoveries

The most impressive result: FuzzingBrain V2 found 41 previously unknown vulnerabilities in 19 real open-source projects. These are bugs that nobody knew about before.

What Happened After Discovery

* 41 vulnerabilities reported to developers
* 26 confirmed by project maintainers
* 23 already fixed in the software
* 2 received official CVE IDs (CVE-2026-23874, CVE-2026-23952)

Notable Discoveries

* OpenPrint CUPS: 6 vulnerabilities found
* fwupd (firmware updater): 4 vulnerabilities
* UPX (executable packer): 4 vulnerabilities
* ImageMagick, MongoDB, BusyBox, and more

Real impact: These are mature, well-tested projects that have been fuzzed for years. FuzzingBrain V2 found bugs in code paths that traditional fuzzers missed entirely.

7. Conclusion: What This Means for Software Security

FuzzingBrain V2 demonstrates that AI agents can effectively find real security vulnerabilities when combined with proper verification. The key insights are:

* AI + Fuzzing: Combining AI's code understanding with fuzzing's proof capability
* Suspicious Points: A new abstraction that captures bugs at the right level of detail
* Multi-Agent Teams: Specialized agents working together outperform single AI systems
* Real Results: 41 zero-day vulnerabilities found and fixed in popular software

The Future

The team plans to extend FuzzingBrain V2 to support more programming languages (Rust, Go, Python), add automatic patch generation, and work with binary-only software. This technology could eventually make software security testing accessible to everyone, not just big companies with dedicated security teams.

Read more research paper simplifications at:

https://medium.com/@muhamedfazalps7