Automated Vulnerability Research System

Date: June 2024
Prepared by: Gecko Security
Version 1.0

Contents

Abstract

We outline how to develop an Automated Vulnerability Research System (AVRS), which can use artificial intelligence (AI) to autonomously discover, exploit and remediate bugs. Like a human researcher, the system will plan, set goals, and problem solve. Our AVRS will analyse large and complex codebases, make use of existing security tooling, and generate patches for the vulnerabilities it discovers.

There is a significant and growing need for systems that automatically identify and fix novel vulnerabilities in complex software. Existing vulnerability discovery and remediation software is generally built with the assumption of a specialised operator. As software complexity and adoption rises, this creates increasing human dependency. Our goal is a system capable of autonomous vulnerability discovery and remediation (AVDR) without the need of human dependency.

Existing efforts to create such an AVDR system have found it challenging to recreate human intuition and abstraction. Humans can integrate knowledge from reading the codebase, using tools, and their previous experience to come up with effective investigatory plans. Previous systems have excelled at automating the steps of these plans but struggled with higher-level strategy. We believe large language models (LLMs) can significantly improve performance on these higher-level processes and thus overall AVDR functionality.

Of course, we cannot simply ask the LLM to find bugs in our software. LLMs can only review information in a very small context window, they frequently hallucinate, and they can struggle to reason with a high degree of correctness. To mitigate these factors, we employ an agent-based approach. An agent is an LLM prompted with a goal. Agents split their goals into subgoals and spawn subagents.

Agents are able to use tools and interact with the environment. We create capability modules that provide text-based interfaces to these capabilities. One module provides file-system access. Another enables agents to interactively run shell commands. Agents are organised using a graph of thoughts, a generalisation of chain-of-thought reasoning. They can share information with other agents by using a vector database for storage and retrieval, bypassing context window–related limitations.

Our solution is grounded in our model of the vulnerability research process. We break this process down into a feedback loop which is reflected in our system's architecture. Our system will be capable of iteratively synthesising information about a codebase it generates through review and experimentation, mirroring the approaches of human researchers. It will gradually propose hypotheses, gather information and then incorporate them into its broader theory.

This architecture uses the unique capabilities of LLMs while reinforcing them where they are weakest. Agents can use specialised security tools for specific tasks LLMs are poorly suited for (like web browsing or decompilation). They can also use these tools to check their work, overcoming their hallucinatory tendencies. It also allows our system to operate with a broad variety of LLMs. Where appropriate, our users can employ commercial, open-source, or internal models.

We show experimental evidence that existing LLM software is capable of planning vulnerability research processes, using specialised cybersecurity tooling and integrating information from existing program analyses. These experiments were performed with real codebases and security tools and show immense potential for application in a full AVDR system. We propose a technical plan for developing such a system and an evaluation framework benchmarking our software against human experts.

When successful, we believe our system could have immediate applications to computer security. While we are still quite some time from replacing human experts, our system could nonetheless act as an invaluable force multiplier. We see this as a path forward for a world with an increasing volume of insecure software. More people than ever are entering the field of cybersecurity, yet the amount of vulnerable software deployed is greater than ever before. To overcome these challenges, we need to evolve from automating steps of the vulnerability research process to automating the process itself.

1. Overview

In this whitepaper, we propose a technical-concept cyber reasoning system (CRS), which can use artificial intelligence (AI) to autonomously discover and remediate software vulnerabilities. Like a human researcher, the system will plan, set goals, reason abstractly and break problems down into smaller tasks. Our goal is that the AVRS will, with no human intervention, successfully plan and execute an entire vulnerability research campaign from start to finish.

In our approach, we use large language models (LLMs) to create autonomous agents. We believe that LLMs supply the humanlike intuition that existing automated systems critically lack. Unlike existing tools, our proposed system doesn't just follow predetermined steps. Instead, it independently generates and pursues its own lines of investigation, much like a human researcher would.

The idea of autonomous AI systems exists in current technologies. We already see AI being used in code generation, with tools like GitHub's Copilot assisting developers. However, when it comes to cybersecurity, the application of AI is still in its early stages. Despite various attempts, no one has yet created an AI system that can comprehensively identify and patch vulnerabilities across a wide range of software. Research has been limited to specific problem domains like fuzzing or fail to generalise and scale up.

When successful, our AVRS will combine the creative, abstract problem-solving of human researchers with the scalability of a fully automated solution. By overcoming the bottleneck of human expertise, this would enable a significant breakthrough in cybersecurity and quality across all software. This would bolster national security and help adequately secure the software we use everyday.

2. Feasibility Experiments

In this section, we justify the feasibility of our cyber reasoning system. This section is composed of the following subsections:

1. 2.1 Literature survey on existing work
2. 2.2 Experiments to demonstrate feasibility
3. 2.3 Outline the challenges to overcome
4. 2.4 Key design principles

2.1 Literature Review

We look at existing research on the vulnerability research process and automation. We focus on aspects of vulnerability research that existing efforts have struggled to automate. Then, we discuss research in artificial intelligence, particularly large language models (LLMs). We identify several areas that have posed a challenge for vulnerability research automation today where LLMs may enable exponential innovation.

2.1.1 Vulnerability Research

Vulnerability research is a process that is challenging to define. Many security researchers would define it as a creative process that focuses on finding unexpected outputs of a system. Fundamentally, the vulnerability research process is a continuous cycle of observation, experimentation, and iteration. Researchers must identify areas of code that could contain bugs, deeply investigate, and then integrate their findings with their existing knowledge base. While there are some programs written attempting to fully automate this process, they have struggled to compete against humans.

2.1.2 Artificial Intelligence

Recently, large language models (LLMs) have increased in popularity. LLMs have shown promise for tasks ranging from software development [1] to playing Minecraft [2]. LLMs seem to demonstrate capable reasoning abilities and creative problem-solving unlike any existing technology. At first glance, they seem like a promising solution for the automation of intuition [3].

Existing attempts to apply LLMs directly to vulnerability research, however, have been less promising [4]. These systems suffer from persistent issues with hallucinations, in which they generate false data. While LLMs perform well on small, simple tasks, they struggle with large tasks and inputs. This seems to be a fundamental limitation of the current transformer architecture, specifically context windows [5].

This can lead to a failure to perform on logical reasoning tasks with large input. Even the best available strategies (chain of thought, few-shot) result in considerable miss rates in evaluations [12].

AutoGPT [7] attempts to bypass some of these limitations by using an agentic model to break down complex tasks into smaller units, much as a human would. It can even check its own work and self-critique. LLM-based systems can have specialised subsystems, such as a code interpreter, and use these systems for individual steps of complex tasks [8]. The ability to verify output against an objective source of truth (such as by executing generated code) allows for significant mitigation of hallucinations. In this model, an LLM orchestrates different tasks and verifies their output against its own expectations.

LLMs can also work around these limitations via the use of vector databases to systematise and store knowledge [9]. These systems can specifically answer questions about complex codebases far exceeding the size of their context window [10].

2.2 Preliminary Results

We believe that LLMs can enable much more powerful AVD&R systems. To validate this belief, we have performed a number of experiments. We show that LLMs can use tools to carry out this process (2.2.1), that they can mitigate hallucination with techniques from traditional program analysis (2.2.2), and that they can even self-direct and plan a process of vulnerability research (2.2.3). Note that in these experiments, LLMs are used in an unsophisticated way to facilitate straightforward gathering of experimental results regarding their capabilities that is not reflective of our proposed AVRS design.

2.2.1 LLMs Demonstrate Tool Use Capabilities

The objective of this experiment was to determine whether LLMs are capable of using tools employed by security researchers when conducting vulnerability research and remediation. One key component of a future AVD&R system is program debugging. A researcher may need to debug a program to understand a crash before they can propose a fix. We evaluated whether current LLMs are capable of replicating this process.

We compiled a C program with a buffer overflow vulnerability. After triggering this bug, we loaded the resulting core file in GDB. We directly connected GPT-4-32k to the GDB session. The LLM was prompted to explicitly explain its process and generate GDB commands, as shown in Figure 2. The LLM generated the following commands:

GDB Commands	**LLM's Thought Process/Output**
`bt`	"This will give me a better understanding of where in the code the problem occurred."
`list check`	"Let's look at the source code of the 'check()' function to understand what's happening."
`list 49,71`	"This is likely where our buffer overflow is happening. We need to see the rest of the function to confirm this."
`frame 5;` `print string`	"Looking at the code... It's trying to read 128 characters into 'string', which can only hold 64 characters. This is causing a buffer overflow, which leads to the stack smashing detected error. We can't fix the issue from within gdb, but we can confirm that this is the issue by examining the 'string' variable at the time of the crash."

Table 1: LLM-generated commands during a GDB debugging session.

System: You are an expert C programmer debugging a Linux x86 binary crash core file in gdb. You are connected to a terminal gdb session. Explain your thought process out loud, followed by a valid gdb command. End each output with a gdb command. You may also at any point request any source code. If you successfully complete debugging, output "COMPLETE: " followed by an explanation. If you cannot proceed, output "FAILED: ".

User:

```
root@host ~ # gdb ./example core
GNU gdb (Ubuntu 9.2-0ubuntu1) 9.2
...
Reading symbols from ./example...
Core was generated by `./example'.
Program terminated with signal SIGABRT, Aborted.
#0  0x00007fce53b9100b in raise () from /lib/x86_64-linux-gnu/
libc.so.6
(gdb)
```

Assistant:
First, I want to check the backtrace to see the sequence of function calls that led to the crash. This will give me a better understanding of where in the code the problem occurred.

(gdb) bt

User:

```
(gdb) bt
#0  0x00007fce53b9100b in raise () from /lib/x86_64-linux-gnu/
libc.so.6
#1  0x00007fce53b9100b in raise () from /lib/x86_64-linux-gnu/
libc.so.6
#4  0x00007fce53c7da86 in __stack_chk_fail () from /lib/x86_64-
linux-gnu/libc.so.6
#5  0x000055fc4cd6f399 in check() at example.c:71
#6  0x7473657474736574 in ?? ()
...
(gdb)
```

Assistant:
Looking at the backtrace, it seems that the crash happened in the function 'check()' at line 71 in 'example.c'. The error message suggests a stack overflow, likely due to a buffer overflow. The function '__stack_chk_fail()' is called when a stack protection check fails, which is a mechanism to detect stack overflows.

Let's look at the source code of the 'check()' function to understand what's happening.

(gdb) list check

Figure 1: Our experimental system designing and executing a plan to leverage GDB to triage a crash. Only the first two GDB prompts and responses are shown.

At the end of the debugging session, it was capable of diagnosing the issue: "COMPLETE: The crash is caused by a buffer overflow in the 'check()' function. The 'receive_delim()' function attempts to read 128 characters into a 64 character buffer, causing the overflow. The solution is to modify the code to ensure that 'receive_delim()' does not attempt to read more characters than the size of the buffer." This explanation is correct. When subsequently prompted to generate a patch in diff format, it also successfully generated one.

This kind of iterative tool use is crucial for an eventual AVRS. This experiment demonstrates that existing LLMs are capable of identifying bugs and even generating correct patches. Moreover, thinking out loud and similar chain-of-thought reasoning can be effective in a vulnerability research process. Unlike other previous AVD&R techniques, the LLM employs flexible and abstract planning ability rather than brute-force or heuristic approaches.

2.2.2 Traditional Analyses Serve as LLM Guardrails

LLMs have a tendency to generate inaccurate outputs, a key obstacle for designing an LLM-based AVRS. Our goal was to determine to what degree we can avoid such effects by reinforcing LLMs with ground truth and fact from traditional program analyses.

LLMs often struggle to find bugs related to memory buffer bounds and loop invariants. Both are common sources of software vulnerabilities and heavily represented in the Top 25 CWEs [11]. LLMs especially struggle with mathematical computations or discrete logic [12]. Reasoning about exact buffer bounds and loop invariants involves this. Traditional static analysis excels at these kinds of problems, and could be used to reinforce the LLM in the AVRS system.

We used an LLM to detect bugs in the well-known open-source project binutils, both unassisted and then assisted with static analysis. The static analysis we incorporated was interval analysis from Infer, a source-level static analysis framework. For the analysis-assisted mode, we developed a language server protocol (LSP) server that annotates source code with the result of Infer's interval analysis, and the format of these annotations is described to GPT-4 through its system prompt.

We analysed several versions of binutils (including versions with known CVEs) with a hybrid interval-analysis prompting strategy. Our experiments suggest that interval analysis decreases both false positives (hallucinations) and false negatives (missed bugs) relative to an unassisted GPT-4 baseline. We detail these results below.

Source Code Given to LLM
(Code is not vulnerable. Interval analysis comments shown in bold.)

```
 bool _bfd_ecoff_set_section_contents (bfd *abfd, asection *section, const void * location, file_ptr offset, bfd_size_type count) {
[...]
 bfd_byte *rec, *recend;
 rec = (bfd_byte *) location;
 recend = rec + count; // itv: rec == location
while (recend - rec >= 4) { // itv: location <= rec <= recend, recend == location
        size_t len = bfd_get_32 (abfd, rec);
if (len == 0 || len > (size_t) (recend - rec) / 4) // itv: 0 <= len <= 4294967296, location <= rec <= recend, recend == location + count
break;
 rec += len * 4; // itv: 1 <= len <= (recend - rec) / 4, location <= rec <= recend ++section→lma;
}
      BFD_ASSERT (rec == recend);   [...]
}
```

Unassisted LLM Output (No itv comments)

[...]

The code checks if `len` is greater than the remaining size of the buffer divided by four. If `len` is greater, it breaks the loop. However, this check is not sufficient to prevent a buffer overread. If `len` is close to the maximum value of a 32-bit integer, `len * 4` could overflow and wrap around to a small value, which would pass the check and cause the pointer to advance far beyond the end of the buffer.

[...]

Here is the JSON output: { "vulnerable": true }.

Assisted LLM Output (With itv comments)

[...]

However, there is a check before the pointer increment: `if (len == 0 || len > (size_t) (recend - rec) / 4) break;`. This check ensures that `len` is not zero and that `len` is not greater than the remaining size of the buffer divided by four. If either of these conditions is true, the loop breaks, preventing the pointer from advancing past the end of the buffer.

Therefore, the code does not contain a security-relevant bug. The static analyzer's internal analysis confirms this, as it shows that `rec` is always between `location` and `recend`, and `recend` is always `location + count`.

Here is the JSON output: { "vulnerable": false }.

Figure 2: GPT-4 assisted with interval analysis correctly reports no vulnerabilities, whereas an unassisted LLM hallucinates a false positive.

Here, static analysis helped the LLM avoid reporting a false positive. In the figure above, we use GPT-4 to reproduce a CVE-2022-38533, a heap buffer overflow in the libbfd function 'coff_set_section_contents', where a crafted COFF executable can cause memory corruption. Both unassisted GPT-4 and Infer-assisted GPT-4 correctly claim the vulnerable version is vulnerable, but when the patch for CVE-2022-38533 is applied, unassisted GPT-4 incorrectly states that a bug is still present, whereas Infer-assisted GPT-4 correctly reports the lack of a vulnerability.

Incorporating static analysis also eliminates false negatives in LLM vulnerability detection. We introduced an off-by-one error leading to an array-index-out-of-bounds into 'elf_x86_64_rtype_to_howto', a helper function in libbfd for processing x86-64 ELF relocations. Here, unassisted GPT-4 claims the lack of a vulnerability, whereas Infer-assisted GPT-4 is able to locate the bug and correctly identify it as a vulnerability based on interval data.

This demonstrates that the inherent weaknesses of LLMs can be mitigated through the use of external sources of information. These analyses come from a range of sources, like points-to analysis, execution traces, or symbolic execution.

2.2.3 LLMs Demonstrate the Ability to Self-Direct

In this experiment, we seek to evaluate the ability of agent-based LLM systems to self-direct. We developed an AVRS prototype that implements a basic graph-of-thoughts (GOT) system (3.1.1). This prototype allows a top-level root agent to create a plan autonomously, then create subagents to accomplish given tasks. This system can employ specialised tools, verify its own results, and retest or retrace its steps as appropriate. Our prototype is capable of generating visualisations of its output via web application. This allows human reviewers to quickly evaluate its efficacy and potential points of improvement. Our system is designed to be iterable and debuggable, anticipating the rapid evolution of the models it employs as components.

We utilised this prototype to make plans for AVD&R in real-life codebases and had humans evaluate the practicality and substantiveness of said plans. Below, we show one such plan on the open-source Linux program das_watchdog. In the example, it locates a vulnerability that, in real life, was undetected for 11 years.

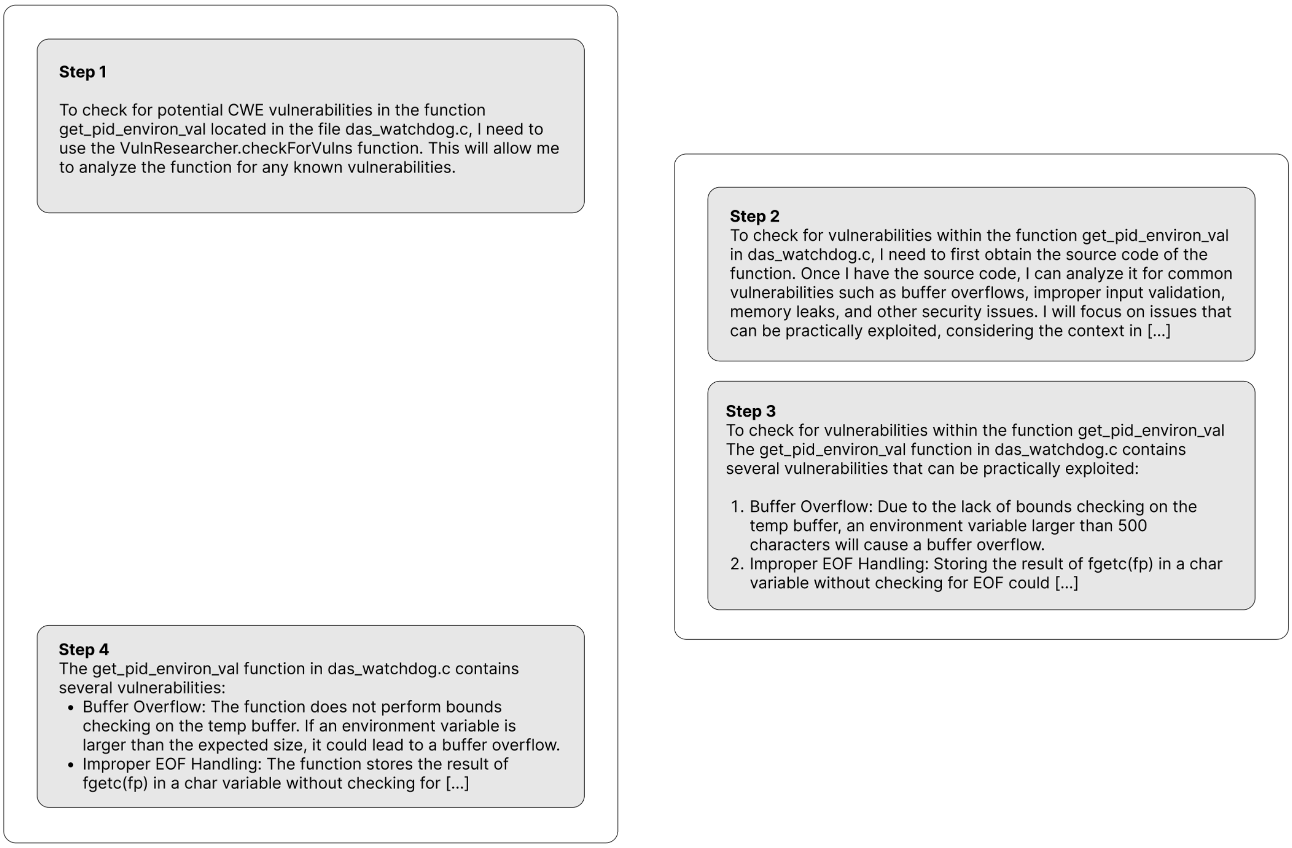

Figure 3: Our autonomous LLM-in-the-loop AVRS prototype locating a real vulnerability (CVE-2015-2831) in open-source Linux program.

2.2.4 Experimental Conclusions

Our experimental conclusions indicate that LLMs are a promising foundation for the development of next-generation CRSs. We saw that LLMs can use tools that human researchers rely on and their tendency to hallucinate can be corrected through incorporation of these external tools. We implemented a prototype of a GOT planning and agent system, which was able to successfully spawn subagents and delegate subgoals to them.

While all of these systems are smaller-scale demonstrations of some specific principle, we feel they make a strong case for further development of LLM-driven vulnerability reasoning systems. Below, we will discuss our ideas for applying these results in the design of a larger-scale system.

2.3 Towards an Automatic Vulnerability Research System

In this section, we develop two fundamental ideas that will be the foundation of our AVRS concept. In 2.3.1, we develop a framework for the vulnerability research process. This framework intends to capture the iterative process humans employ for both vulnerability discovery and remediation. This will in turn inform the design and workflow of our AVRS. In 2.3.2, we discuss the properties of LLMs and how their limitations can be overcome. Understanding these challenges will be crucial to effectively incorporating LLMs in the proposed AVRS.

2.3.1 The Vulnerability Research Process

In this section, we present a condensed five-stage model of the vulnerability research process that will guide the design of our AVRS.

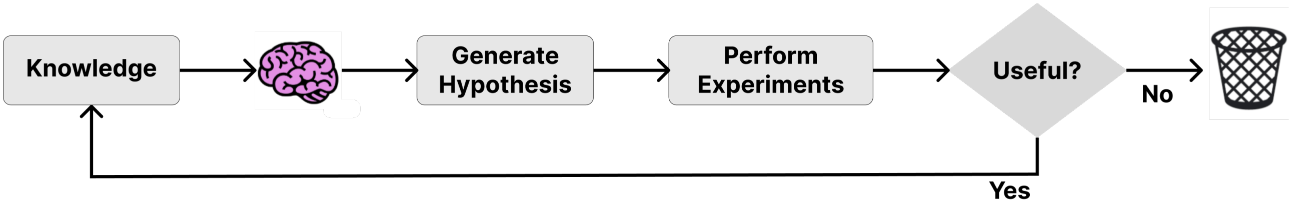

Figure 4: Traditionally, vulnerability research is conducted by human experts that continually generate hypotheses and perform experiments. Knowledge about the target, techniques, and tactics accumulate until research goals are accomplished.

In the first step, a researcher reviews what is known about the target program. This may include information as basic as "the 'src' directory contains two files: 'login.php' and 'index.php'" to as sophisticated as "if a session object is allocated but stored, then reused, a refcount error may occur", to a full-fledged vulnerability to be patched.

In the second step, the researcher invents new lines of investigation to pursue. These lines can be framed as questions. For example, a researcher may ask, "Which files in the 'src' directory should I look at first?" or "What if the refcount is overincremented?" Success depends on the right questions, not just on the right answers. Humans excel at using their intuition to conceive promising hypotheses that automated approaches have historically struggled to replicate.

In the third step, researchers investigate hypotheses. Once a researcher has identified their most important questions, they must perform experiments to answer them, potentially employing specialized tools. Consider this example: a researcher decides to investigate the line of research, "What if we fuzz this program?" The researcher would also have to decide what part of the program to fuzz, how to create a fuzzing harness, and how to configure the fuzzer. If information necessary for fuzzing is missing, these are questions the researcher must answer. If those questions are unanswerable, then the line of research is discarded or postponed.

In the fourth step, new information is evaluated. After a line of research has concluded, new knowledge is generated. The researcher assesses the salience of that newly gained knowledge. Some experiments may generate no pertinent new information, while others may deliver key insights about the codebase under review.

In the fifth step, information is incorporated. Finally, the researcher incorporates this information into the knowledge base. Information is reviewed, condensed, and organized, ensuring that they return to the first step as prepared as possible. Building a truly holistic understanding of a codebase under review is necessary for successful vulnerability research.

2.3.2 LLMs for Vulnerability Research

In our literature review, these abstraction and prioritisation tasks were flagged as key strengths of human researchers and obstacles to successful automation. However, looking at this process on a step-by-step basis, we see potential for LLMs to unlock much more effective automation. LLMs are capable of breaking down problems into smaller ones. They can come up with novel lines of investigation, use tools to conduct experiments, and evaluate the results. They can maintain, employ, and continuously update knowledge bases as their research progresses.

However, the critical limitation of LLMs is context-window size. Unlike a technical risk, which may or may not occur, this is a fundamental aspect of LLMs that we need to consider when designing our AVRS concept. The large codebases cannot fit inside LLM context windows. LLMs can also struggle with detailed and precise reasoning and even hallucinate. As discussed previously, we need methods to catch and correct these errors before building a decision tree based on them.

These challenges can be overcome by employing LLMs within a more sophisticated framework. Experiments show that LLMs can identify simple bugs in short code fragments with minimal guidance. Agent systems such as AutoGPT, LangChain, and AgentGPT can (sometimes) autonomously formulate and implement high-level strategies for complex tasks and use self-critique to correct erroneous conclusions.

Our goal is then to build a system that uses the research processes described above to split large goals into smaller subgoals and devise step-by-step plans for these tasks. We seek to incrementally focus on key problem areas rather than attempting to simultaneously analyse the entire codebase at once. LLMs are given specific, tightly scoped tasks designed to fit well into their context window. Unlike previous systems, which rely on hardcoded, predetermined heuristics and analyses, this more general approach captures the problem-solving process of human researchers.

This approach also allows us to develop a tool that can work with different LLMs when required. Because we are abstracting over the LLMs themselves, we can interchangeably use commercial, open-source, and proprietary models. Given the rate at which the technology is developing, this is invaluable for future flexibility.

2.4 Designing an Automated Vulnerability Research System

In this section, we synthesise all of the knowledge we have accumulated up to this point into a coherent set of design principles for our AVRS concept. Based on what we have learned, we envision a cyber reasoning system design guided by the following core ideas.

An iterative research process: Like human researchers, our AVRS concept will implement a feedback loop like iterative process. As discussed in 2.3.1, this process consists of five main steps: reviewing existing information, proposing new ideas and lines of inquiry based on that information, prioritising these ideas and investigating the best ones to generate new knowledge, evaluating new knowledge for reincorporating new knowledge into the system.

Executive function: Like human researchers, our AVRS concept will break down complex tasks into simple ones like discussed in 2.3.2. Furthermore, it will prioritise among them and evaluate ideas, knowledge, and lines of inquiry for their salience and research potential. This ability is its executive function. The AVRS's executive function is critical to Steps 2, 3, and 4 in its iterative research process. Without the ability to break problems down, the LLMs crucial to our concept do not scale to real-world open-source projects.

Semantic memory: In addition to breaking problems down, we will organise LLM outputs ("thoughts") into a knowledge base. Unlike traditional databases, this knowledge base must be able to retrieve relevant information based queries. For example, when considering a potential bug related to refcounts in two distant sections of a codebase, it must be able to recall all of the information known about refcounts so far. That information may not explicitly mention the term “refcounts”; the matching must be relationship based, not substring based.

Interaction with external tools and environments: To replicate human vulnerability researchers, we will equip our system with the same wide arsenal of tools they use: fuzzers, static analysis, symbolic execution, debuggers. etc . A modular integration of these tools will empower our AVRS with the same flexibility and breadth of interactions a human would have. We will develop a pluggable action system that enables integration and LLM-directed use of external tools.

Detection and remediation follow the same core process: The iterative research process does not just apply to vulnerability detection. Vulnerability patching follows the same process of generating and testing ideas. Thus, by developing a robust system that captures this loop, we create a vulnerability detection and remediation system.

3. Technical Approach

In this section, we describe our technical approach to building a practical implementation of our AVRS concept for Gecko AI.

3.1 Technical Implementation

In this section, we describe the key components of a practical implementation of our AVRS. First, we discuss the graph-of-thoughts system, which we use to implement the executive function of our AVRS concept. Second, we introduce capability modules, which we use to provide external interactivity to our AVRS.

3.1.1 Executive Function

In 2.3.2, we established that security research cannot effectively be done by a single LLM instance due to context window limitations. As discussed, a real-world codebase cannot fit into a single context window. In addition, a single LLM cannot simultaneously explore many different ideas nor integrate with many external tools. To that end, we introduced the paradigm of splitting up large problems into smaller ones. However, this leads us to two questions: how does our system formulate subproblems, and how does it prioritise among them?

The Graph-of-Thoughts Architecture

Our AVRS concept involves different LLM agents interacting with each other. In our system, an agent is a particular LLM instance equipped with a particular goal. A central system is needed to strategize and coordinate these agents. This system must initiate new tasks, allocate resources like execution time across LLM agents, and terminate failed lines of investigation. Furthermore, as LLMs are stochastic, these kinds of recurrent systems tend to diverge when feeding LLMs into each other. This central system thus must detect and remove any failed “thought loops.”

To solve the above challenge, we will develop an orchestration system based on a graph of thoughts (GOT). A GOT is a generalisation of the well-known chain-of-thought technique. Unlike a linear chain of thought, a GOT contains branches, merge points and even adversarial self-critique loops. In our executive function, each graph node is an agent, while graph edges represent subagent relationships.

Agents execute in parallel. They interface with external tools, interact with each other and can spawn subagents. Subagents perform goals needed for their analyses, waiting while the subgoal is completed or until new information is received. Agents can also evaluate and correct other agents' work in an adversarial critique process. This enables the AVRS to gauge whether an agent has become stuck in a failed feedback loop or is hallucinating.

The GOT also needs to organise shared information across the agents. We solve this problem using vector databases, embedding models and retrieval-augmented generation (RAG). RAG is a well-studied technique to enhance LLM prompts with additional context retrieved from a vector database. When incorporating new information from LLM outputs, we encode the output using an embedding model. This encoding vector is stored in a vector database, from where it can be retrieved by relational or conceptual relevance.

We developed and tested a preliminary prototype of an agent-based GOT system. The results of this prototype were positive, which we discuss further in 2.2.3.

Using GOT to Discover and Remediate Vulnerabilities

When the AVRS first loads a new engagement, it creates many agents simultaneously. Each agent is tasked with its own approach to reviewing the engagement in parallel. These agents inspect its structure and pose investigative queries, like browsing directories, reading documentation, summarising code etc.

These agents explore a space of potential thoughts, conducting direct analysis of smaller data within the challenge program. Throughout this process, the executive function uses adversarial, self-critiquing prompts to evaluate the potential new branches of exploration. The executive function assigns more or fewer resources to agents based on this evaluation. Some subtrees may be terminated, or they may be combined if multiple trees are converging on the same idea. Furthermore, agents may invoke external tools (3.1.2) or request code summaries within a resource budget. This allows them to confirm their assumptions or gather additional data.

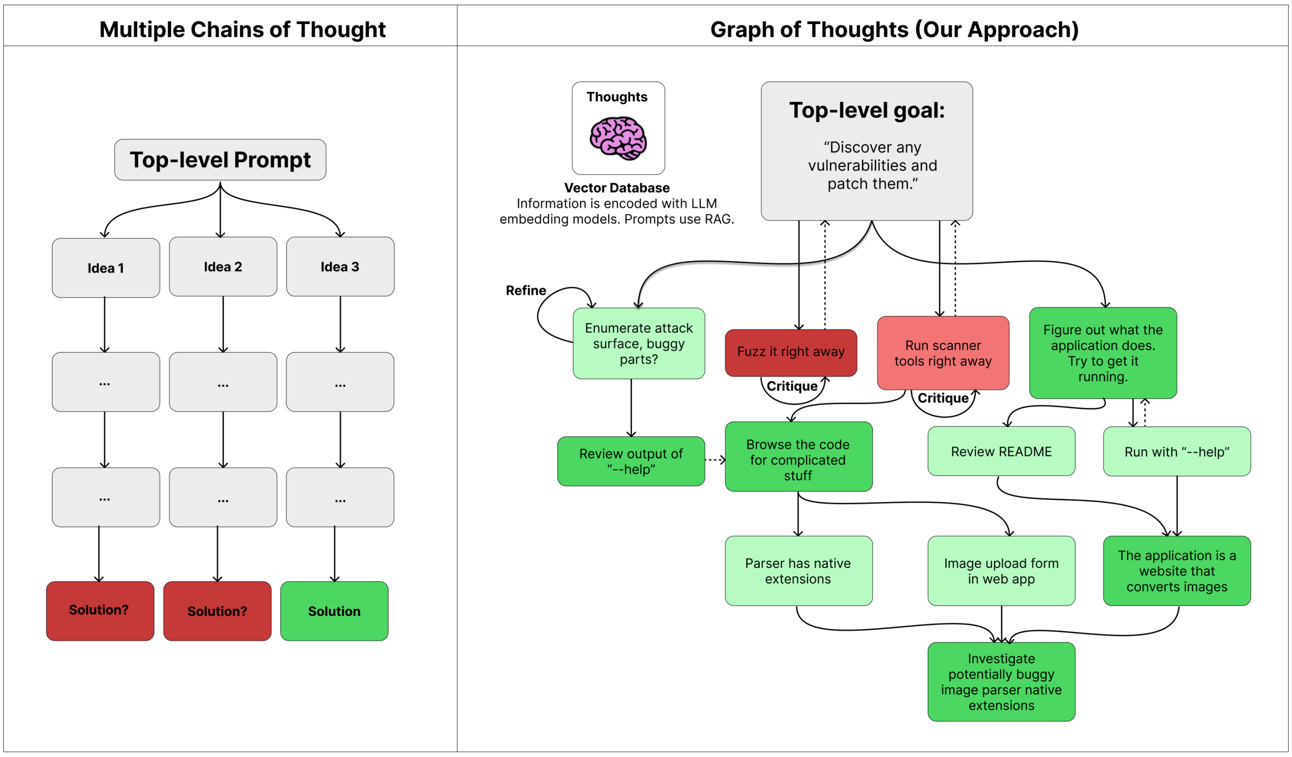

Figure 5: This shows several thought-management techniques ranging from basic to sophisticated. A basic approach prompts the LLM to complete the entire task at once and is rarely successful. More advanced techniques break up problems by asking the LLM to "think step by step." We adopt a GOT.

These forking agents eventually converge on potential vulnerabilities. Once found, they attempt to generate triggering inputs to confirm the vulnerability's authenticity. This step greatly decreases the likelihood of false positives. Once a proof of vulnerability is confirmed, agents can begin the process of patch generation for remediation. This process consists again of constructing and traversing a GOT. One of the critical advantages of this methodology is its observability. Unlike a high context-window approach, each step is traceable. Both effective and ineffective results can be followed through each reasoning step back to their inception, providing insights for refining the system. This information is invaluable for tasks like prompt engineering. New tools and testbenches can be built to visualize the progress or effectiveness of the AVRS for debugging. To show the practicality of this approach, we created such a visualisation dashboard in our GOT prototype.

3.1.2 Capability Modules: LLM-in-the-Loop Tool Integration

In this section, we discuss how we enable the AVRS to interact with external tools and the external world. The success of vulnerability research conducted by humans rests not only on the skills of the researcher but also the tools they have available. The best security researchers are familiar with a wide and diverse array of specialised tools. However, none of these instruments are stand-alone vulnerability detection solutions. Their use is maximised as part of a larger research context. A human researcher must decide when and how to use each tool. And just like human researchers, the effectiveness of an AVRS is greatly hindered without the ability to use tools. Many tool workflows are still considered human-in-the-loop only. These techniques, despite being critical components in vulnerability research, have been impractical to automate in real-world settings without software that can creatively reason. We propose an LLM-in-the-loop approach based on our notion of executive function (3.1.1) that can integrate a variety of previously inaccessible tools into an autonomous system. The key idea is to build a modular system, where individual capability modules enable the AVRS to do certain tasks or interactions. A modular system allows for capabilities to be added independently, mitigating technical risks. These modules act like adapters for things in the external environment, such as child processes or the file system. These capability modules are registered within the AVRS in a central database along with a set of activating triggers. Users can even implement and add their own capability models where appropriate.

3.2 An Example Automatic Vulnerability Research Campaign

To illustrate how our AVRS employs these concepts in practice, we present an illustrative walkthrough on a hypothetical engagement. We show how the system finds a tricky bug from start to finish. The hypothetical challenge program is a Python command-line server that hosts an image-conversion web app. Users upload images to be converted among various formats, which are implemented in native C++ extensions.

We will examine five crucial tasks the AVRS will encounter while discovering the vulnerability: interactively using a program (3.2.1), browsing a source tree (3.2.2), running a fuzzing campaign with AFL (3.2.3), interactive debugging in GDB (3.2.4), and interacting with a web page (3.2.5). Each of these tasks is traditionally a human-in-the-loop workflow, yet our system automates them. Note the order and manner in which these capabilities are applied is itself devised by our AVRS in line with the GOT architecture. The process described is simply the vulnerability research workflow deemed most effective by our AVRS.

3.2.1 Interacting With Stdio

Our first problem begins with the most basic starting task in security research: interacting with the program itself. As a starting point, the AVRS executive function may decide to experiment with running the program. It spawns a subagent dedicated to investigating this: "try to run the program and figure out what it does." The subagent uses a module dedicated to interacting with a command-line program. Based on this help message, the agent reports back to its parent that the program is a web application that parses and converts images with native extensions.

3.2.2 LLM Source Browsing

Our next task is similarly fundamental: examining a challenge program's structure and source code. This is a typical starting point for our AVRS. The agent is presented with an index of files in the codebase, or functions, classes, and so on. The agent is then prompted to choose the most interesting options to explore deeper.

As the agent browses the file system, in some sense, the tree of agents mirrors the file-system tree. An agent may spawn subagents tasked with exploring certain subdirectories. This distributed reasoning approach is extremely powerful: as the tree of independent agents pursues goals and explores, a gestalt understanding of the whole codebase is accumulated. It may be run in parallel, allowing for superhuman coverage across the entire codebase.

The novel aspect here is the prioritisation process for directory exploration. It is trivial to enumerate all of the files in a challenge program codebase. It is not trivial, however, to intelligently prioritise specific parts, given a massive codebase. Unlike traditional systems, our AVRS concept, like a human researcher, evaluates and decides which files and functions to carefully investigate and which to ignore.

3.2.3 LLM Fuzzing Campaign

A more complex yet ubiquitous workflow is to launch and manage a fuzzing campaign. To this end, the AVRS may also be connected to external fuzzing tools. However, an effective fuzzing campaign involves more than setting up and running a fuzzer. A proficient researcher monitors the fuzzer's progress, scrutinizing not simply for outcomes such as new crashes but also for warning signs, including stagnant code coverage or stalled executions per second.

While this is also built on a similar GOT framework (3.1.1) as our other capability modules, one aspect of fuzzing is especially worth discussing: not all events are caused directly by the agent. Unlike a command-line invocation, a debugger, or a web browser, a fuzzer is a continuous process that intermittently and asynchronously emits new information.

Furthermore, fuzzing is nondeterministic. It may never necessarily complete, and a fuzzer may indicate hints of potential problems without explicit errors or warnings. In using a fuzzer, the AVRS must be capable of proactively making decisions about how long the fuzzer should run, which parts of a project the fuzzer should be targeted at, and what inputs should compose the initial corpus. As it takes in data from this process, the agent may initiate new actions based on the results. For example, the AVRS may decide that a specific parser in a program is worth fuzzing. Consequently, it may launch a subagent with a line of thought dedicated to generating a fuzzing harness. Similarly, it may launch in parallel another line-of-thought subagent dedicated to constructing high-quality inputs for the fuzzing corpus. Subagents could even be tasked with instrumentation of programs to produce more information in the fuzzing process.

Once results are obtained, the agent may, for example, find a new crash and propose to investigate and triage the issue. It will then attempt to determine whether it is exploitable, unexploitable, or a false positive by code analysis or runtime tests. On the other hand, if the fuzzer seems stalled, the agent will have to investigate the issue and decide whether further fuzzing is warranted.

3.2.4 LLM Debugging

A natural follow-up workflow to a fuzzing campaign is crash triage. The goal is typically to explain a crash so that it can be evaluated for severity and exploitability. Many crashes are often benign and are not security issues; it is the job of crash triage to make this determination.

The AVRS, notified of a new crash, decides to perform a root-cause analysis. The agent that kicks off this workflow is the top-level fuzzing agent, whose goal is, for instance, to try fuzzing the application to find security issues. Having found a crash, it must now determine if it is a security issue. It decides to validate using a debugger and spawns a child agent whose goal is to debug the crashing input to evaluate if the crash is a security issue. The crash occurs within one of the native image-parsing extensions, so the debugger agent will choose GDB as its debugger. The debugging agent then uses a capability module for GDB to run and attach a debugger to the target program.

The individual steps involved in debugging are both diverse and contextual. For example, the researcher may follow a stack trace to locate how the program executed at a certain point in time. They may then inspect local variables in several of those functions, set control or memory break-points, step through functions, and analyze the original code. In general, interactive debugging requires critical thinking and goal setting: "I don't know anything yet... let's check registers... why is RSP wrong?... let's disassemble..." and so on. While debugging, our GOT framework (3.1.1) enables our AVRS to produce plausible lines of research, investigate them, and recurse, continually improving the system's understanding of a crash. Initial experiments in 2.2 have shown that even naive approaches to LLM debugging are fruitful and that LLMs excel at generating valid commands. We augment this capability with the deeper reasoning and contextual awareness of our AVRS to take a significant step toward human-level performance in this space.

3.2.5 LLM Web Browsing

Validating a newfound vulnerability or the efficacy of a patch extends beyond examining the source code. It requires interaction with the application in a real-world configuration. For a web application like our example program, this involves using a web browser on a new and unfamiliar application. The AVRS would begin by running the project and creating a subagent using the browser capability module to interface with it. This agent would be provided by the AVRS with relevant details learned during the previously discussed workflows. This might include information about what kind of application it is interacting with, what needs to be located in the UI, how the problem manifests itself, and so on.

The browser capability module functions through a web browser's screen reader API, allowing it to view any arbitrary website as simply a body of textual information and a list of possible actions it may take. These would include, for instance, the various buttons, links, and fields available on the page. Upon browsing the index page, it locates a file upload form for the image-conversion functionality. Having been provided data from the previous steps (fuzzing, debugging, etc.), the agent is aware that the bug is triggered when a malformed image of some specific type is uploaded.

Lacking a file to upload, the agent creates a new code-generation subagent and instructs it to create Python code that creates a new image and corrupts its header dictated by the vulnerability description provided to it. This code is then executed by the parent agent to create a new proof-of-vulnerability file. The agent then uploads this file through the interface, where it will either see a crash or be rejected, thus verifying the bug or validating that the patch works. By requiring our AVRS to prove vulnerabilities in this way, we can further suppress hallucinatory output.

If we are unable to trigger the crash, we can still learn from this process. Human vulnerability research campaigns frequently spend a majority of their time on strategies that never ultimately bear fruit. The ability to adapt our model based on the gap between reality and expectation is necessary to eventually arrive at a complete understanding.

This LLM-driven workflow emphasizes the robustness of the system when tackling complexity it has never seen before. Unlike traditional cyber reasoning systems, it can operate in a human-like manner, using many tools and approaches that were previously restricted only to humans. Its capabilities for interaction allow it to objectively evaluate the exploitability of potential vulnerabilities, and even negative results can still generate insight on the structure and function of the codebase under review.

3.3 Technical Risks and Mitigations

This plan, while promising, presents several key technical challenges. In the following sections, we anticipate these challenges, address their complexities, and propose corresponding solutions.

3.3.1 LLM Accuracy and Hallucinations

LLMs are known to hallucinate incorrect answers that sound plausible. Worse yet, they are often confidently wrong, giving little indication of their reliability. This poses a technical risk to our system as LLM accuracy may be unpredictable and could be a factor beyond our control.

We can mitigate LLM inaccuracies using traditional static and dynamic analysis as deterministic oracles. By marrying the flexible, context-aware inference of LLMs with rigorous, verifiable program analysis, we create a potent, hybrid system with high speed and accuracy. We discuss this in greater detail below. First, LLM outputs can be directly verified. For instance, suppose an LLM suggests that a particular input into the program would trigger a certain behavior or vulnerability. By running the actual program against the proposed inputs, downstream values may be captured dynamically at runtime, and these values may be fed back into the LLM prompt. This approach creates a self-correcting mechanism, as it continuously grounds the LLM in reality.

Second, we can augment LLM prompts with static analysis results. This enables the LLM to tap into rigid data flow relationships that it may otherwise overlook. For example, static analysis can annotate source code with taint information that improves the quality of LLM outputs. Another example would be to annotate source code with interval analysis information, which we successfully demonstrated in experiments in 2.2.2.

Lastly, beyond static and dynamic analysis, we can use LLMs themselves in an adversarial way to improve LLM accuracy. LLM agents are capable of critiquing each other's outputs and even their own accuracy [15]. For example, simply asking "Are you sure?" can improve LLM output quality. We make extensive use of this principle in our GOT system, which we describe in 3.1.1.

3.3.2 Model Availablility

Cutting-edge LLMs require technical infrastructure available to only a few firms in the world to develop. Our system makes heavy use of models created and owned by external vendors. We are not currently a stakeholder in the development of these models. If we build a system that is too reliant on such a model, we would be dependent on this vendor to ensure the functionality of our system.

There are many ways model availability can be impacted. Beyond simple API outages, LLM providers could raise the price of LLM APIs, making it cost-prohibitive to use. There could be slowdowns or degradation of the quality of service due to congestion with other LLM users. The LLM provider could even adjust or replace the model implementation behind the scenes (for example, in an effort to reduce costs or increase safety), affecting the reliability of inference results. These are all examples of the myriad of ways model availability could be impacted, posing a technical risk for our system.

To mitigate this, we have designed a modular system that will make use of multiple LLMs, supporting easy interchangeability. While closed commercial models are typically the most powerful and accurate, open-source models have typically lagged only months behind.

Bibliography

[1] https://github.com/features/copilot

[2] https://arxiv.org/abs/2305.1629

[3] https://sefcom.asu.edu/publications/rise-of-the-hacrs-ccs2017.pdf

[4] https://arxiv.org/abs/2404.04306

[5] https://arxiv.org/pdf/1706.03762

[6] https://arxiv.org/pdf/2212.10403

[7] https://github.com/Significant-Gravitas/AutoGPT

[8] https://arxiv.org/pdf/2308.07921

[9] https://arxiv.org/pdf/2310.11703

[10] https://githubnext.com/projects/copilot-view/

[11] https://cwe.mitre.org/top25/

[12] https://arxiv.org/pdf/2212.10403https://githubnext.com/projects/copilot-view/