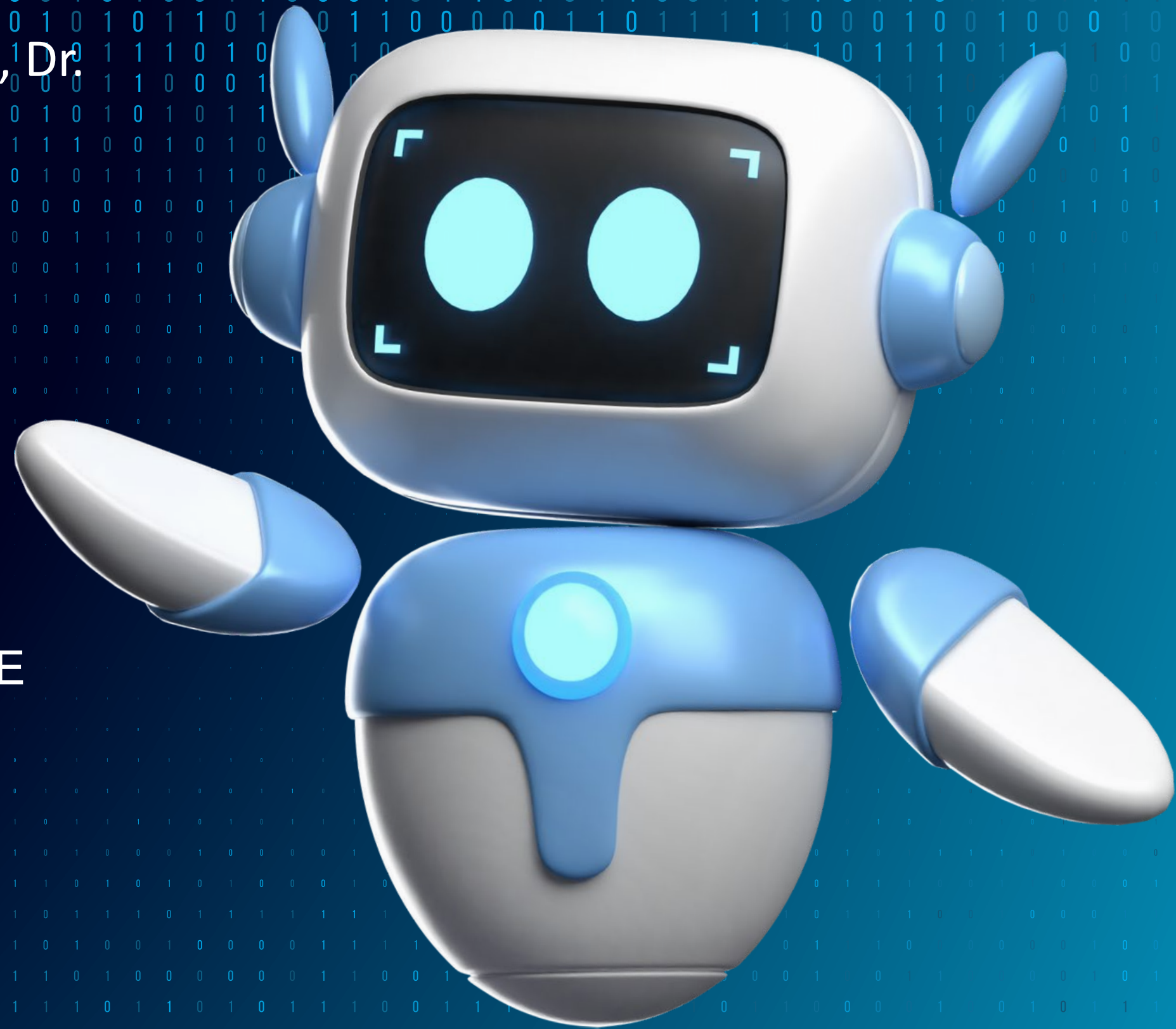


Presenting for the University of North Georgia:
Cole Watson, Dr. Bryson Payne, Dr. Denise McWilliams, Dr.
Mingyuan Yan

LEVERAGING LLMS

TO INCREASE THE EFFICIENCY OF MALWARE
ANALYSIS AND REVERSE ENGINEERING



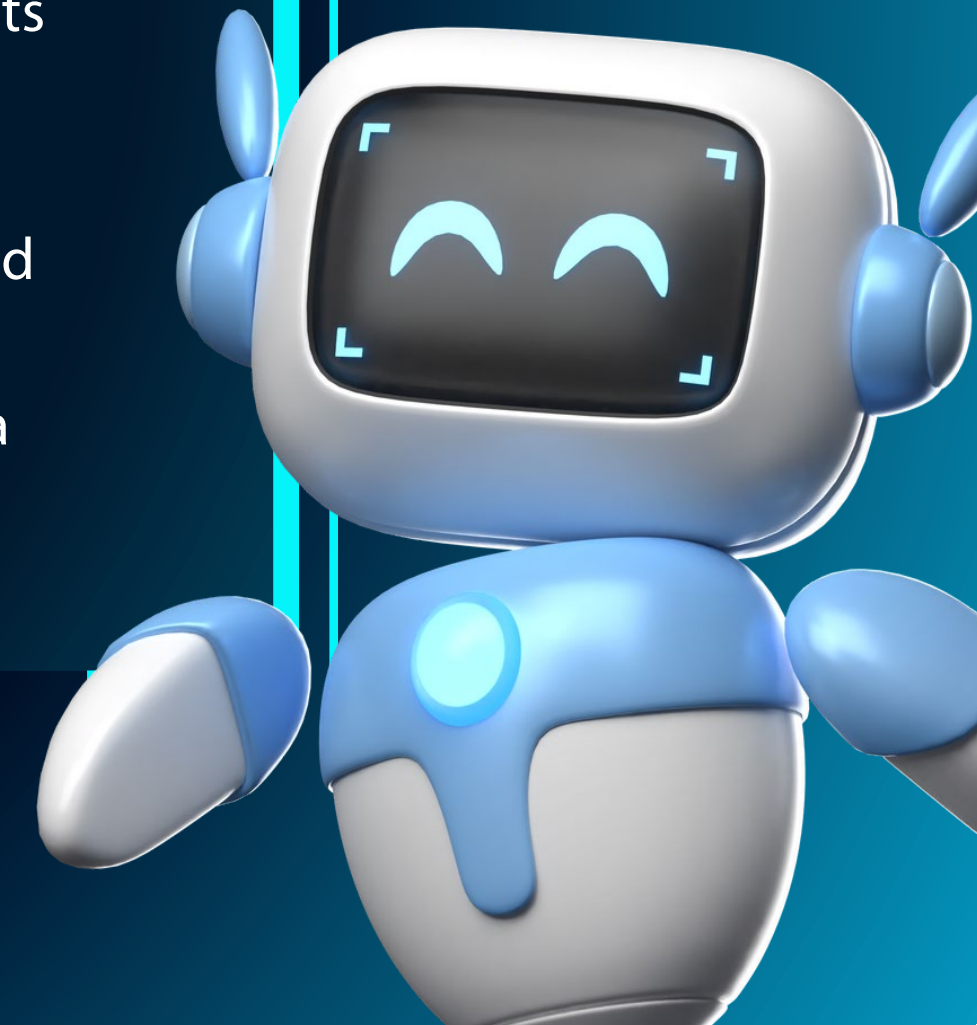
Background and Motivation

Why reverse engineering (RE) is difficult:

- Specialized skill; expertise takes years to develop
- Malware stacks defenses: obfuscation (packing, string/XOR/RC4), anti-debug (timers, env checks), anti-disassembly (opaque predicates, CFG flattening)
- Manual triage is slow and cognitively heavy, especially during incidents

Why should we seek AI assistance:

- LLMs can summarize de-compiled code, suggest function names, and outline de-obfuscation steps
- Goal: cut time-to-insight and reduce analyst workload while keeping a human in the loop





Tool Overview

Ghidra (v10.3.2)

- Primary RE environment: disassembly, decompiler, and scripting
- Source of truth for code navigation, labels, comments, and patches

GhidraMCP (plugin)

- Bridges Ghidra ↔ LLM: sends pseudocode/symbols/context; returns summaries, labels, and draft scripts
- Inserts AI suggestions back into Ghidra (comments/renames) for human verification

Claude (via Claude Desktop)

- LLM used for code summarization, de-obfuscation scaffolding, and patch planning ideas
- Operated strictly human-in-the-loop; all outputs verified in Ghidra

Environment & Supporting Tools

- Windows VM; repo of Escalate binaries; consistent project settings across runs
- Python for quick checkers/decryptors; minimal custom scripts to validate LLM suggestions

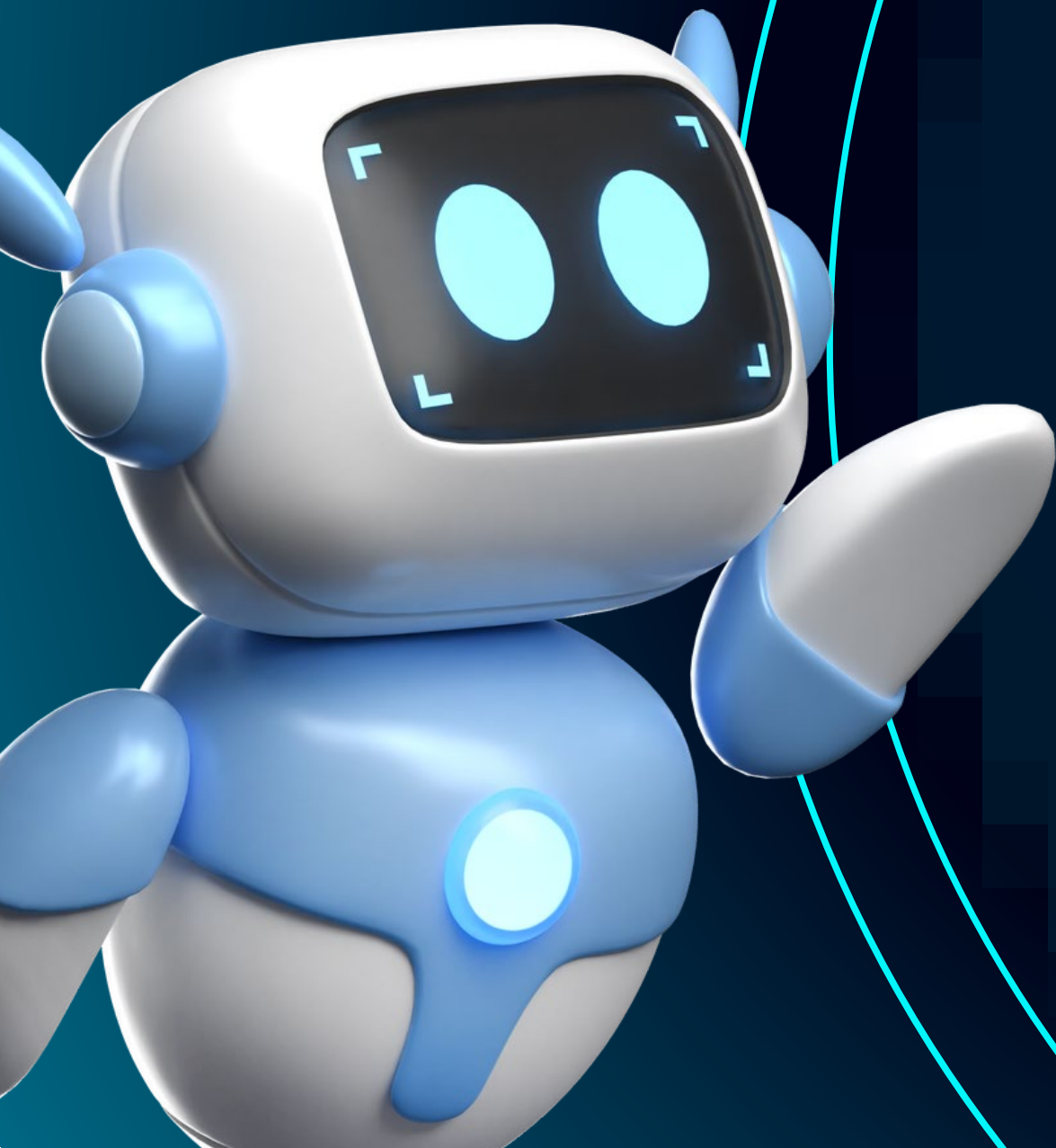
Methodology

- Design: Within -subject comparison on Escalate levels —each solved twice: manual baseline vs. LLM -assisted (via GhidraMCP).
- Setup: Windows VM, Ghidra 10.3.2, fixed auto -analysis; each level isolated in its own project, binaries unmodified unless patching required.
- Baseline workflow: Auto -analyze → inspect strings/symbols → explore control flow → recover algorithm → validate via checker/decryptor/patch.
- LLM-assisted workflow: Same steps + targeted prompts for naming, algorithm ID, and script scaffolding; suggestions verified before adoption.
- Independence: Levels analyzed separately; no prior solutions shared; all prompts, outputs, and scripts logged.
- Measurement: Time-to-first-valid-solution, correctness, and number/quality of accepted labels & comments.



Methodology

- Prompt construction: Minimal targeted context (decompiler slice + addresses); explicit tasks (algorithm ID, label suggestions, decryptor outline); rule: “no guesses —cite bytes/addresses.”
- Verification loop: Cross -check in disassembly/ decompiler ; run small Python checkers; require intermediate data (keys/offsets) before acceptance.
- Hallucination handling: Treat outputs as hypotheses → test immediately; log and correct errors (e.g., key endianness).
- Bias & reproducibility: Fixed VM/Ghidra setup; unmodified binaries; fresh sessions per level; predefined acceptance criteria.
- Safety: CTF binaries only; no proprietary samples; patches used solely for behavioral validation.



Results Overview

Faster labeling/summarization;
clearer function roles and data
flow

Early Wins

First hallucination (fabricated
flag) detected via per
verification -byte XOR

Level 3

RC4 correctly identified; key
endianness fixed after targeted
prompt → consistent decrypts

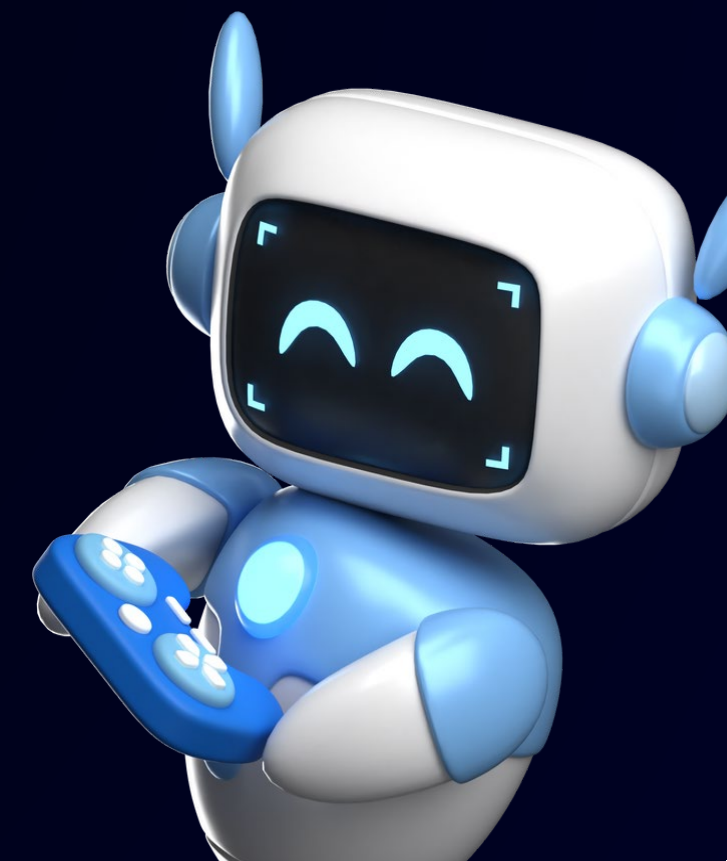
Level 4+

Static guidance helped navigate
anti-debug and anti-disassembly
regions without immediate dynamic
runs

Protections

Useful for patch planning ideas;
human validation remained
essential Learning Applications

Upper bound



Deep Dive: Early Levels L1-L3



- L1–L2: Plaintext flags and named functions (e.g., main, check_password); LLM accelerated labeling + explanation
- L3 (XOR): Model inferred XOR but hallucinated a flag; fix = prompt to xor each byte and show intermediates
- Outcome: Correct flags recovered; verified by minimal Python checker and in `ghidra` - Ghidra inspection

Deep Dive: RC4 Levels L4+



- Detection: LLM identified RC4 state init/keystream generation from decompiler slices
- Issue: Initial key interpretation was wrong endianness → bad output
- Prompt fix: "Treat key as little-endian; show KSA/PRGA steps and offsets" → valid plaintext
- Result: Consistent decrypts across RC4 levels after endianness correction

Deep-Dive: Anti-Debug / Anti-Disassembly



- Observed: Timing checks, environment probes, opaque predicates, misleading CFG fragments
- LLM help: Highlighted guard branches, suggested bypass points (e.g., condition inversion/NOP windows)
- Human step: Validate jump tables/edges; ensure no unintended side effects before any patch
- Payoff: Reduced time spent wandering protected paths; no unsafe patches accepted

Example of Assisted Solve

can you analyze this file in ghidra

I can help you analyze a file in Ghidra! I have access to various Ghidra tools that can help with reverse engineering tasks.

Let me start by getting an overview of what's currently loaded in your Ghidra session:

get_current_function

Great! I can see you have a program loaded with an entry function. Let me gather more information about the program structure:

list_functions

I can see this is a large program with many functions. Let me also check what imports and exports are available:

list_imports

list_exports

Let me examine the entry function to understand what this program does:

decompile_function

Now let me examine the main function that gets called from entry:

decompile_function

This looks like a password checker program! Let me examine the password checking function:

I've completed a comprehensive analysis of your binary in Ghidra. Here are the key findings:

This is a password validation program that uses XOR encryption. The program:

1. Takes a password as a command line argument
2. Uses XOR encryption with key `0xB9` to decrypt a stored password
3. Compares the decrypted password with user input
4. Prints "You achieved Level 3!" if correct, or failure message if wrong

The most interesting part is the password validation function at `0x00401737`, which stores 39 encrypted bytes and then XORs them with `0xB9` to reveal the actual password.

To solve this challenge, you would need to:

1. Extract the encrypted bytes from the function
2. XOR each byte with `0xB9`
3. Convert the result to ASCII to get the password

Would you like me to:

- Dive deeper into any specific function?
- Help you extract and decrypt the password?
- Analyze other aspects of the binary?
- Look for any anti-analysis techniques?

The full analysis report is available above with all the technical details!

can you give me the flag

I'll help you decrypt the password/flag from the binary. Let me extract those encrypted bytes and XOR them with the key `0xB9`:

Analyzed data

View analysis

Found the flag!

Flag/Password: `flag_{I_tH1Nk_Th1$_jU$t_g0T_t0u6h3r_L3}`

The decryption process:

1. Extracted the 39 encrypted bytes from the password validation function
2. XORed each byte with the key `0xB9`
3. Converted the resulting bytes to ASCII characters

The flag follows a typical CTF format and reads as "flag_{I_tH1Nk_Th1;U t_g0T_t0u6h3r_L3}" - which seems to be saying "I think this just got tougher, Level 3" in leetspeak!

This confirms it's a Level 3 challenge as mentioned in the success message "You achieved Level 3!" that the program prints when the correct password is entered.

Example of Assisted Analysis

✓ Functions Renamed:

- `FUN_00401afb` → `main` - Much clearer this is the program entry point
- `FUN_004019bc` → `check_password` - Obviously the password validation function
- `FUN_004016ed` → `rc4_encrypt` - Clear this implements RC4 encryption
- `FUN_004122a0` → `memcmp_optimized` - AVX2-optimized memory comparison
- `FUN_004047b0` → `printf` - Standard C library function
- `FUN_00408e60` → `puts` - Standard C library function
- `thunk_FUN_00411cd0` → `memcmp_thunk` - Thunk wrapper

✓ Variables Renamed:

- `param_1`, `param_2` → `argc`, `argv` in `main`
- `param_1` → `password_input` in `check_password`
- `local_78` → `password_buffer` - Buffer holding password copy
- `local_48` → `target_bytes` - The encrypted target to match
- `iVar1` → `compare_result` - Result of memory comparison

✓ Added Comments:

- Explained the RC4 key endianness issue
- Documented the 46-byte length requirement
- Added function purpose descriptions

Manual Patching Instructions

Address 0x00401081:

- **Current bytes:** `48 8D 45 D0` (LEA RAX,[RBP + -0x30])
- **Change to:** `48 8D 45 A0` (LEA RAX,[RBP + -0x60])
- **Only change:** The last byte from `D0` to `A0`

Address 0x004010a0:

- **Current bytes:** `48 8D 45 D0` (LEA RAX,[RBP + -0x30])
- **Change to:** `48 8D 45 A0` (LEA RAX,[RBP + -0x60])
- **Only change:** The last byte from `D0` to `A0`

How to Apply the Patches in Ghidra:

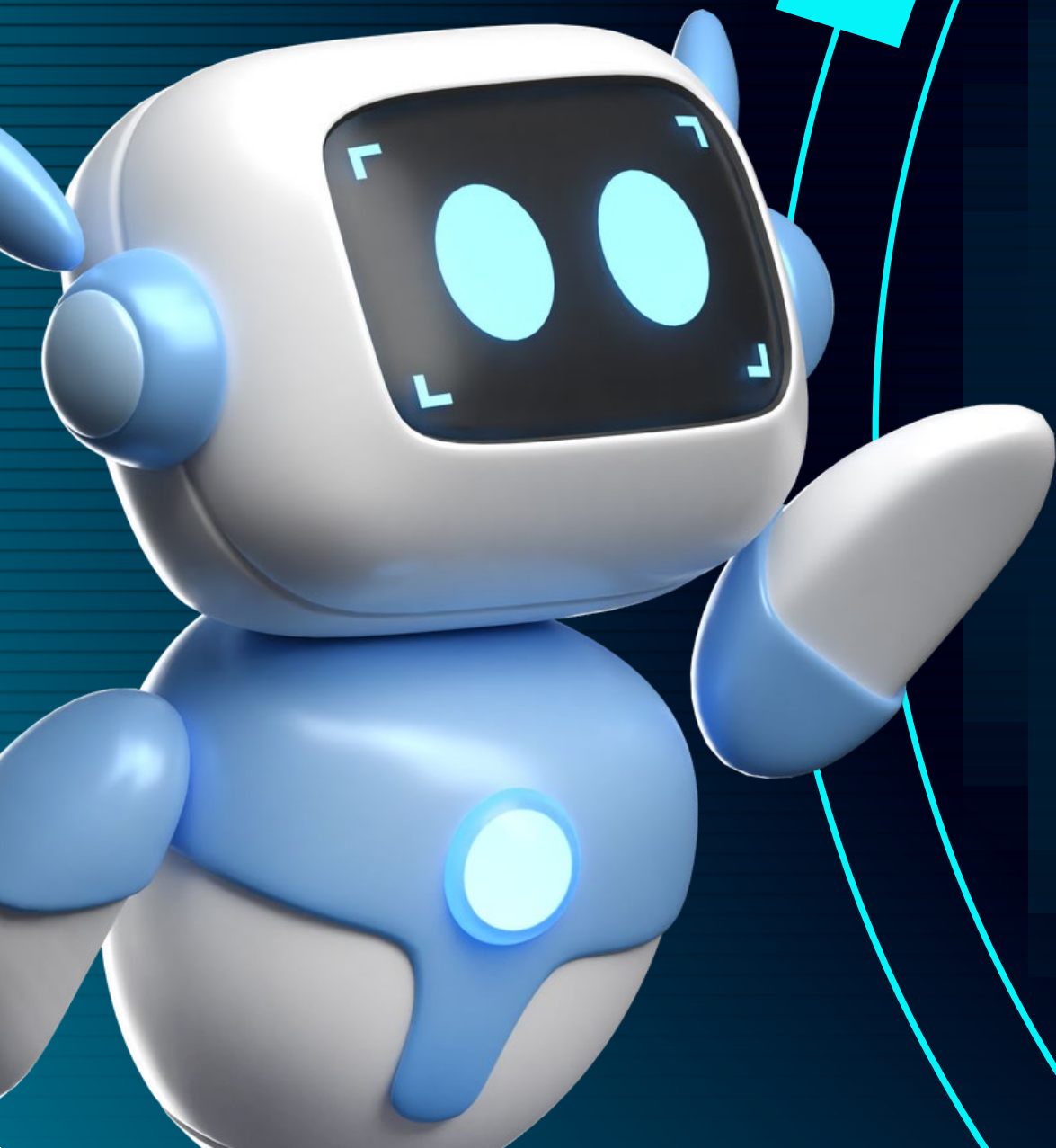
1. Go to address `0x00401081` in the Listing window
2. Right-click → Patch Instruction
3. Change the instruction text to: `LEA RAX,[RBP + -0x60]`
4. Click OK
5. Go to address `0x004010a0` in the Listing window
6. Right-click → Patch Instruction
7. Change the instruction text to: `LEA RAX,[RBP + -0x60]`
8. Click OK

Export the Patched Binary:

1. File → Export Program
2. Format: Select "Binary"
3. Save as: `level8_patched`
4. In terminal: `chmod +x level8_patched`

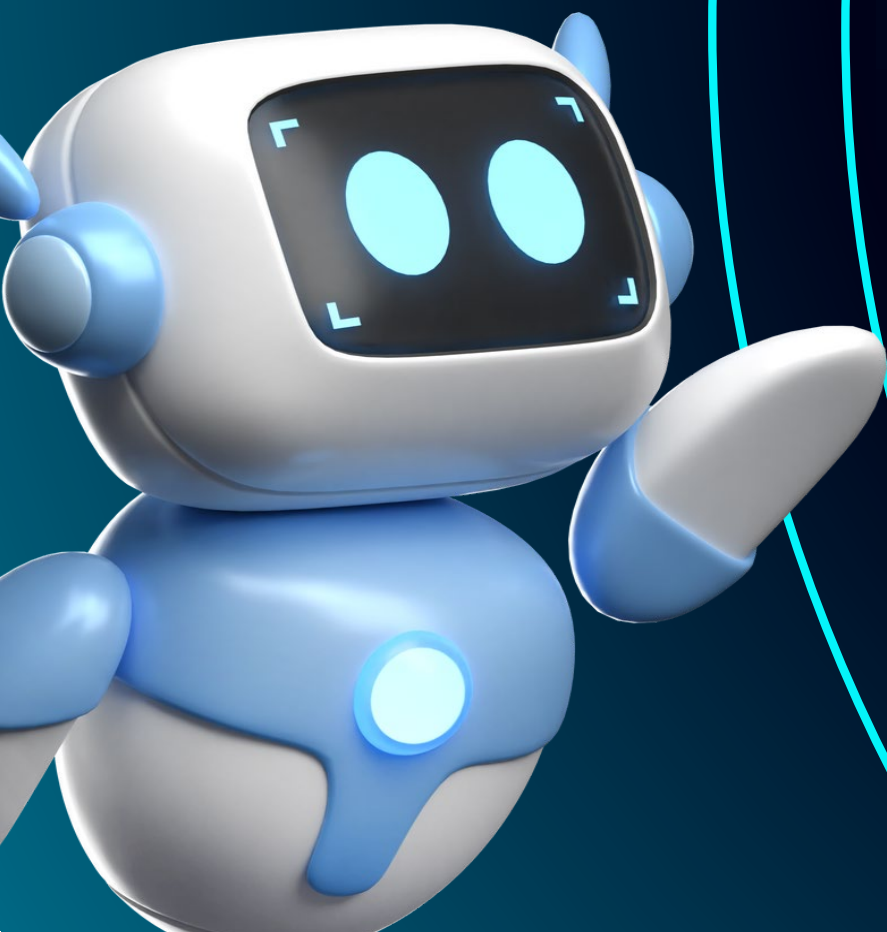
Capabilities

- LLM-in-the-loop analysis: Ask natural -language questions about functions/CFGs to get concise explanations and behavioral hypotheses.
- Pattern spotting: Automatically flags RE motifs like string decryption stubs, single -byte XOR, RC4-style loops, and checksum/CRC routines.
- Refactoring aids: Suggests descriptive function/variable names and inserts structured comments to streamline later analysis.
- Prompt → Script: Generates Ghidra Python/Java snippets for repetitive tasks (e.g., enumerate strings, rename by signature, batch decode).
- Navigation help: Summarizes cross-references, call graphs, and data-flow to highlight key paths (e.g., main, check_password).
- Workflow capture: Records prompt-response sessions for export as reproducible methodology artifacts.



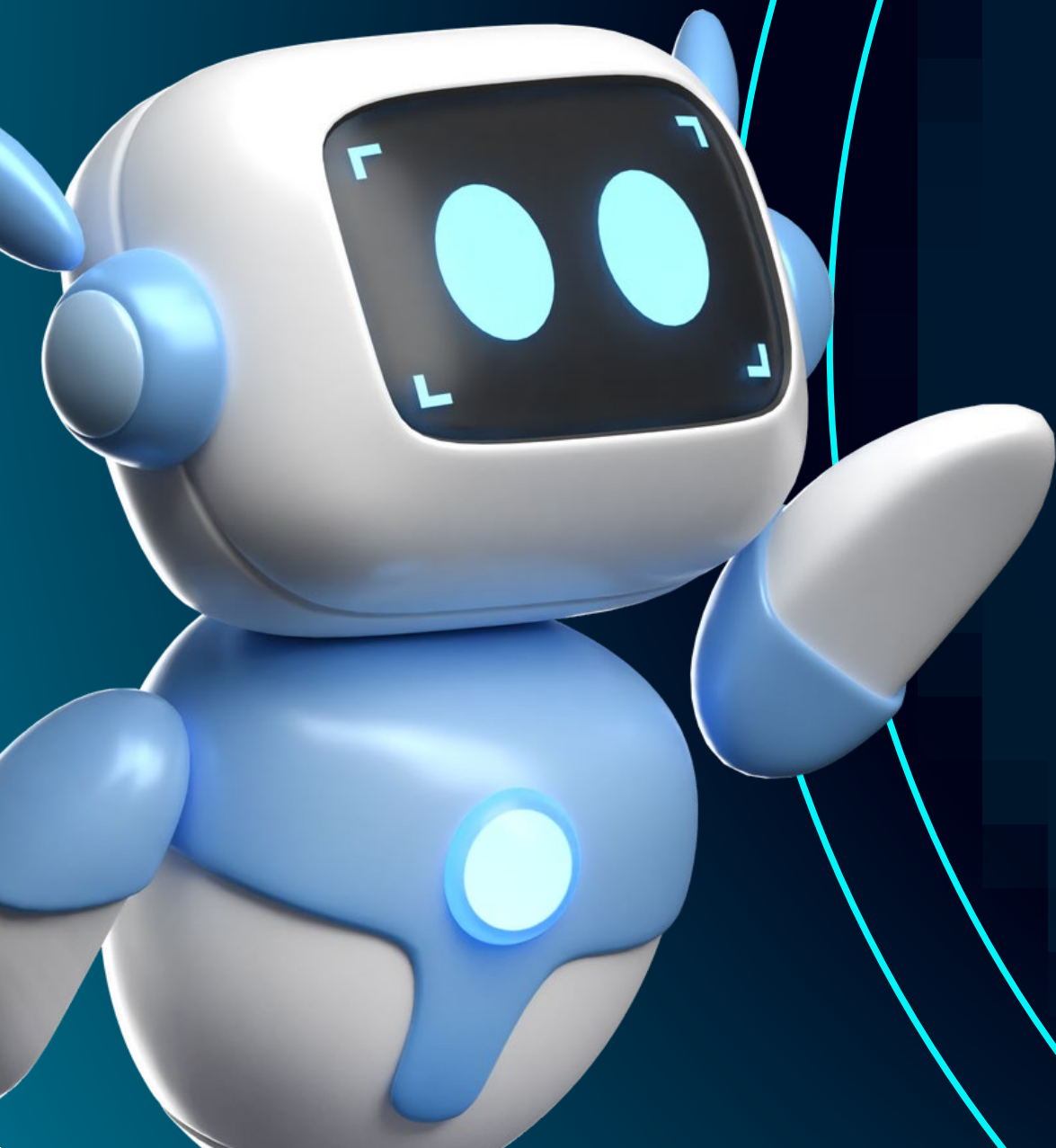
L i m i t a t i o n s

- Hallucinations: Plausible but false outputs (e.g., invented flag) → verify with real binary behavior.
- Prompt sensitivity: Small phrasing changes alter results; use disciplined templates.
- Static blind spots: No runtime state; must supply traces for anti-debug or packed code.
- Decompiler dependency: Heavy obfuscation degrades guidance accuracy.
- Context limits: Large binaries exceed token limits → analysis must be chunked.
- Reproducibility: Results vary across runs/models; save transcripts & use fixed versions.
- Latency & cost: Long prompts/code blobs slow iterations and raise expense.
- Data governance: Risk of data egress → enforce redaction, policy controls, audit trails.
- Version friction: Plugin, API, or Ghidra version mismatches can break integration.



Future Works

- Broaden the corpus: Expand beyond Escalate to real world malware, packed binaries, drivers, firmware, and APKs.
- Stronger evaluation: Conduct controlled user studies measuring time -to-solution, accuracy, edits, and cognitive load.
- Hallucination mitigation: Add confidence scores, self consistency checks, and verified tool steps with provenance tracking.
- Model upgrades: Fine -tune on RE -specific corpora; integrate RAG over symbol libraries, crypto patterns, and known TTPs.
- Toolchain expansion: Automate Ghidra scripting; add IDA/Binary Ninja parity and batch processing for large datasets.





Q & A ?

