Chinmay Dalal Snehashish Reddy Manda

SQuire (Trainee KNighter)
Group 1 - Project Proposal

Introduction

Large systems (e.g., Linux kernel) need robust static analysis to catch critical bugs early. Traditional static analyzers are expensive to design/maintain and cover limited, predefined bug patterns. Our goal is to leverage LLMs to synthesize scalable, explainable, and updatable checkers that encode real-world bug knowledge and run efficiently on large codebases.

Background

We draw inspiration from a paper called KNighter (SOSP '25). In the paper, the authors deploy LLM-synthesized static checkers, derived from bug-fix patches. The multi-stage flow looked as follows: pattern -> plan -> checker -> validate -> refine.

The impact:

- 92 kernel bugs found
- 30 CVEs
- These checkers were orthogonal to Smatch

One limitation and prospect of future work mentioned by the authors in the paper is that they've generalized these checkers to both complex and simple fixes, reducing its effectiveness in both domains. It would be beneficial for us to build on this and synthesize checkers for simple fixes only, advancing the work in this field. We need a method for determining which fixes are simple, so we plan to develop a confidence self-reporting system for the LLM.

The Benefits of an LLM-based Approach

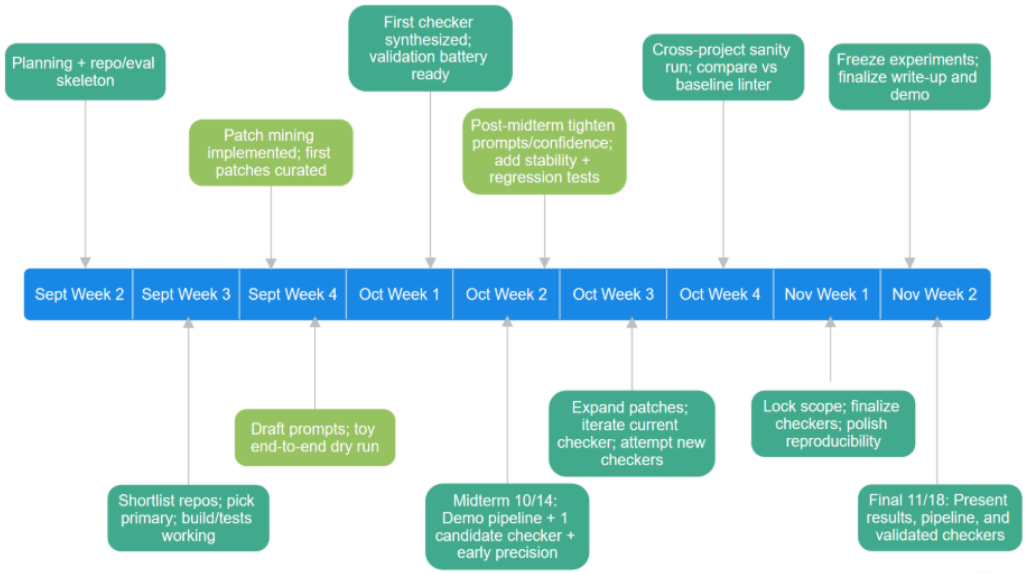
We cannot use current LLMs to check code directly, due to the low context limits and high compute cost. LLMs are also prone to hallucinations and inconsistent reasoning at scale. Thus, we aim to leverage LLMs not to scan code directly, but to synthesize specialized static checkers from historical bug-fix patches.

LLMs solve some particular problems that make them appealing over a human-only solution:

- Rapid coverage of new patterns: A working system, in theory, can process simple patterns significantly faster than a human could.
- Scale and iteration: Humans would have to mine commits, and then go through each bug and see if it can be implemented into a static checker. With LLMs, most of the work goes into prompt engineering and understanding deeply how static checkers work.
- Knowledge aggregation: Knowledge transfer in humans is a slow and delicate process. Knowledge transfer within LLMs is instant. The knowledge of multiple LLMs on different codebases can be aggregated quickly.

As such, we strongly believe that an LLM-based approach would benefit the goal of the systems research that we're exploring as part of COMP790-199.

Project Timeline



Team Setup

We have opted for a rotating leads setup approach for two reasons. To use our specific technical knowledge and strengths to get the project set up quickly and have it moving. And to ensure both of us receive full exposure to the project and achieve the primary goal of this course, which is to gain in-depth knowledge of static checkers and understand the system components behind the Linux kernel.

This is how we plan to rotate:

- Pre-milestone leads: Snehashish - LLM pipeline (prompts, synthesis, confidence); Chinmay - checker engineering (rule design, integration)
- Post-milestone swap: Snehashish takes checker refinement; Chinmay owns LLM prompt/tuning
- Shared throughout: patch mining, eval, triage, and presentation

Current Progress

- LLM Agent: Gemini 2.5-Flash [Chosen for high cost-to-performance ratio]
- Scope: 8–12 simple, intra-procedural bug classes (null-deref after alloc, missing error checks, memcpy/memset size, off-by-one bounds, missing refcount put, single-function double free, unchecked copy_from_user, IS_ERR/PTR_ERR misuses).
- Method: LLM extracts patterns from bug-fix patches → emits Coccinelle (Spatch) rules → validates on historical snapshots → refines; optional port of best rules to Smatch/CodeQL.
- Dataset: Linux kernel history; train ≤ v5.17, tune v5.18–v6.1, test v6.2–v6.8; patches mined via keyword heuristics and light clustering to avoid leakage.
- Metrics/targets: precision on top-50 findings per checker ≥ 60–75%; runtime per kernel tree ≤ 30 min; recover ≥ 20–30% of held-out historical fixes in targeted classes; confidence score correlates with correctness (Spearman ≥ 0.3).
- Baselines: Smatch where applicable; otherwise, a minimal hand-written Coccinelle script.
- Deliverables: code + generated checkers, mined patch dataset, evaluation table, 10–20 triaged true positives, brief disclosure plan.