KNighter: Transforming Static Analysis with LLM-Synthesized Checkers

B R I E F R E P O R T

KNighter implements this vision through a multi-stage synthesis pipeline that validates checker correctness against original patches and employs an automated refinement process to iteratively reduce false positives

Presenter: *Chen Miao*
HUST OS³Lab

目录

CONTENT

01 Background

Why we need LLM and Why we need static analysis

02 Design & Implement

The harder you work, the luckier you will be.

03 Evaluatie

The harder you work, the luckier you will be.

04 Limititation

The harder you work, the luckier you will be.

Why Static Analyzer

1. **Execution-Free Code Examination** – Unlike dynamic approaches, static analysis inspects source code without execution, making it ideal for analyzing hardware-dependent drivers, rarely executed paths, and hard-to-reproduce configurations in real environments.

2. **Coverage of Untested Paths** – Dynamic methods only test actual runtime paths, missing unexercised code. Static analysis, however, can uncover potential issues in all possible execution flows, including edge cases.

3. **Scalability & Feasibility for Large Systems** – While formal verification provides stronger guarantees, its high manual effort makes it impractical for large-scale systems (e.g., OS kernels). Static analysis offers a more scalable and automated solution without sacrificing broad coverage.

Why LLM Static Analyzer

1. **Overcoming Rule-Based Limitations** – Traditional static analyzers rely on predefined, rule-based, or formally modeled checks, requiring extensive domain expertise and manual effort to develop/maintain. They are narrowly tuned to known bug patterns, limiting detection of unforeseen defects and scalability for broader issues.

2. **Learning from Real-World Fixes** – LLMs excel at discovering novel bug patterns by directly learning from historical patch commits—a rich source of real fixes and contextual bug data. Unlike rule-based tools, they can adapt to new defect types without explicit rule-crafting.

3. **Handling Complex, Unstructured Code** – LLMs' ability to parse both natural language and code enables them to generalize across diverse coding patterns, addressing gaps where traditional tools struggle (e.g., implicit logic errors or cross-module issues).

Simple Introduce Design

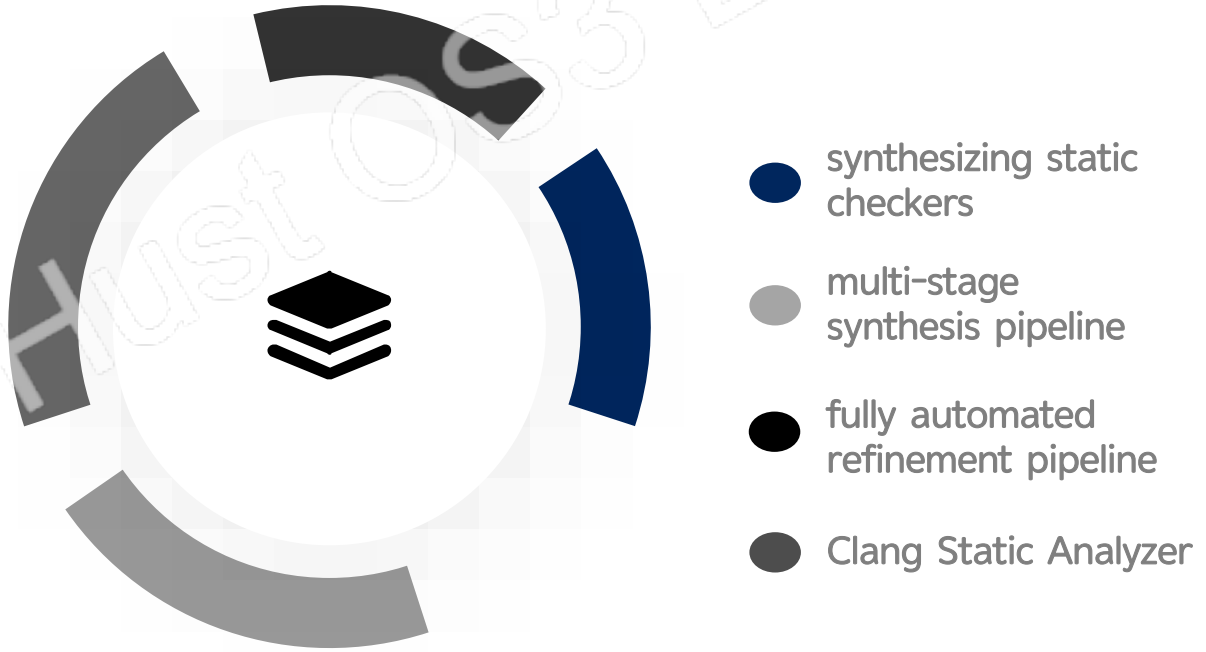
Can we scale and automate static analysis to handle both diverse bug patterns and enormous codebases?

1. Learning bug patterns from historical patches

2. Synthesizing specialized static checkers

3. Validating the effectiveness of the checkers

4.Using a multi-stage synthesis pipeline to split the task into smaller steps and generate high-quality checkers step by step.



5. Applying an automated refinement pipeline with bug report triage agents to reduce false positives and improve usability

Simple Example for devm_kzalloc

KNighter demonstrates its effectiveness through a case of null-pointer-dereference vulnerability. The issue stems from a missing null check after a devm_kzalloc call, which could cause system crashes. Although this pattern has recurred since 2017 and has been fixed in multiple patches, existing static analysis tools—including kernel-specific checkers like Smatch—have failed to detect it, largely due to the lack of domain knowledge that devm_kzalloc may return NULL on failure.

```
--- a/drivers/spi/spi-pci1xxxx.c
+++ b/drivers/spi/spi-pci1xxxx.c
@@ -275,6 +275,8 @@ static int pci1xxxx_spi_probe
  spi_bus->spi_int[iter] = devm_kzalloc(&pdev->dev, ...);
+ if (!spi_bus->spi_int[iter])
+   return -ENOMEM;
  spi_sub_ptr = spi_bus->spi_int[iter];
  spi_sub_ptr->spi_host = devm_spi_alloc_host(...)
```

(a) Patch for a Null-Pointer-Dereference bug. The pointer returned by devm_kzalloc should be checked.

```
int asoc_qcom_lpass_cpu_platform_probe(...)
{
  drvdata = devm_kzalloc(dev, ...);
+ if (!drvdata)
+   return -ENOMEM;   Patch
...
  drvdata->variant = variant;   ! Without NULL checking
```

(b) A new bug detected by KNighter with **CVE-2024-50103**.

As shown in the right figure, KNighter, by learning from historical patches, successfully identified a new instance of this vulnerability, which was later assigned CVE-2024-50103.

```
void checkPostCall(...) const {
  ...
  if (!ExprHasName(OriginExpr, "devm_kzalloc", C))
    return;
  State = State->set<PossibleNullPtrMap>(MR, false);
}

void checkBranchCondition(...) const {
  // Pattern 1: if (!ptr)
  if (const UnaryOperator *UO =
dyn_cast<UnaryOperator>(CondExpr)) {
    if (UO->getOpcode() == UO_LNot) {
      ...
      State = markRegionChecked(State, MR);
    }
  }
  // Pattern 2: if (ptr == NULL) or if (ptr != NULL)
  ...
}
void checkLocation(...) const {
  ...
  // Look up the region in the PossibleNullPtrMap.
  const bool *Checked = State->get<PossibleNullPtrMap>(MR);
  // If the region is recorded as unchecked, warn.
  if (Checked && *Checked == false)
    reportUncheckedDereference(MR, S, C);
}
void checkBind(...) const {
  ...
  // For pointer assignments, update the aliasing map.
  State = State->set<PtrAliasMap>(LHSReg, RHSReg);
  State = State->set<PtrAliasMap>(RHSReg, LHSReg);
 }
```

(c) A checker synthesized by KNighter for the patch in Fig. 2a.

checkPostCall: Triggered after function calls. It uses ExprHasName to detect devm_kzalloc calls and marks the returned memory as potentially null (unchecked) in PossibleNullPtrMap.

checkBranchCondition: Handles pointer-related conditions (e.g., if (!ptr)or if (ptr == NULL)), and updates the state via markRegionChecked to indicate the memory has been checked for null.

checkLocation: Triggered on memory access. If PossibleNullPtrMap indicates the memory is unchecked, it reports an unchecked dereference.

checkBind: Manages pointer assignments by updating PtrAliasMap to track aliasing between memory regions, ensuring correct handling of shared potentially null pointers.

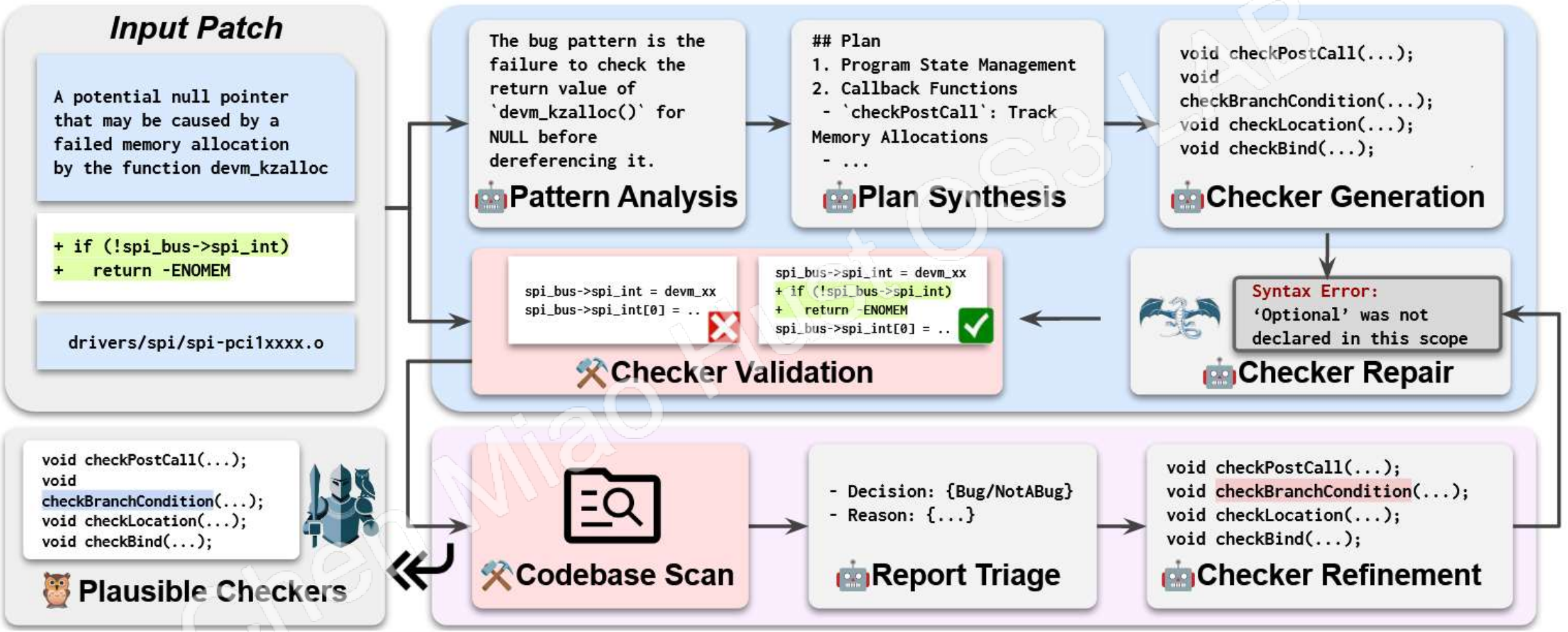


Figure 3. Overview of KNighter.

Checker Synthesis

1. **Bug Pattern Analysis**: Analyze patch submissions (including diff patches and possible developer comments) to identify potential defect patterns, which can be translated into static analysis rules.

2. **Plan Synthesis**: After the defect patterns are identified, KNighter generates a high-level implementation plan for building the static analyzer.

3. **Analyzer Implementation and Syntax Repair**: Provide comprehensive inputs, including refined defect patterns, structured implementation plans, predefined checker templates, and a list of utility functions, to maximize implementation accuracy.

4. **Validation**: Conduct differential analysis by applying the checker to both the buggy version (pre-patch code) and the patched version.

Checker Refinement

1. **Scanning and False Positive Issues**: Effective checkers are deployed to scan the entire codebase.

2. **Iterative Refinement Process**: The checker scans the entire codebase and generates potential defect reports.

3. **Conclusion and Output**: If the number of reports generated from the scan is manageable and the false positive rate is low, KNighter considers the checker to have reached a reasonable (plausible) level of practical usability.

Algorithm 1: Synthesize checkers with input patch.

```
Function GenChecker(patch):
    # Iterative checker generation and evaluation
    for i = 1 to maxIterations do
        # Stage 1: Bug Pattern Analysis
        pattern ← AnalyzePatch(patch)
        # Stage 2: Detection Plan Synthesis
        plan ← SynthesizePlan(patch, pattern)
        # Stage 3: Analyzer Implementation and Repair
        checker ← Implement(patch, pattern, plan)
        attempts ← 0
        while hasCompilationErrors(checker) AND attempts
          < maxAttempts do
            checker ← RepairChecker(checker)
            attempts ← attempts + 1
        if hasCompilationErrors(checker) then
            # Skip evaluation if checker still has errors
            Continue
        # Stage 4: Validation
        isValid ← ValidateChecker(checker, patch)
        if isValid then
            return checker
    return Null
```

Checker Synthesis
Bug Pattern Analyzer

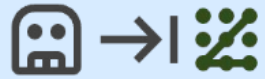
```
spi: mchp-pci1xxx: Fix a possible null pointer dereference in
pci1xxx_spi_probe

In function pci1xxx_spi_probe, there is a potential null
pointer that may be caused by a failed memory allocation
by the function devm_kzalloc. Hence, a null pointer check
needs to be added to prevent null pointer dereferencing
later in the code.
To fix this issue, spi_bus->spi_int[iter] should be checked.
The memory allocated by devm_kzalloc will be automatically
released, so just directly return -ENOMEM.
```

Figure 4. Patch commit message.

```
# Instruction
You will be provided with a patch in Linux kernel.
Please analyze the patch and find out the **bug pattern** in
this patch.
A **bug pattern** is the root cause of this bug, meaning that
programs with this pattern will have a great possibility of
having the same bug.
Note that the bug pattern should be specific and accurate,
which can be used to identify the buggy code provided in the
patch.
# Examples
...
# Target Patch
{{input_patch}}
```

Commit message
Buggy code
Diff patch



(a) Prompt template for bug pattern analysis

```
# Instruction
Please organize a elaborate plan to help to write a CSA
checker to detect such **bug pattern**.
# Utility Functions
...
# Examples
...
# Target Patch
{{input_patch}}
# Target Pattern
{{input_pattern}}
```

(b) Prompt template for plan synthesis.

To synthesize the implementation plan for the checker, we have designed an LLM-based agent, whose prompt template is shown in Figure 5b. **This agent takes the previously summarized bug pattern as input.**

Checker Synthesis
Plan Synthesis

Once the bug pattern is identified, KNighter **generates a high-level plan for implementing the static analyzer**. This plan serves two key purposes:

first, **it provides structured guidance** to large language models (LLMs) during implementation, preventing confusion and promoting efficient execution;

second, it **enhances the debugging of the entire pipeline** by making the LLMs' reasoning process transparent and traceable.

```
{{Customize Program States}} // If necessary
namespace {
class NewChecker : public Checker<{{Callback Functions}}> {
   mutable std::unique_ptr<BugType> BT;
   public:
      NewChecker() : BT(new BugType(this, "{{Bug desc}}")) {}
      {{Declaration of Callback Functions}}
   private:
      {{Declaration of Self-Defined Functions}}
};
 {{Self-Defined Functions (should be complete and runnable)}}
}
```

Figure 6. Pre-defined checker template for CSA.

After identifying the bug pattern and creating the plan, we use an LLM-based agent to implement the corresponding checker. To improve accuracy, we provide the agent with key inputs: **the refined bug pattern**, **a structured implementation plan**, and **a predefined checker template** (Figure 6) that standardizes the structure and reduces errors. We also supply a list of utility functions to aid implementation.

Checker Synthesis Analyzer Implementation

The generated checker may have compilation errors, such as incorrect static analysis API usage or wrong variable types. To address these, **we employ a specialized debugging agent.** Inspired by program repair research, **it automatically analyzes compiler errors and applies fixes**, resolving syntax issues caused by LLM hallucinations. This automated process ensures the final checker is syntactically correct and compilable.

Checker Synthesis Validation & Refinement

After the checkers are synthesized, each valid checker is used to scan the entire system. However, **the initial validation does not prevent potential false positives when analyzing a broader codebase** — that is, correctly written code may be incorrectly flagged as problematic.

Automating this refinement faces two key challenges:
1. Bug reports are **lengthy** and contain too **much context** to process efficiently;
2. Fixing checker logic based on false positives requires **complex analysis and precision.**

```
int sh_pfc_register_pinctrl(struct sh_pfc *pfc) {
        struct sh_pfc_pinctrl *pmx;
        int ret;
        pmx = devm_kzalloc(pfc->dev, sizeof(*pmx), GFP_KERNEL);
        if (unlikely(!pmx))                    FP by triage agent
                return -ENOMEM;
        pmx->pfc = pfc;                        Reported by checker
...
```

Figure 7. A report labeled as FP by our triage agent.

Distill the generated bug reports down to their essential components:
1. **The "relevant lines" highlighted by the static analyzer** (e.g., CSA);
2. **The corresponding trace path**.

Triage Agent: Responsible for **classifying each distilled report**, focusing specifically on **whether the report aligns with the target bug pattern** (rather than making a general judgment on whether the code is "correct"). If the triage agent identifies a report **as a false positive** (as shown in Figure 7), a dedicated **Refinement Agent** takes over to handle the case.

A refined checker is accepted **only if**:
1. **It no longer flags previously identified false positives**;
2. **It can still correctly distinguish buggy code from patched code**.

Implementation

1. **Input**: Historical Bug Patches

We use **10 bug types** (e.g., **null dereference**, **memory leak**). Only patches agreed upon by two reviewers are included.

2. **Few-Shot Examples**

We manually prepare **3 example cases** (e.g., **null deref**, **use-after-init**, **double-free**). *Done once, can be reused.*

3. **Helper Functions**

Provide 9 reusable functions to simplify writing checkers(e.g., **getMemRegionFromExpr**).

4. **Definitions**

(a) **Valid Checker**: Can find bugs in buggy code, but not in fixed code. Fewer than T_valid=50 reports on fixed code.

(b) **Trusted Checker**: Not just valid, but also useful in practice; Few total reports(T_plausible=20), or Low false positive rate in testing.

$$N_{buggy} > N_{patched} and N_{patched} < T_{valid}, T_{valid} = 50$$

5. **Checker Optimization Process**

Goal: *Automatically reduce the false positive rate of a valid checker.*

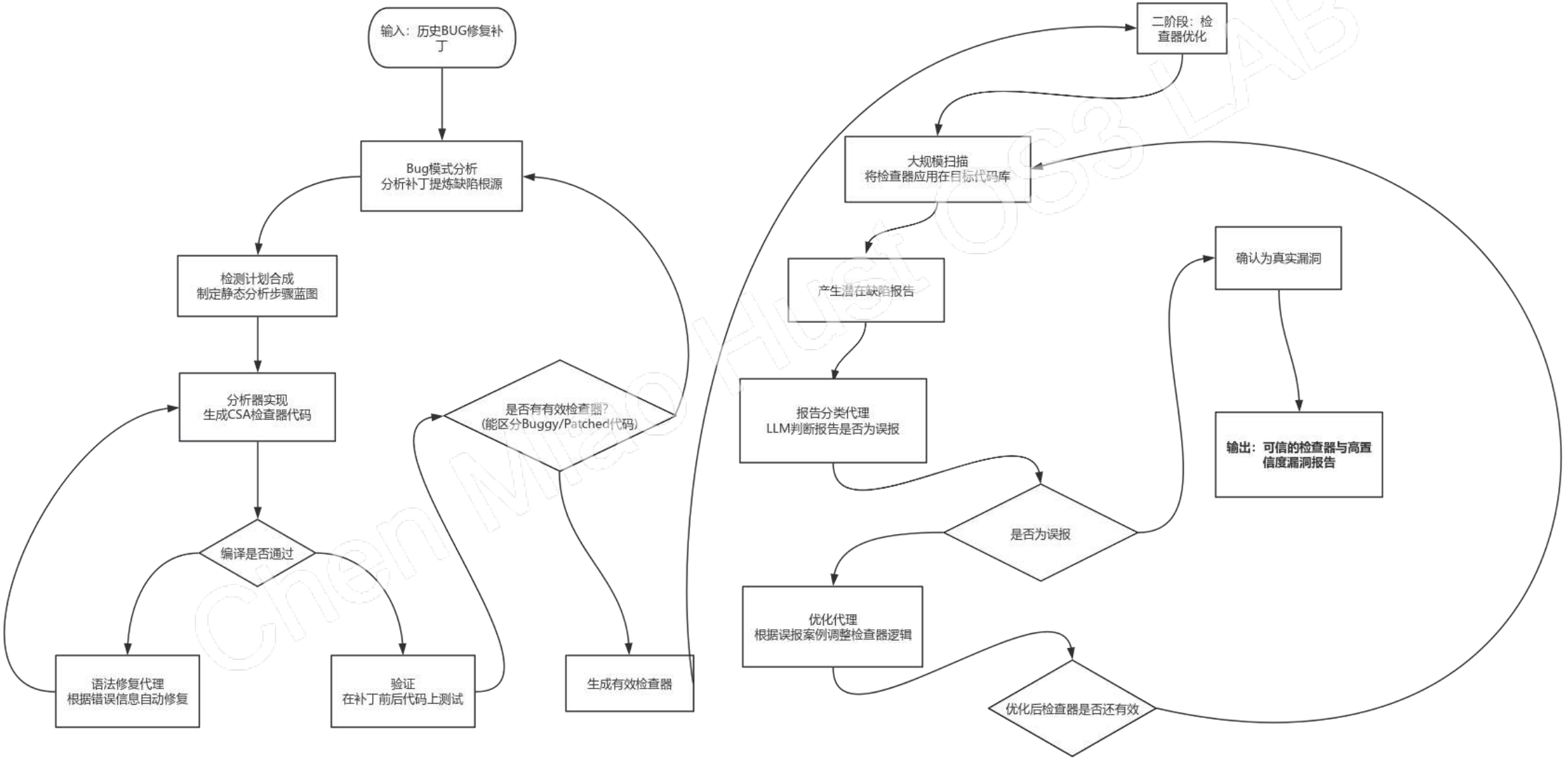
Steps:

1. **Large-Scale Scan**: Run the valid checker across the full Linux kernel codebase.
2. **Sampling & Classification**: Due to cost, randomly select up to 5 reports, and use an LLM-driven classification agent to judge whether each is a real bug or a false positive.
3. **Iterative Refinement**: If false positives are found, an optimization agent automatically modifies the checker logic based on those cases. This process is repeated up to 3 times.
4. **Success Criteria**: The checker is labeled as trusted if, after optimization:

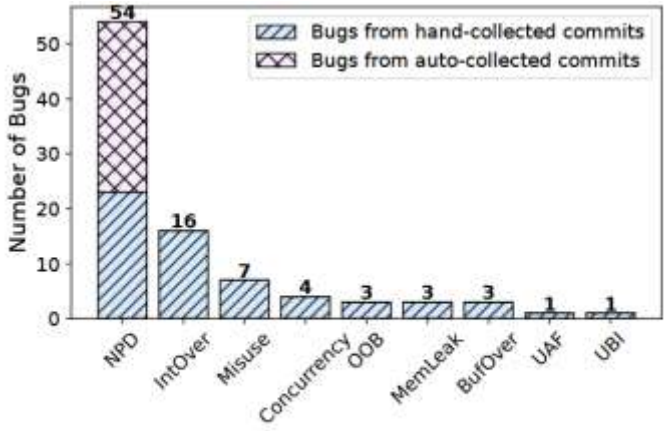
(a) **The total number of reports is small, or**

(b) **The false positive rate in samples is very low.**

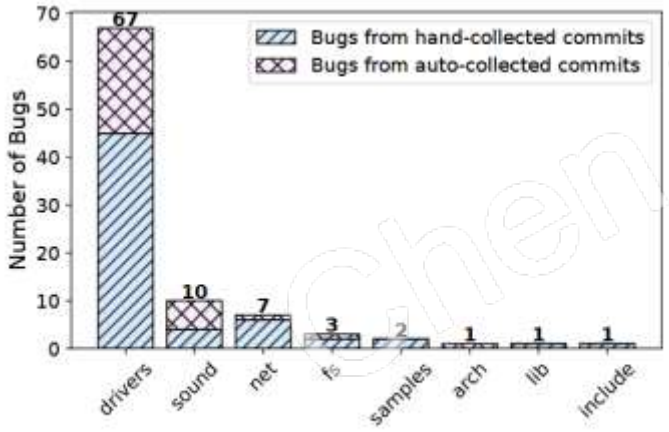
Implementation



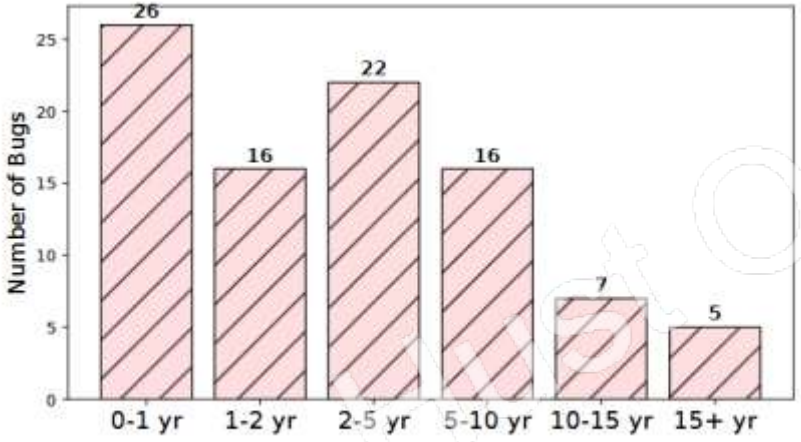
Evaluation



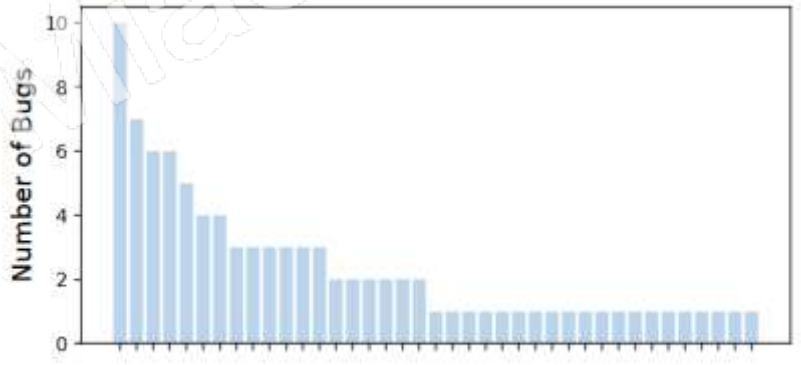
(a) Number of bugs in each type.



(b) Number of bugs in each subsystem.



(c) Number of bugs with different lifetimes.



(d) Number of bugs detected by each commit.

KNighter shows strong capability in automatically synthesizing complex and accurate static checkers from historical bug patches.

On 61 patches (10 bug types), KNighter synthesized "effective" checkers for **39 (61%)**—**avg. 125.7 lines**, mostly(**37/39**) with path-sensitive logic. Optimizing **13 high-false-positive checkers improved 11 (84.6%)**, yielding **37 "trustworthy" ones with 32.2% false positives (61/90 real vulnerabilities)**. This led to **92 new**, long-latent Linux kernel flaws (**avg. 4.3-year latency**): **77 confirmed, 57 fixed, 30 with CVEs.**

Limitations

1. **Complex Program Semantics & State Reasoning**
   (a) **State Machine Bugs** (e.g., Use-After-Free): Hard to track object lifecycles (allocation → use → free → reuse) across functions.
   (b) **Concurrency Bugs** (e.g., Race Conditions): Difficult to model thread interactions, locks, and timing issues.
2. **Static Analysis Limitations**
   (a) **Path Explosion**: Too many code paths slow down analysis.
   (b) **Value Uncertainty**: Can't predict runtime values (e.g., buffer sizes).
   (c) **Environment Dependency**: External factors (hardware/drivers) are hard to model.
3. **Patch Quality Dependence**
   (a) **"Garbage in, garbage out"**: Low-quality patches teach flawed patterns.
   (b) **Expressiveness**: Some fixes (e.g., architectural changes) can't be rule-based.
4. **"Hindsight Bias"**
   Only detects known bug types (seen in history), not new/zero-day vulnerabilities.

Thanks for View

Presenter: *Chen Miao*
HUST OS³Lab

KNighter: Transforming Static Analysis with LLM-Synthesized Checkers