Presenter: Dimas Parikesit (vqx2dc)

KNighter: Transforming Static Analysis with LLM-Synthesized Checkers

Chenyuan Yang
University of Illinois at Urbana-Champaign
USA
cy54@illinois.edu

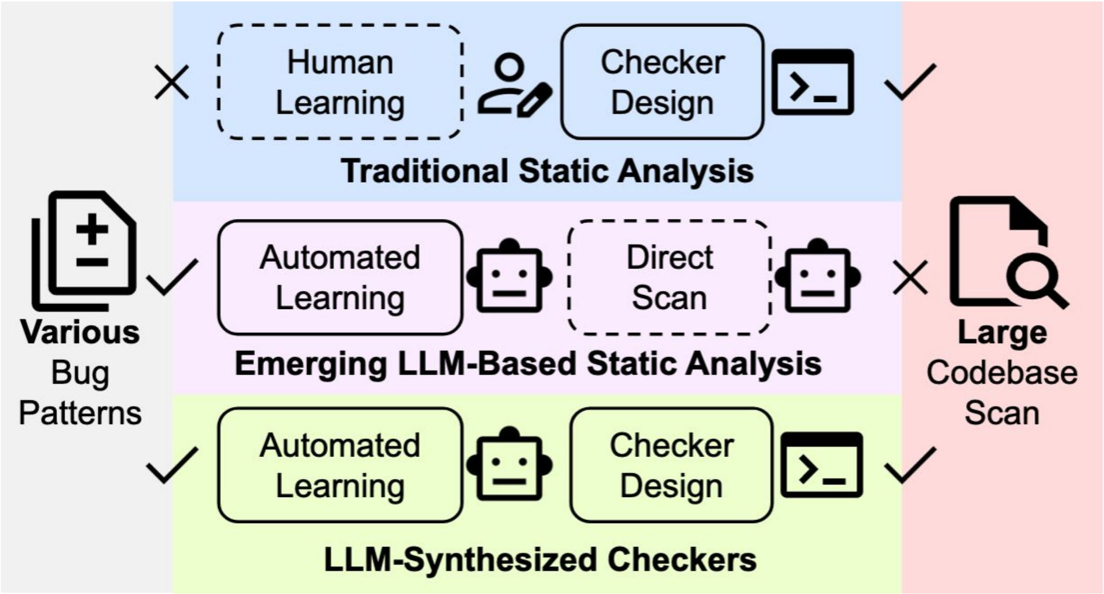
Zijie Zhao
University of Illinois at Urbana-Champaign
USA
zijie4@illinois.edu

Zichen Xie
Zhejiang University
China
xiezichen@zju.edu.cn

Haoyu Li
Shanghai Jiao Tong University
China
learjet@sjtu.edu.cn

Lingming Zhang
University of Illinois at Urbana-Champaign
USA
lingming@illinois.edu

Current state of Static Analysis Checker



Traditional Static Analysis

- Clang Static Analyzer (CSA) has a set of handmade checkers

1. Available Checkers

The analyzer performs checks that are categorized into families or “checkers”.

The default set of checkers covers a variety of checks targeted at finding security and API usage b
Checkers checkers list below.

In addition to these, the analyzer contains a number of **Experimental Checkers** (aka *alpha* checkers).
off by default. They may crash or emit a higher number of false positives.

The **debug** package contains checkers for analyzer developers for debugging purposes.

Table of Contents


```
// C
void test(int *p) {
  if (!p)
    *p = 0; // warn
}
```

https://clang.llvm.org/docs/analyzer/checkers.html

Traditional Static Analysis

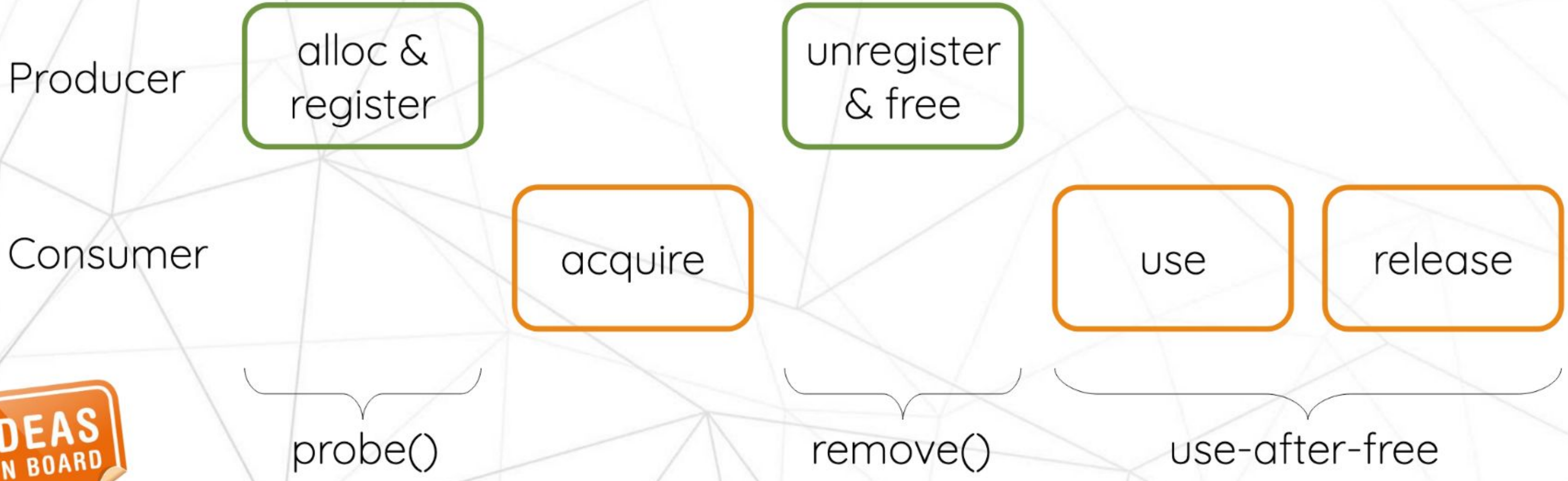
- Clang Static Analyzer (CSA) has a set of handmade checkers
- CSA can check the entire program
- But it lacks domain knowledge (such as which functions may return null?)

Traditional Static Analysis

- Clang Static Analyzer (CSA) has a set of handmade checkers
- CSA can check the entire program
- But it lacks domain knowledge (such as which functions may return null?)

Use-after-free

A typical conversion to devm_* helpers doesn't conceptually **introduce** bugs, because **the bugs have been there all along**. If a resource is **freed** at remove time (a.k.a. detach, a.k.a. disconnect, a.k.a. unbind), a use-after-free may occur as consumers may hold references.



Traditional Static Analysis

- Clang Static Analyzer (CSA) has a set of handmade checkers
- CSA can check the entire program
- But it lacks domain knowledge (such as which functions may return null?)

```
--- 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.

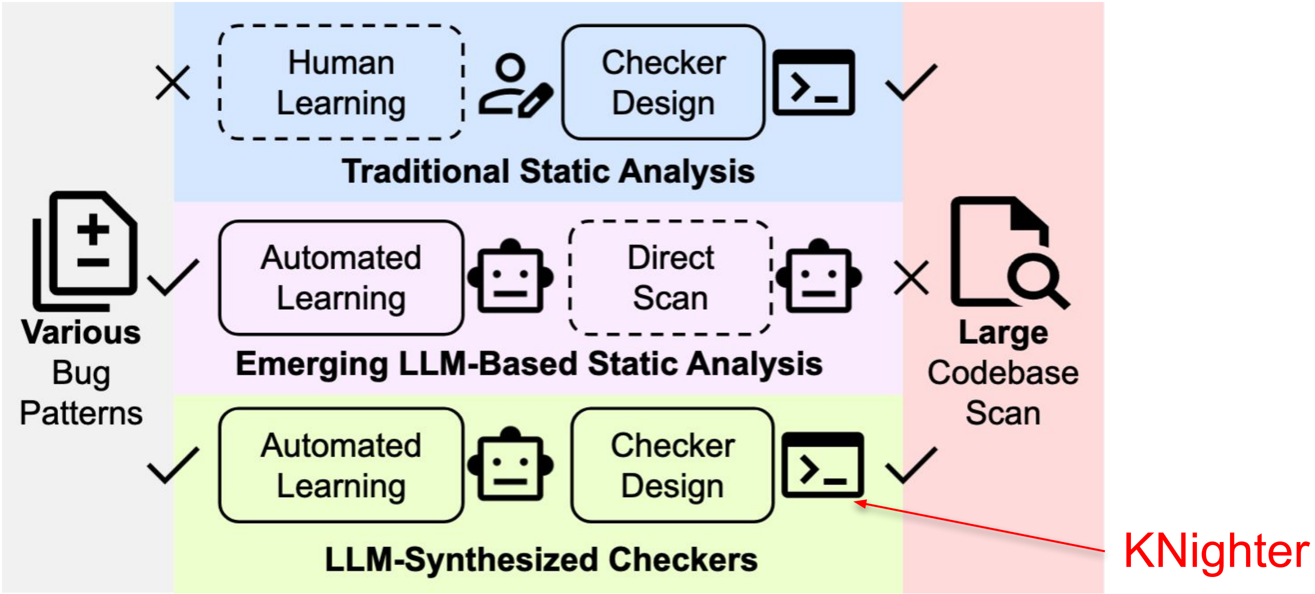
```
int asoc_qcom_lpass_cpu_platform_probe(...)
{
  drvdata = devm_kzalloc(dev, ...);
+ if (!drvdata)
+   return -ENOMEM;   Patch
...
  drvdata->variant = variant; ! Without NULL checking
```

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

LLM-Based Automated Scan

- LLM can learn domain specific knowledge
  (such as devm_kzalloc may return null)
- But directly using LLM to **scan the entire codebase is impossible** because of limited context window

Current state of Static Analysis



How to combine ease-of-use of LLM with the scalability of traditional static analysis?

KNighter

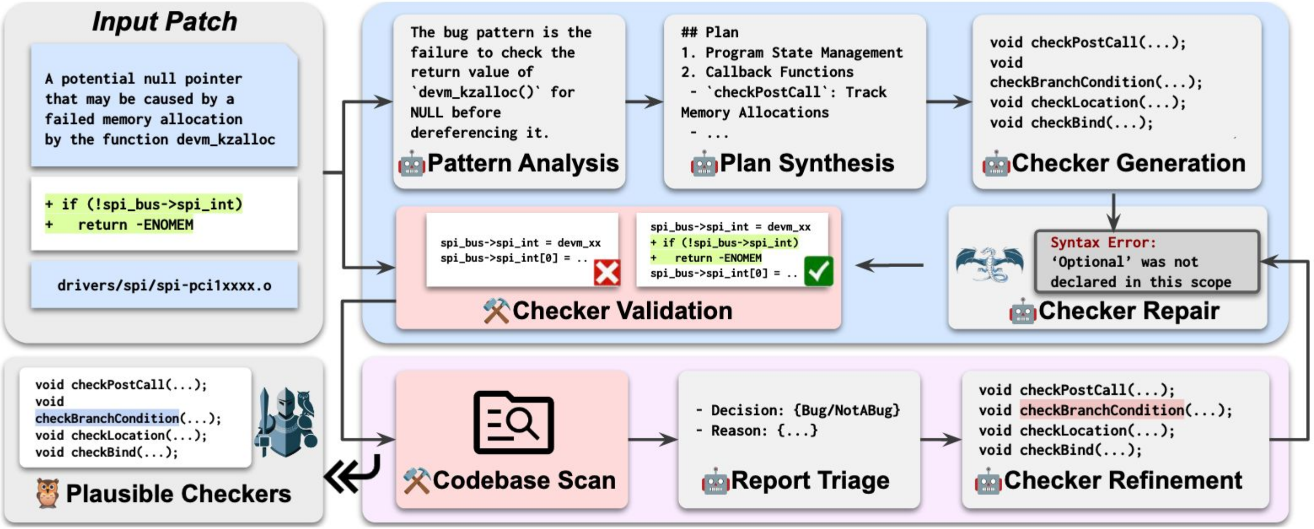
KNighter

Automatically synthesize CSA checker from a patch commit

KNighter

Automatically synthesize CSA checker from a patch commit

Synthesis



Refinement

Checker Synthesis

(1) Pattern Analysis

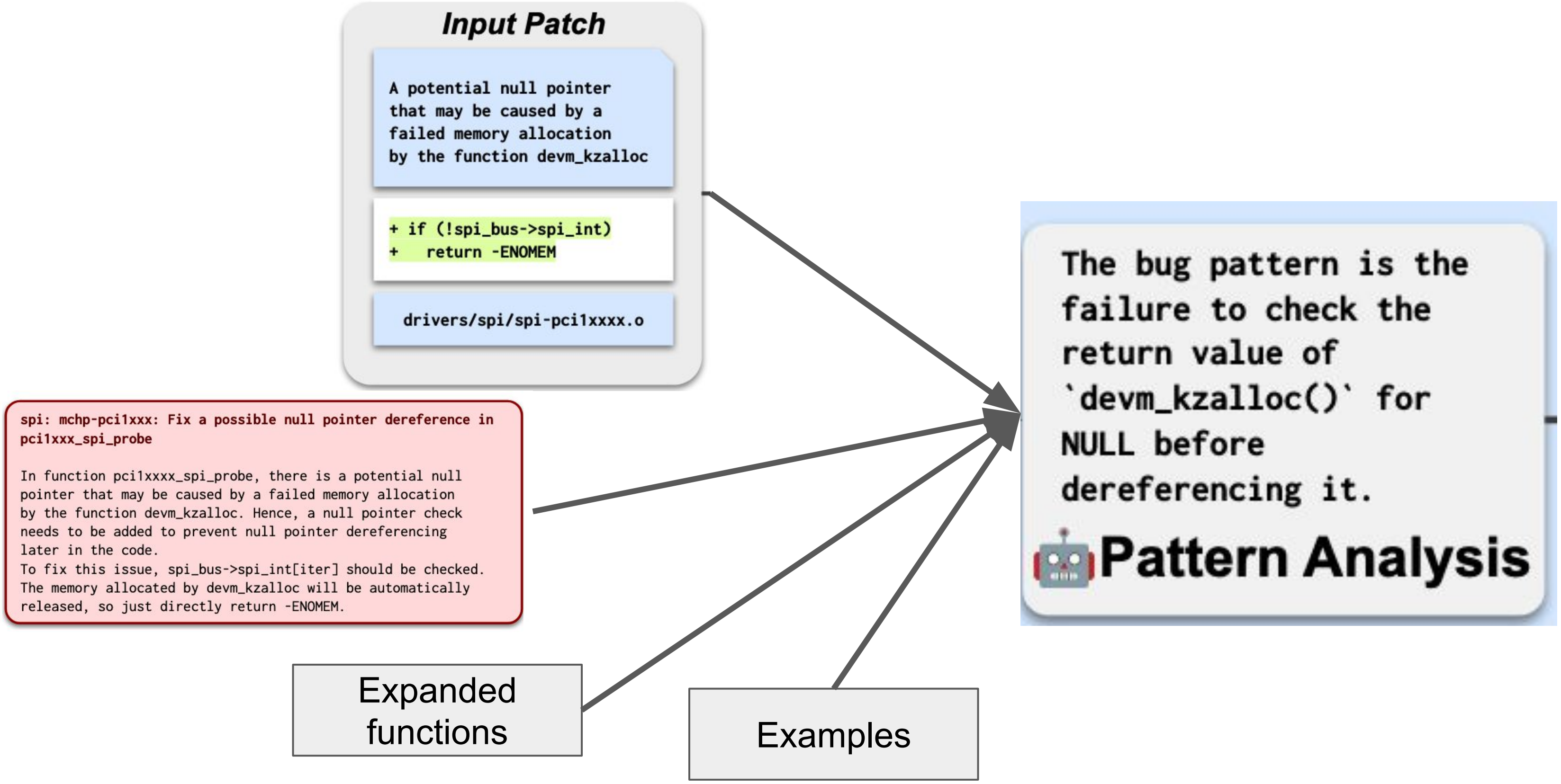
- Extract **targeted** bug patterns derived from the patch context.
- A bug pattern is the **root cause** of this bug, meaning that programs with this pattern will have a great possibility of having the same bug.

(1) Pattern Analysis

- These patterns are pre-determined

Bug Type
NPD
Integer-Overflow
Out-of-Bound
Buffer-Overflow
Memory-Leak
Use-After-Free
Double-Free
UBI
Concurrency
Misuse

(1) Pattern Analysis



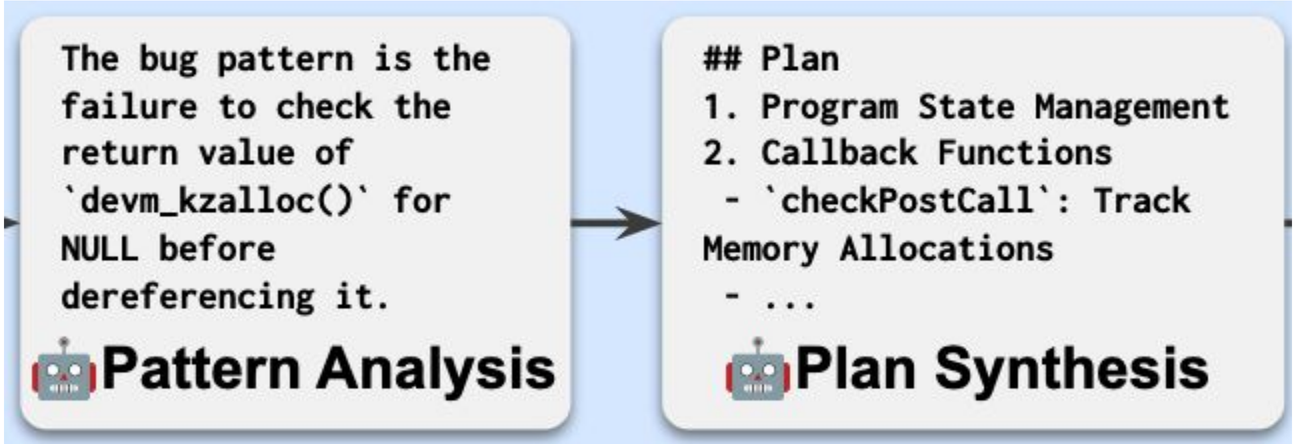
(1) Pattern Analysis

```
# 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}}
```

(a) Prompt template for bug pattern analysis

(2) Plan Synthesis

- Generates a high-level plan for implementing the static analysis checker
- Goal:
  - Provides **structured guidance** to the LLMs during the actual checker generation to prevent confusion
  - **Facilitates debugging** of the entire pipeline by making the LLMs' reasoning process transparent and traceable



(2) Plan Synthesis: Prompt

```
# 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.

(3) Checker Generation

- Generate CSA checker based on the plan

```
## Plan
1. Program State Management
2. Callback Functions
  - `checkPostCall`: Track
Memory Allocations
  - ...
```

Plan Synthesis

```
{{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.

```
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.

(4) Checker Repair

- Any LLM-generated code might be broken
- Use an **LLM debugging agent** to fix **syntax error** by automatically processes compiler error messages and applies necessary fixes

(4) Checker Repair

- Any LLM-generated code might be broken
- Use an **LLM debugging agent** to fix **syntax error** by automatically processes compiler error messages and applies necessary fixes

Instruction

The following checker fails to compile, and your task is to resolve the compilation error based on the provided error messages.

Here are some potential ways to fix the issue:

1. Use the correct API: The current API may not exist, or the class has no such member. Replace it with an appropriate one.
2. Use correct arguments: Ensure the arguments passed to the API have the correct types and the correct number.
3. Change the variable types: Adjust the types of some variables based on the error messages.
4. Be careful if you want to include a header file. Please make sure the header file exists. For instance "fatal error: clang/StaticAnalyzer/Core/PathDiagnostic.h: No such file or directory".

The version of Clang environment is Clang-18. You should consider the API compatibility.

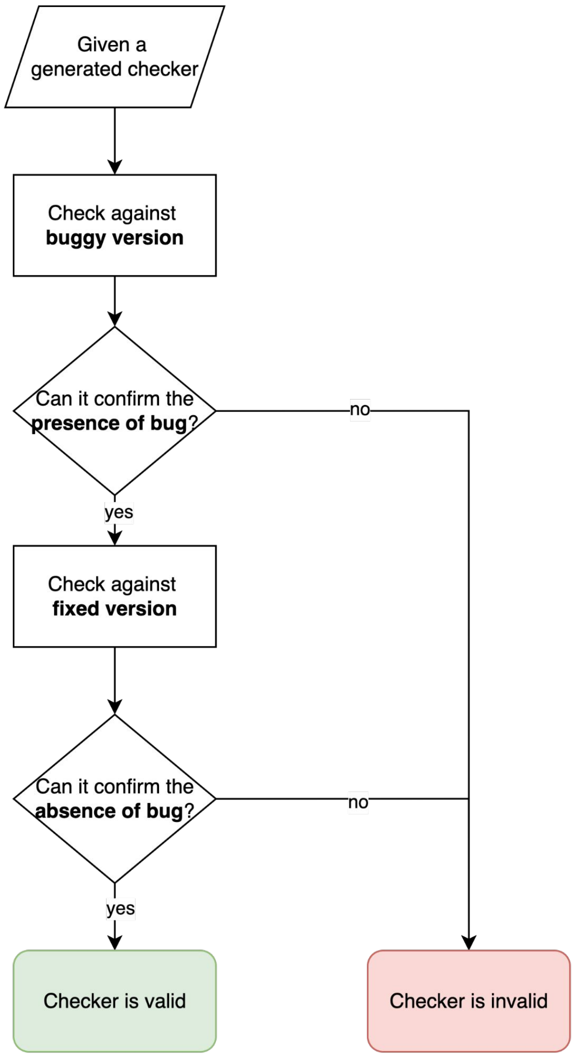
Please only repair the failed parts and keep the original semantics. Please return the whole checker code after fixing the compilation error.

(5) Checker Validation

- Mitigate LLM inaccuracies

(5) Checker Validation

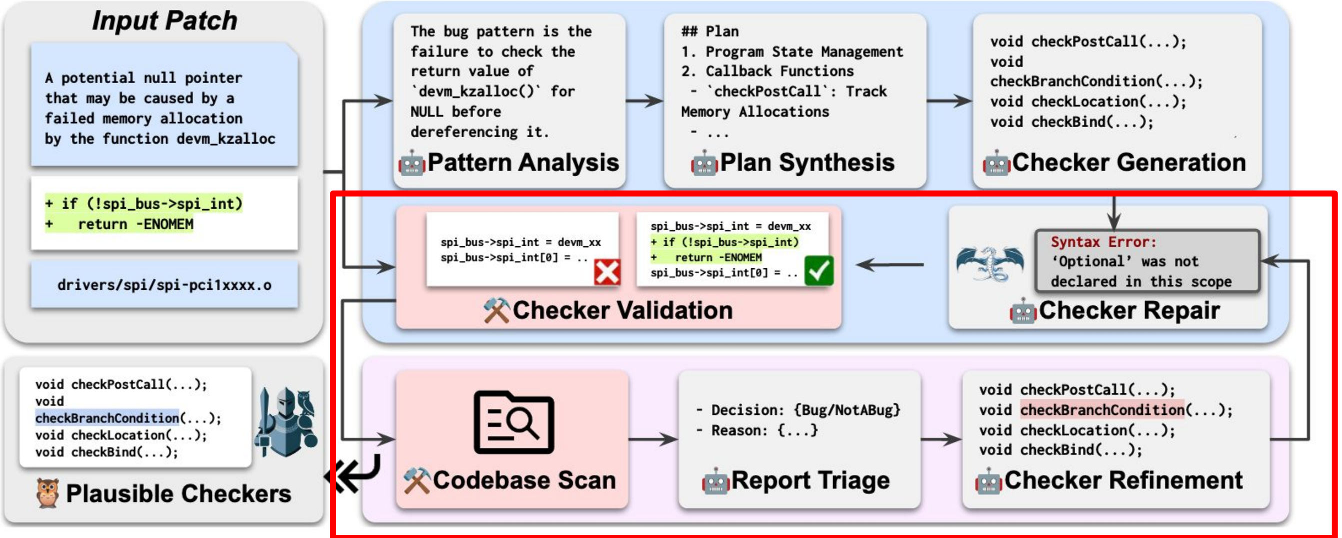
- Mitigate LLM inaccuracies
- Scoped to only the files modified by the patch



Checker Refinement

How to prevent potential false positive?

- Run checkers on the entire program
- For all reported potential bugs
  - Evaluate the generated bug report to identify the false positives
  - Use the identified false positives back to refine the checker



How to identify false positive?

- Validate the bug pattern

How to identify false positive?

- Validate the bug pattern
- Validate against pre/post patch behavior

How to identify false positive?

- Validate the bug pattern
- Validate against pre/post patch behavior
- Evaluate the feasibility of false positive patterns

How to identify false positive?

- Validate the bug pattern
- Validate against pre/post patch behavior
- Evaluate the feasibility of false positive patterns
  - Bounds

- **Numeric / bounds feasibility** (if applicable):
  - Infer tight **min/max** ranges for all involved variables from types, prior checks, and loop bounds.
  - Show whether overflow/underflow or OOB is actually triggerable (compute the smallest/largest values that violate constraints).

How to identify false positive?

- Validate the bug pattern
- Validate against pre/post patch behavior
- Evaluate the feasibility of false positive patterns
  - Bounds
  - Null-pointer dereference

- **Null-pointer dereference feasibility** (if applicable):

  i. **Identify the pointer source** and return convention of the producing function(s) in this path (e.g., returns **NULL**, **ERR_PTR**, negative error code via cast, or never-null).

  ii. **Check real-world feasibility in this specific driver/socket/filesystem/etc.**:
    - Enumerate concrete conditions under which the producer can return **NULL/ERR_PTR** here (e.g., missing DT/ACPI property, absent PCI device/function, probe ordering, hotplug/race, Kconfig options, chip revision/quirks).
    - Verify whether those conditions can occur given the driver's init/probe sequence and the kernel helpers used.

  iii. **Lifetime & concurrency**: consider teardown paths, RCU usage, refcounting (`get/put`), and whether the pointer can become invalid/NULL across yields or callbacks.

  iv. If the producer is provably non-NULL in this context (by spec or preceding checks), classify as **false positive**.

An example of false positive

- “unlikely” is a hint for the branch predictor

```
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))
                return -ENOMEM;
        pmx->pfc = pfc;
...
```

FP by triage agent

Reported by checker

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

What do we do with the false positives?

- Refine the checkers based on the identified false positives using an **LLM agent**

```
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.

add 3rd pattern here to check if(unlikely(!ptr))

What do we do with the false positives?

- Refine the checkers based on the identified false positives
- Continue to refine until
  - it **no longer generates warnings** for the previously identified false positive cases
  - it **maintains its validity** by correctly differentiating between the original buggy and patched code versions

Evaluation

Evaluation Setup

RQ-1. Can KNighter generate high-quality checkers?
RQ-2. Can the checkers generated by KNighter find real-world kernel bugs?
RQ-3. Are the capabilities of KNighter orthogonal to the human-written checkers?
RQ-4. Are all the key components in KNighter effective?

RQ1: Checkers Quality

- Evaluated on **61 hand-picked** commits

			Valid		
Bug Type	Total	Invalid	Direct	Refined	Fail
NPD	6	1	2	2	1
Integer-Overflow	7	3	1	3	0
Out-of-Bound	6	2	4	0	0
Buffer-Overflow	5	3	2	0	0
Memory-Leak	5	2	3	0	0
Use-After-Free	7	4	2	1	0
Double-Free	8	1	5	1	1
UBI	5	1	1	3	0
Concurrency	5	2	3	0	0
Misuse	7	3	3	1	0
Total	61	22	26	11	2

RQ1: Checkers Quality

- 22 invalid checkers due to
  - 2 inaccurate bug patterns
  - 7 inaccurate plan
  - 13 inaccurate implementations
    - Static analysis struggles with establishing buffer bounds during compilation

RQ1: False Positive

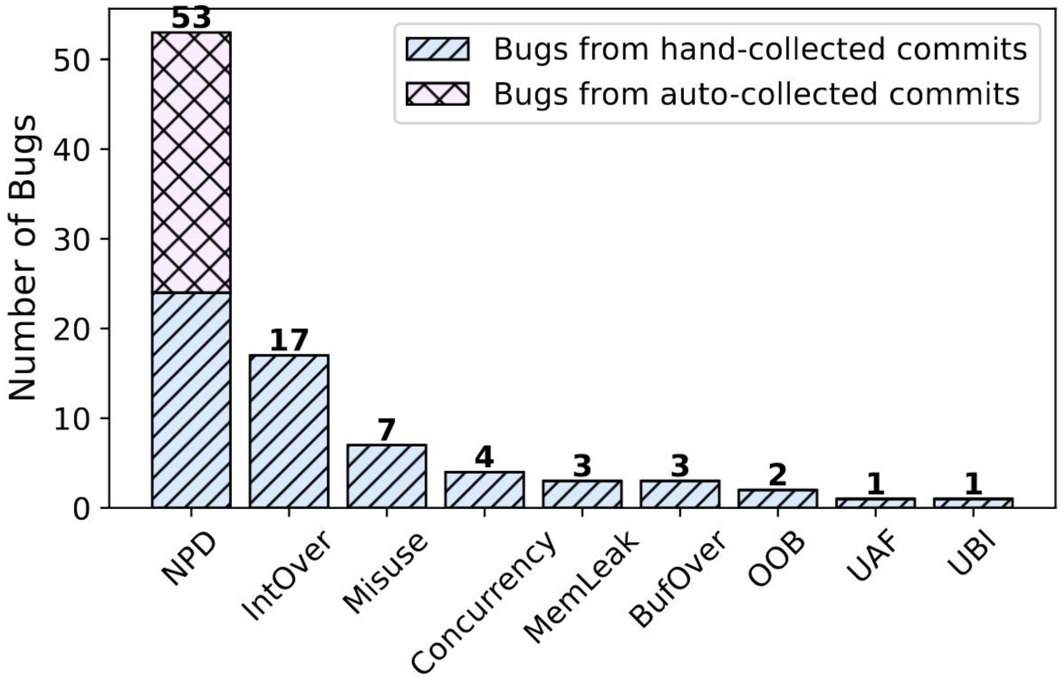
- Run the 37 valid checkers on the entire codebase
- Found 29 false positives (**32.2%**)
  - Most of them are caused by **trigger condition management** such as failing to recognize a pointer had already been validated before use

RQ2: New Bugs

- Run KNighter on **61 hand-picked** commits + **100 automatically** collected commits

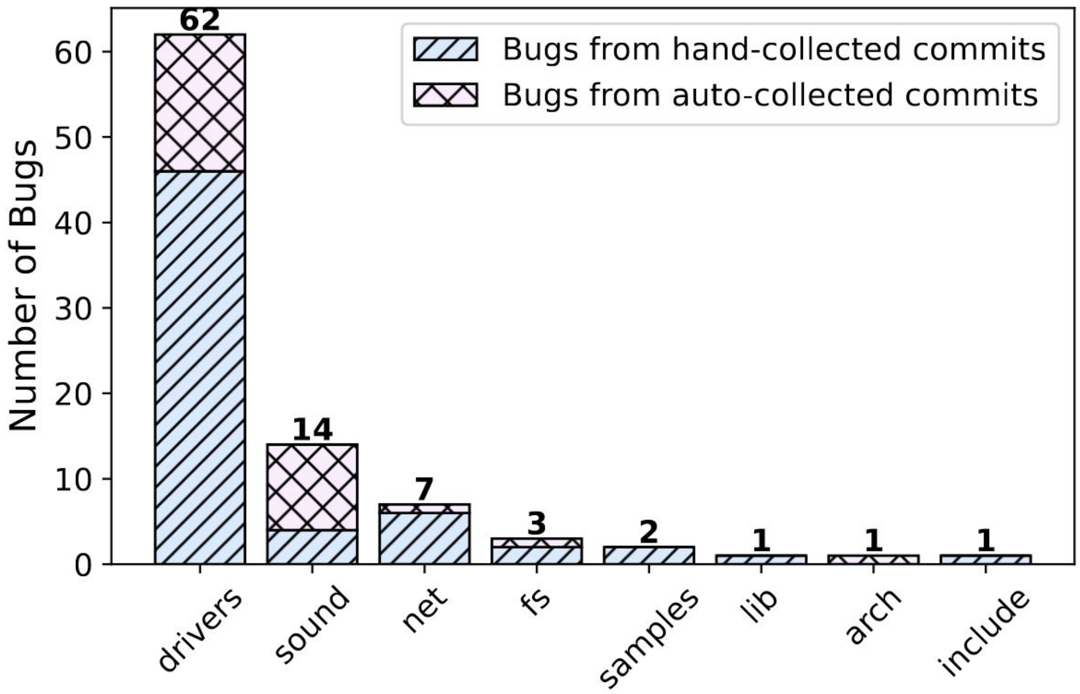
	Total	Confirmed	Fixed	Pending	CVE
KNighter	92	77	57	15	16

RQ2: New Bugs



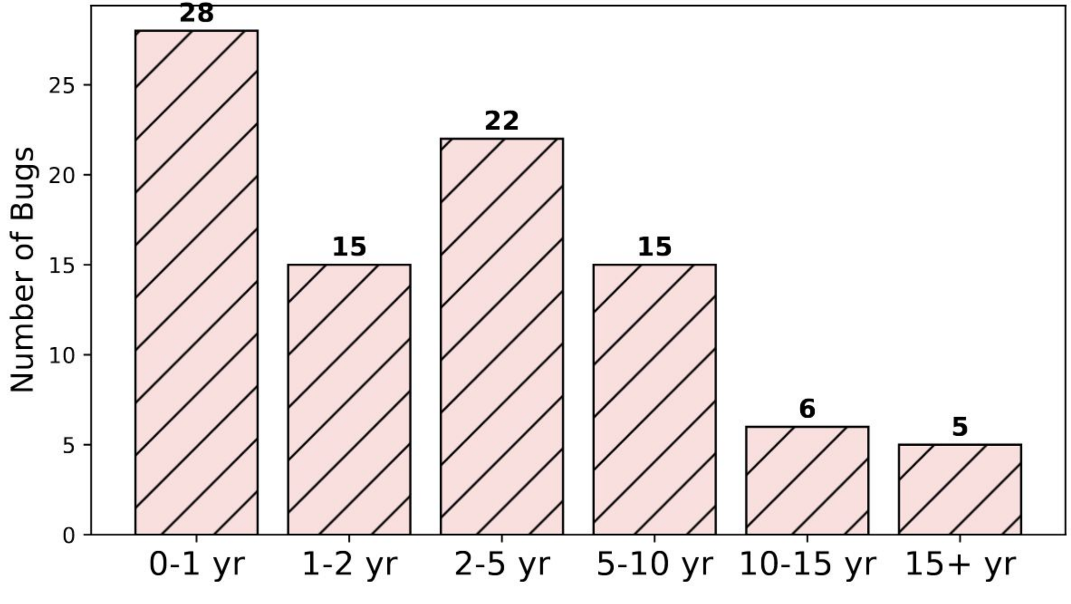
Bugs from hand-collected commits
Bugs from auto-collected commits
Number of Bugs
0
10
20
30
40
50
53
17
7
4
3
3
2
1
1
NPD
IntOver
Misuse
Concurrency
MemLeak
BufOver
OOB
UAF
UBI

RQ2: New Bugs



Number of Bugs
0
10
20
30
40
50
60
62
14
7
3
2
1
1
1
drivers
sound
net
fs
samples
lib
arch
include
Bugs from hand-collected commits
Bugs from auto-collected commits

RQ2: New Bugs



(d) Number of bugs with different lifetimes.

RQ3: Comparison with handmade checkers

- Compared with Smatch (static analysis tool used for linux kernel)
- **Smatch failed to detect** any of our true positive bugs
  - Smatch do not fully leverage the domain-specific knowledge embedded in the Linux kernels

RQ4: Ablation Test

- Evaluate the effect of a component independently
  - Multi-stage vs single-stage
  - Example selection (manual vs RAG)
  - Different LLM models

Table 2. Ablation study results. "Default" means KNighter's standard configuration utilizing multi-stage synthesis, fixed few-shot examples, and the O3-mini model. Alternative configurations are compared against this baseline.

Variants	Valid	Errors		
		Syntax	Runtime	Semantics
Default	12	28	0	75
W/o multi-stage	8	52	3	75
W/ RAG	12	37	4	62
W/ GPT-4o	11	31	0	76
W/ DeepSeek-R1	11	29	8	66
W/ Gemini-2-flash	4	130	2	44

Limitations

- It’s limited to bug patterns specific for C and doesn’t account for the semantic of the bug
- The heuristics used to detect false positive can be improved

Thanks!