





BugAuditor: Detecting Bugs via Inconsistent Defensive Code Auditing

Miaoqian Lin[1], Kai Chen[2], Hao Chen[1]

[1]*The University of Hong Kong*
[2]*Institute of Information Engineering, Chinese Academy of Sciences*

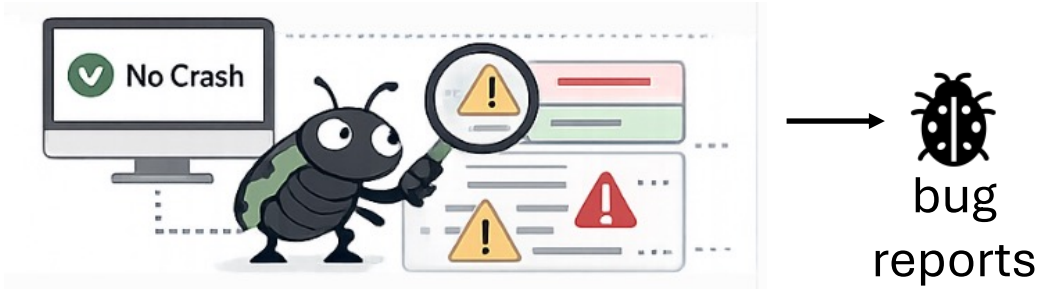
Bug auditing

Where to check?



locations

How to decide?



Diverse and complex behaviors

Hard to enumerate effectively

Silent and **subtle** bugs

Hard to capture via crashes

The key to bug auditing: Bug oracle



Oracle that guides **where to check** and decides **what is a bug.**

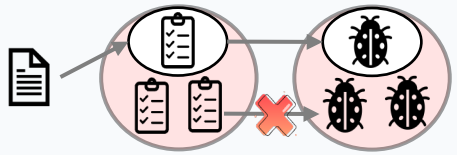
covers critical project-specific behaviors

without relying on external signals

How existing methods fail?

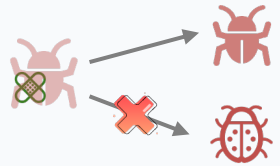
- Fail to cover many project-specific behaviors and thus miss relevant bugs.

① Using software document



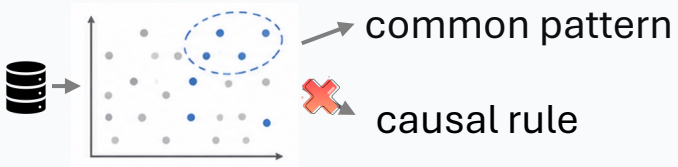
Limited to documented specifications

② Using known bugs



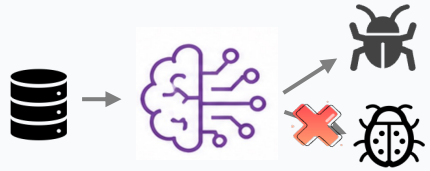
Limited to known bug patterns

③ Mining frequent patterns



Limited to frequent code patterns

④ Asking LLM directly

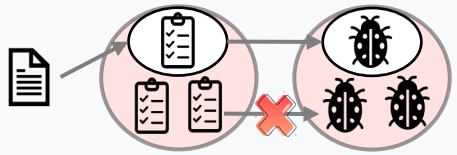


Limited to memorized knowledge

How existing methods fail?

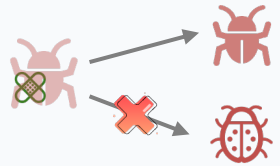
- Fail to cover many project-specific behaviors and thus miss relevant bugs.

① Using software document



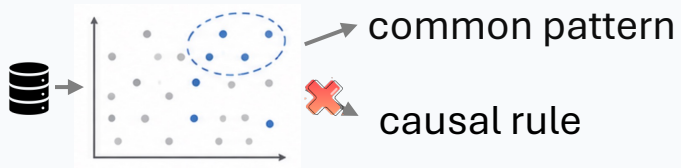
Limited to documented specifications

② Using known bugs



Limited to known bug patterns

③ Mining frequent patterns



Limited to frequent code patterns

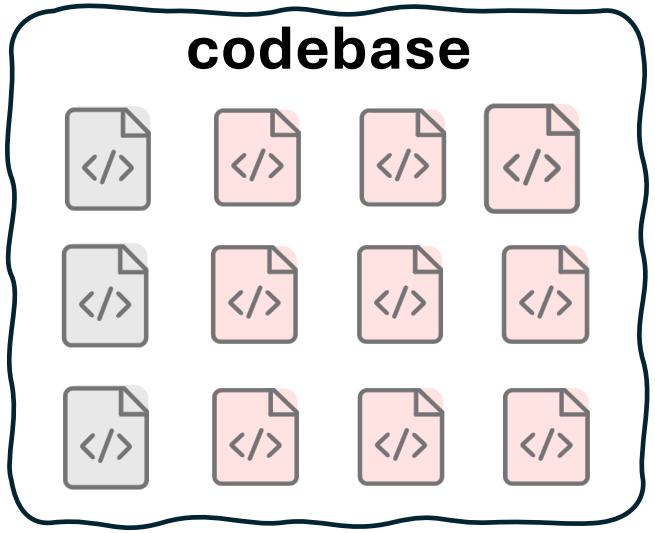
④ Asking LLM directly



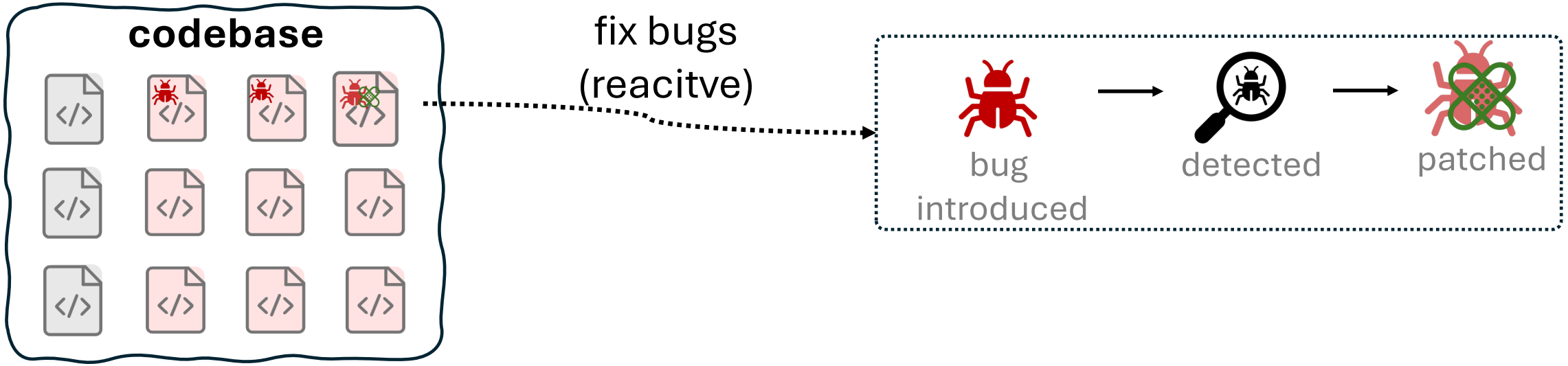
Limited to memorized knowledge

Our idea: Detecting bugs via inconsistent defensive code auditing.

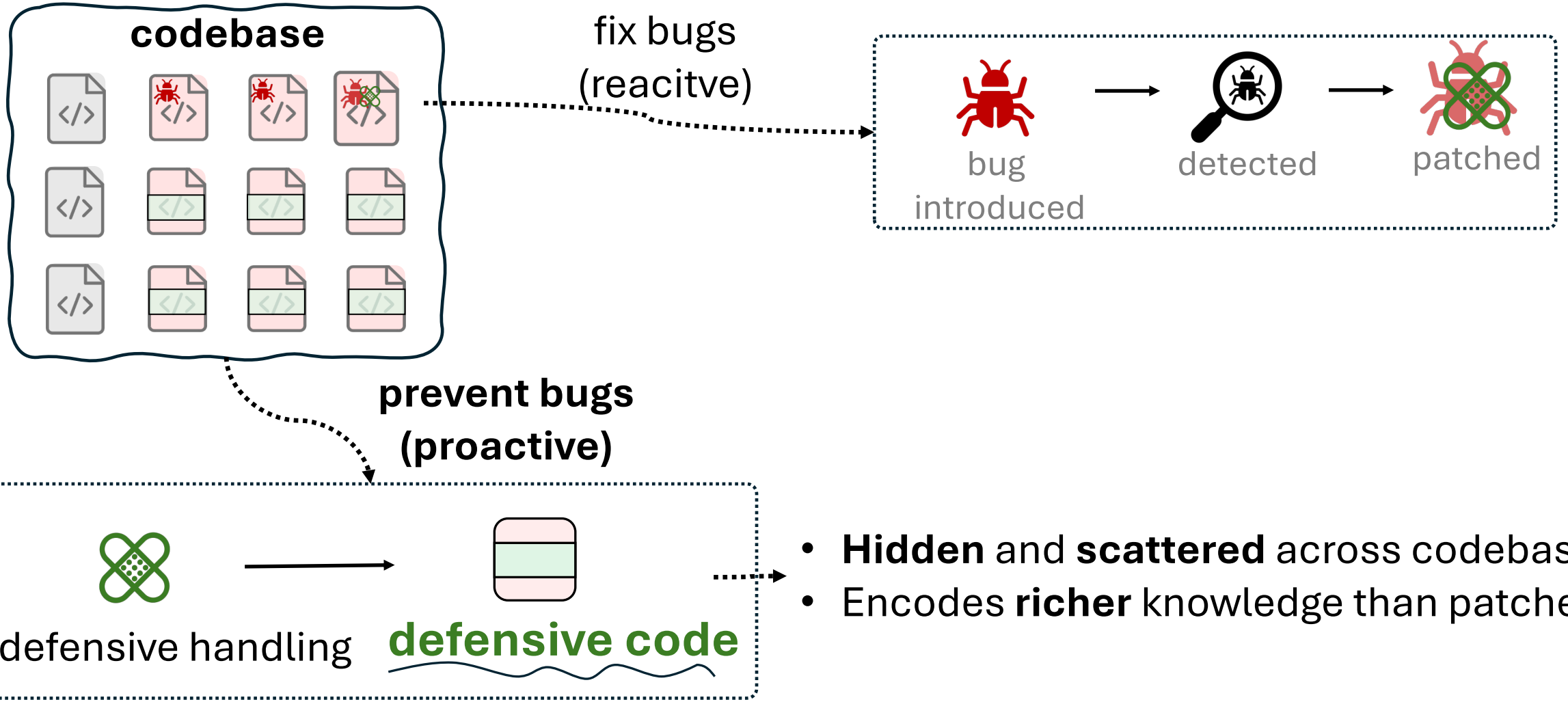
Observation: Defensive code in codebase



Observation: Defensive code in codebase

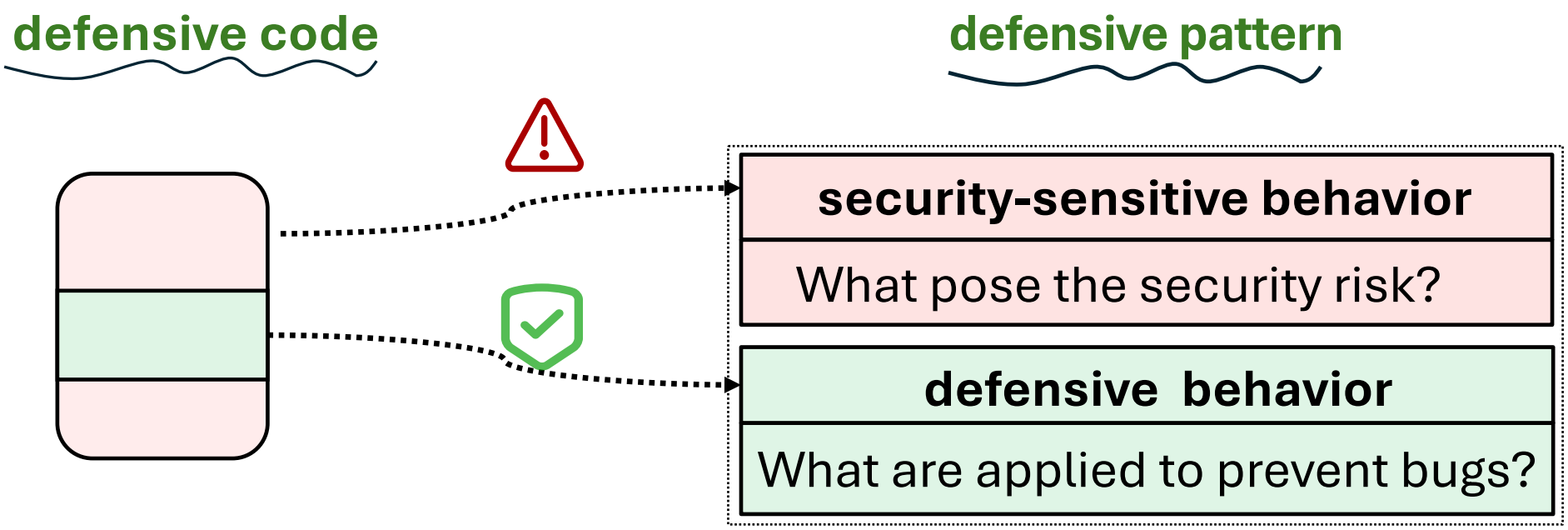


Observation: Defensive code in codebase



- **Hidden** and **scattered** across codebase
- Encodes **richer** knowledge than patches

Observation: Defensive patterns encoded in defensive code



Source of **implicit** knowledge

Make implicit knowledge **explicit**

Observation: Defensive patterns encoded in defensive code

Example of defensive code snippet

```
01 static int __init lpc32xx_clocksource_init(...) {
02   ...
03   clk = of_clk_get_by_name(np, "timerclk");
04   if (IS_ERR(clk)) {
05       ...
06   }
07   ret = clk_prepare_enable(clk);
08   if (ret) {
09       pr_err("clock enable failed (%d)\n", ret);
10       goto err_clk_enable;
11   }
12   ...
13  err_clk_enable:
14    clk_put(clk);
15    return ret;
16 }
```





Example of defensive pattern

security-sensitive behavior

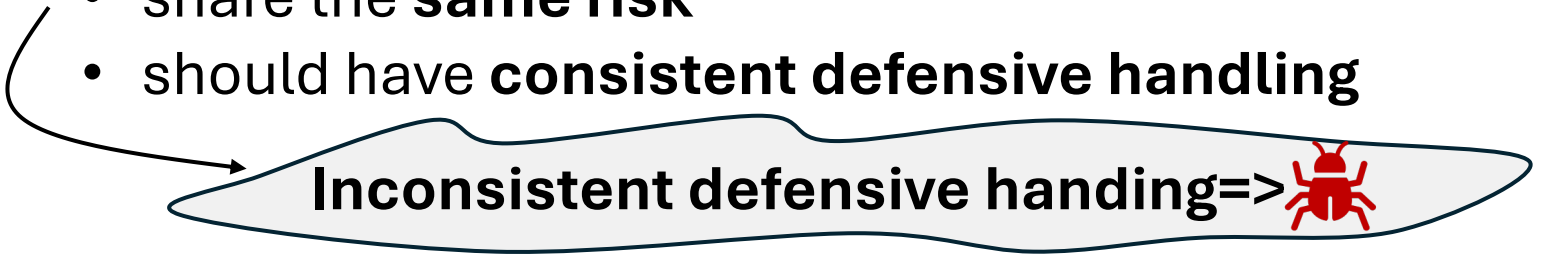
When of_clk_get_by_name is called, it creates a resource leak risk because failing to release the acquired clock reference with clk_put leads to unbalanced resource management.

defensive behavior

The clk_put is executed in cleanup sections to prevent resource leaks by releasing the clock reference acquired by of_clk_get_by_name

Idea: Inconsistent defensive code auditing

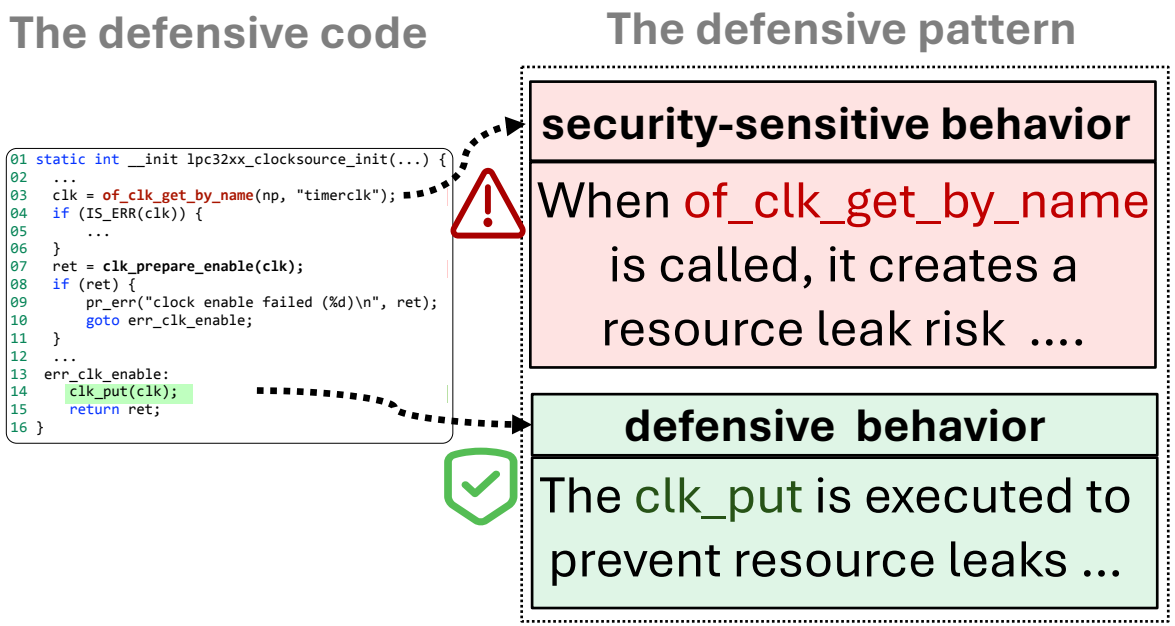
- **Same security-sensitive behavior** across different contexts
  - share the **same risk**
  - should have **consistent defensive handling**



Idea: Inconsistent defensive code auditing

- **Same security-sensitive behavior** across different contexts
  - share the **same risk**
  - should have **consistent defensive handling**

Inconsistent defensive handing=>



Idea: Inconsistent defensive code auditing

- **Same security-sensitive behavior** across different contexts
  - share the **same risk**
  - should have **consistent defensive handling**

Inconsistent defensive handing=> 

The defensive code

```
01 static int __init lpc32xx_clocksource_init(...) {
02   ...
03   clk = of_clk_get_by_name(np, "timerclk");
04   if (IS_ERR(clk)) {
05       ...
06   }
07   ret = clk_prepare_enable(clk);
08   if (ret) {
09       pr_err("clock enable failed (%d)\n", ret);
10       goto err_clk_enable;
11   }
12   ...
13 err_clk_enable:
14     clk_put(clk);
15     return ret;
16 }
```

The defensive pattern

security-sensitive behavior

When of_clk_get_by_name is called, it creates a resource leak risk ....

defensive behavior

The clk_put is executed to prevent resource leaks ...

same security-sensitive behavior

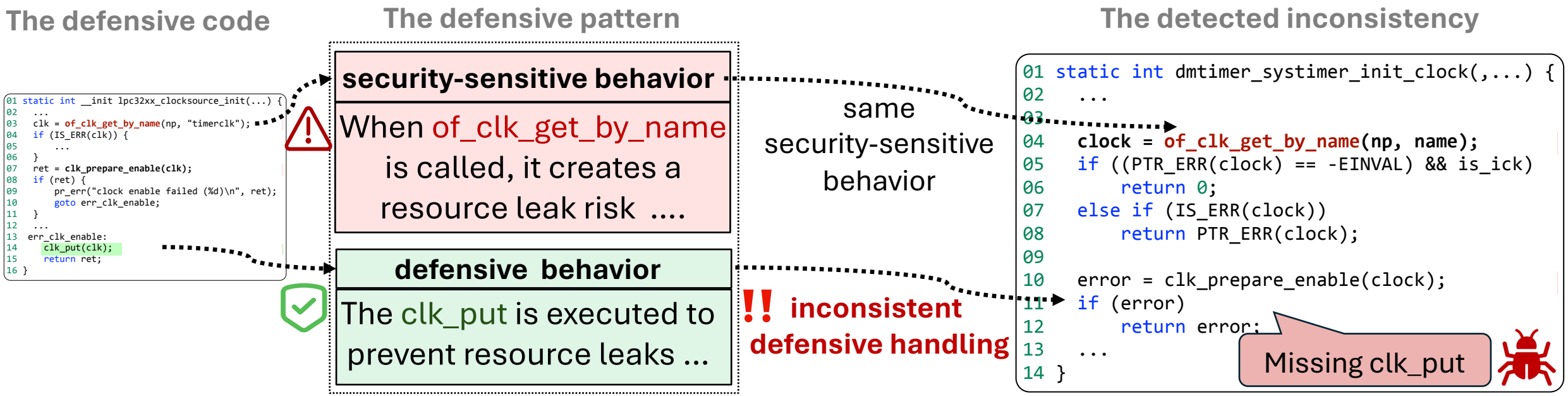
The detected inconsistency

```
01 static int dmtimer_systimer_init_clock(,...) {
02   ...
03
04   clock = of_clk_get_by_name(np, name);
05   if ((PTR_ERR(clock) == -EINVAL) && is_ick)
06       return 0;
07   else if (IS_ERR(clock))
08       return PTR_ERR(clock);
09
10   error = clk_prepare_enable(clock);
11   if (error)
12       return error;
13   ...
14 }
```

Idea: Inconsistent defensive code auditing

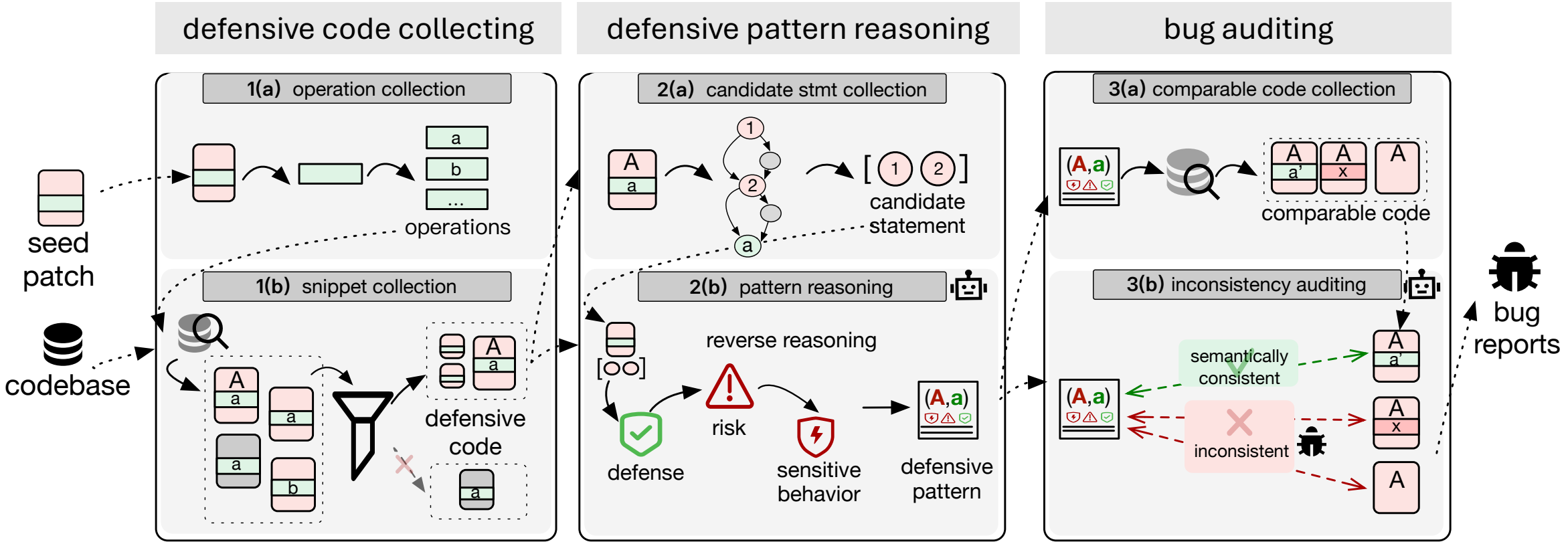
- **Same security-sensitive behavior** across different contexts
  - share the **same risk**
  - should have **consistent defensive handling**

Inconsistent defensive handing=>



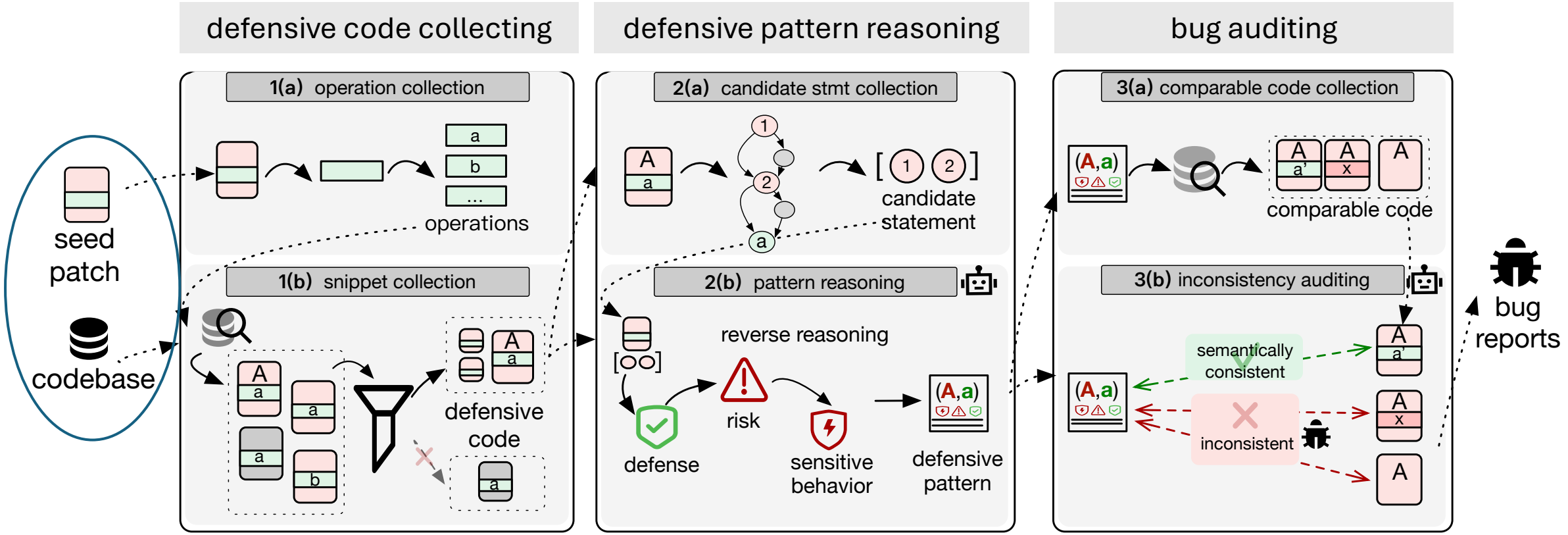
BugAuditor: Auditing bug via inconsistent defensive handling

- Collecting defensive code, inferring defensive patterns and auditing bugs.



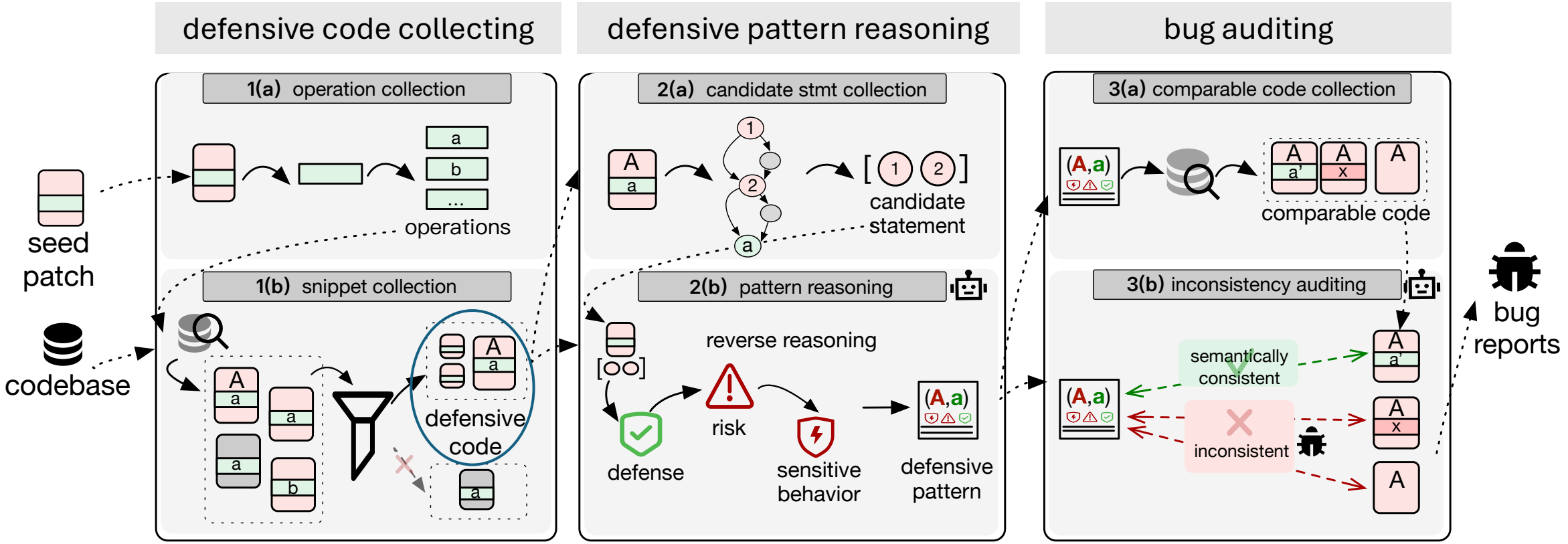
BugAuditor: Auditing bug via inconsistent defensive handling

- Collecting defensive code, inferring defensive patterns and auditing bugs.



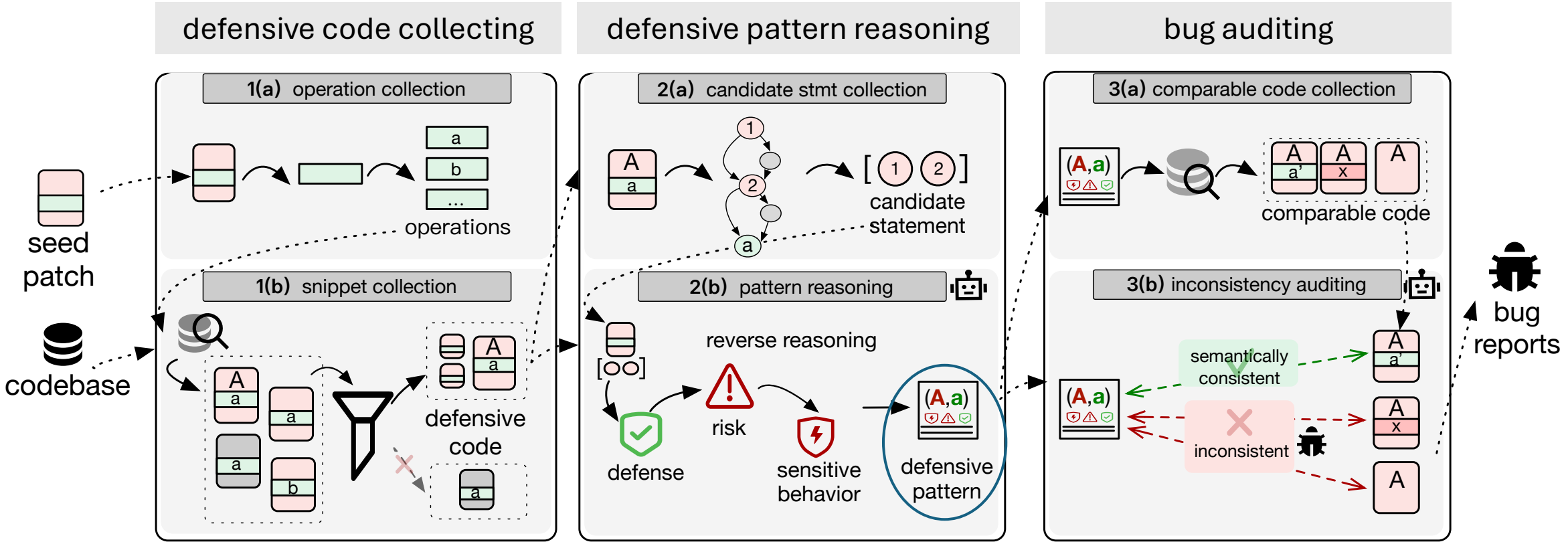
BugAuditor: Auditing bug via inconsistent defensive handling

- Collecting defensive code, inferring defensive patterns and auditing bugs.



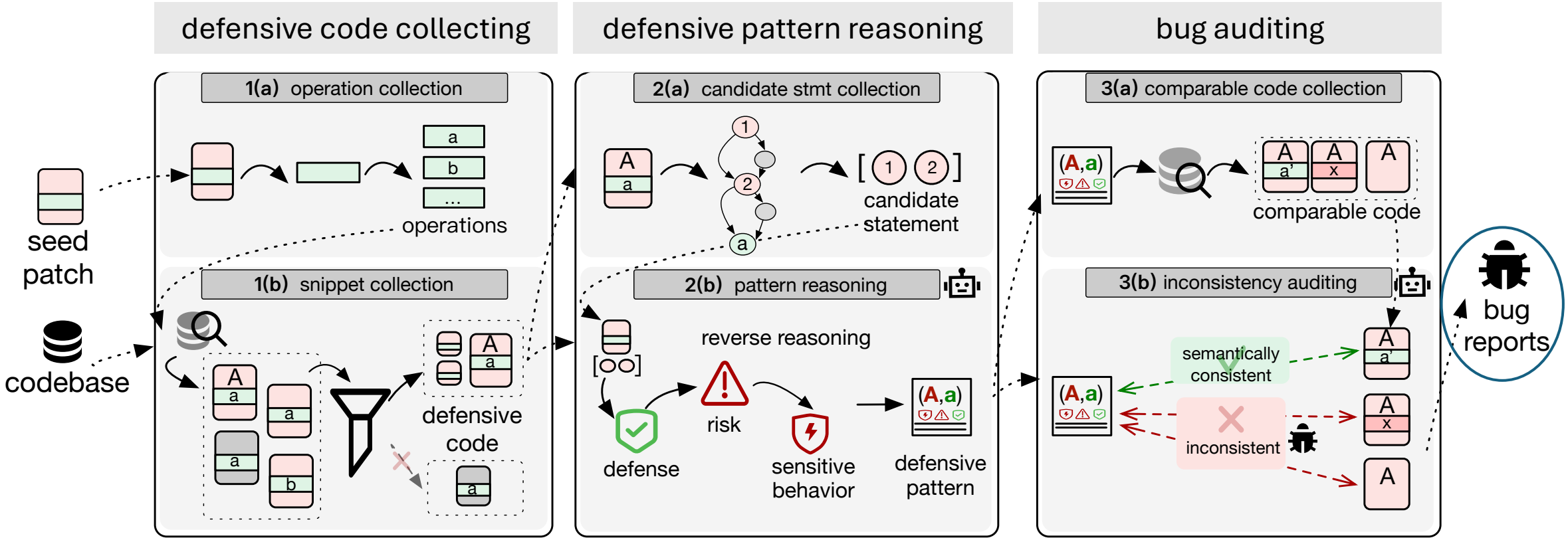
BugAuditor: Auditing bug via inconsistent defensive handling

- Collecting defensive code, inferring defensive patterns and auditing bugs.



BugAuditor: Auditing bug via inconsistent defensive handling

- Collecting defensive code, inferring defensive patterns and auditing bugs.



BugAuditor: Collecting defensive code

- How do we find defensive code hidden in the codebase?

BugAuditor: Collecting defensive code

- How do we find defensive code hidden in the codebase?
- **Start simple:** One simple defense exposes many security-sensitive contexts.

BugAuditor: Collecting defensive code

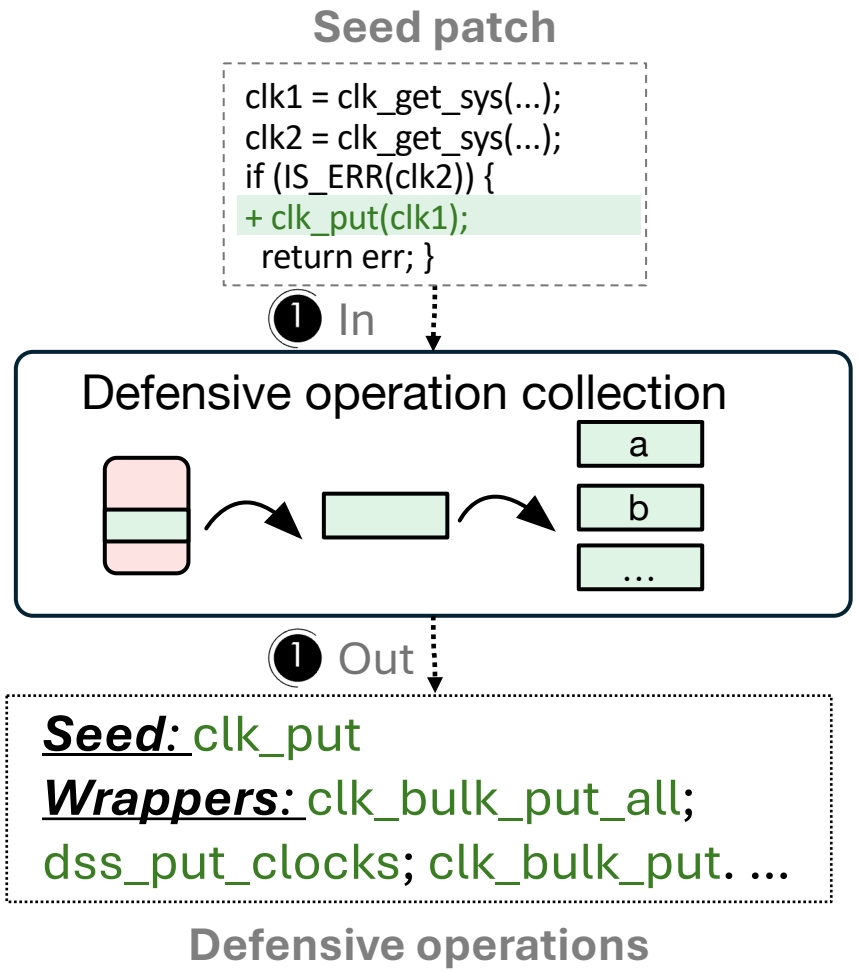
- How do we find defensive code hidden in the codebase?
- **Start simple:** One simple defense exposes many security-sensitive contexts.

Seed patch

```
clk1 = clk_get_sys(...);
clk2 = clk_get_sys(...);
if (IS_ERR(clk2)) {
+ clk_put(clk1);
  return err; }
```

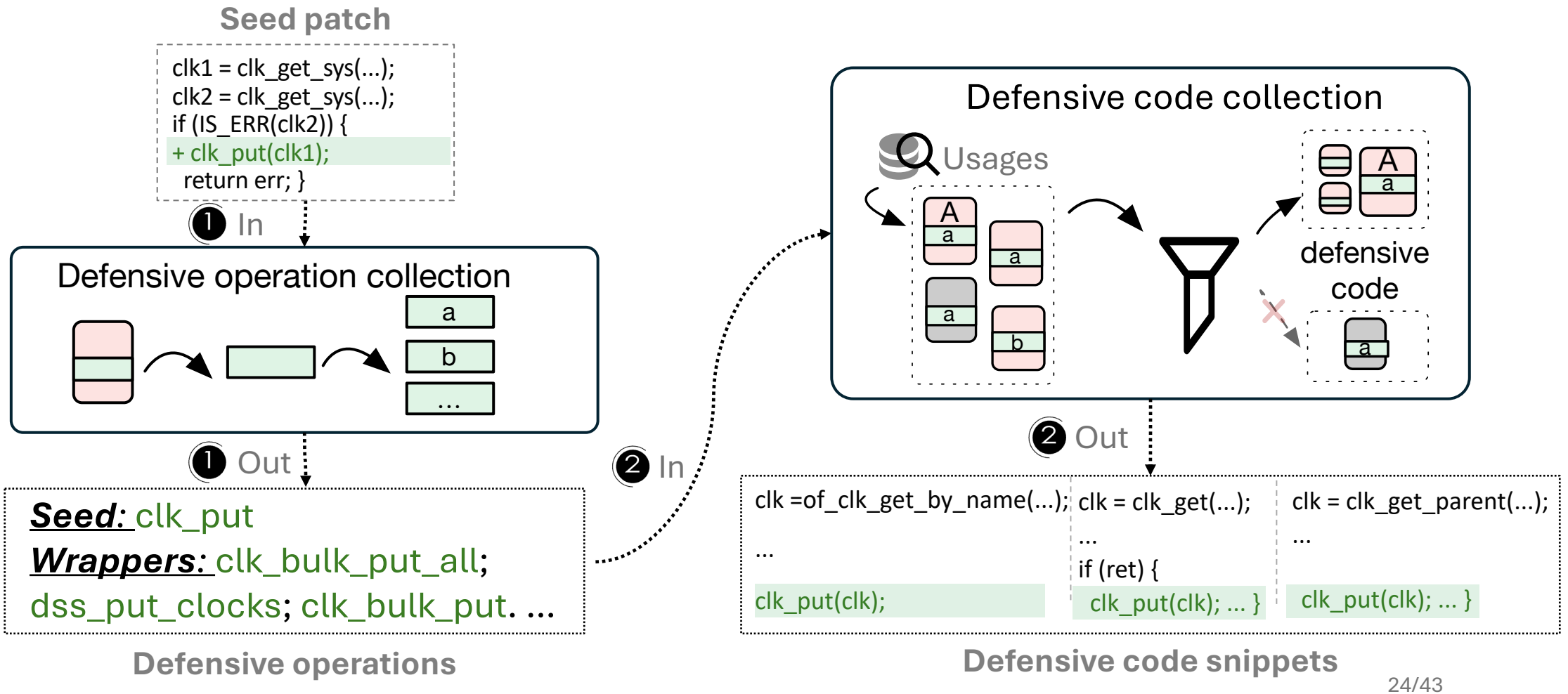
BugAuditor: Collecting defensive code

- How do we find defensive code hidden in the codebase?
- **Start simple**: One simple defense exposes many security-sensitive contexts.



BugAuditor: Collecting defensive code

- How do we find defensive code hidden in the codebase?
- **Start simple**: One simple defense exposes many security-sensitive contexts.



BugAuditor: Defensive pattern reasoning

- Which behavior makes this defense necessary?

BugAuditor: Defensive pattern reasoning

- Which behavior makes this defense necessary?
- **Abductive reasoning**: Infer security-sensitive behaviors from the observed defense

BugAuditor: Defensive pattern reasoning

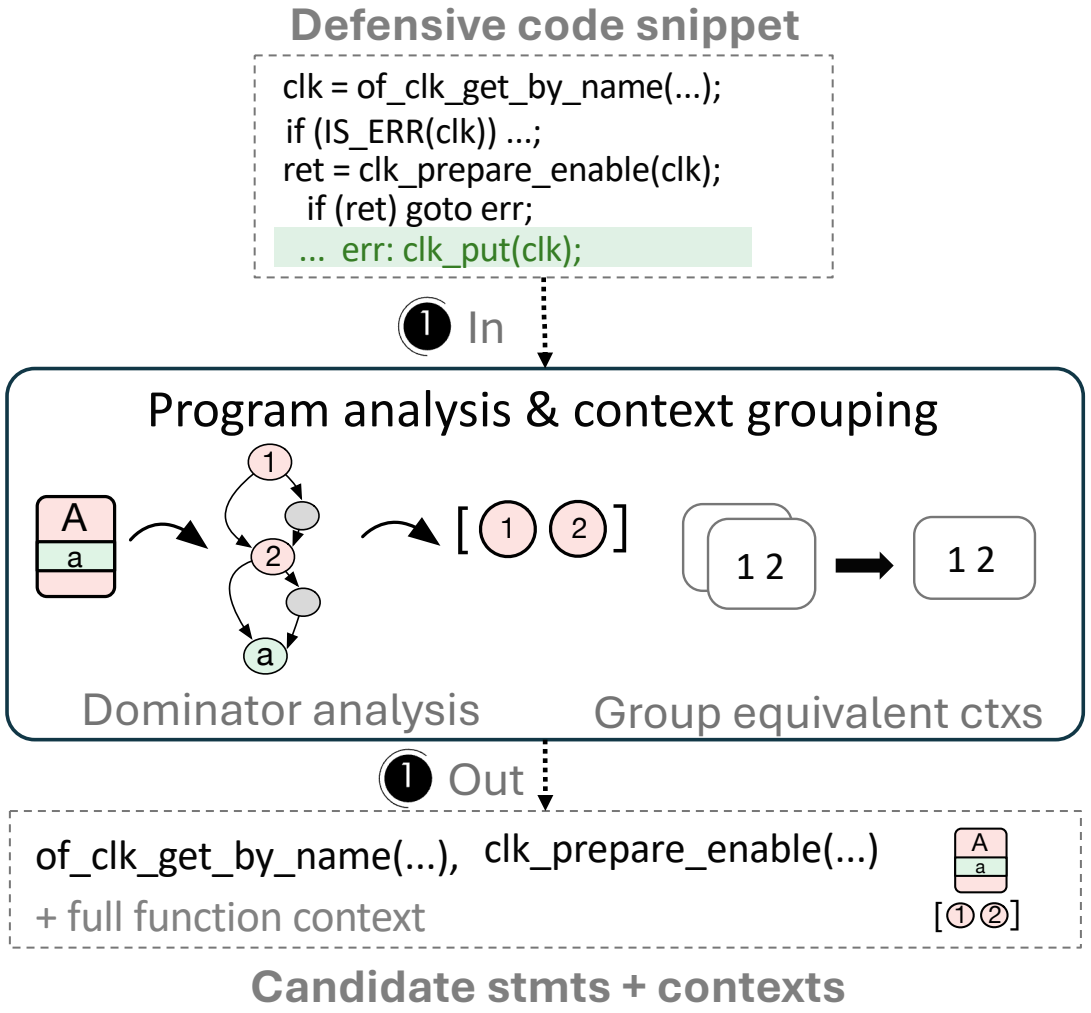
- Which behavior makes this defense necessary?
- **Abductive reasoning**: Infer security-sensitive behaviors from the observed defense

Defensive code snippet

```
clk = of_clk_get_by_name(...);
if (IS_ERR(clk)) ...;
ret = clk_prepare_enable(clk);
  if (ret) goto err;
... err: clk_put(clk);
```

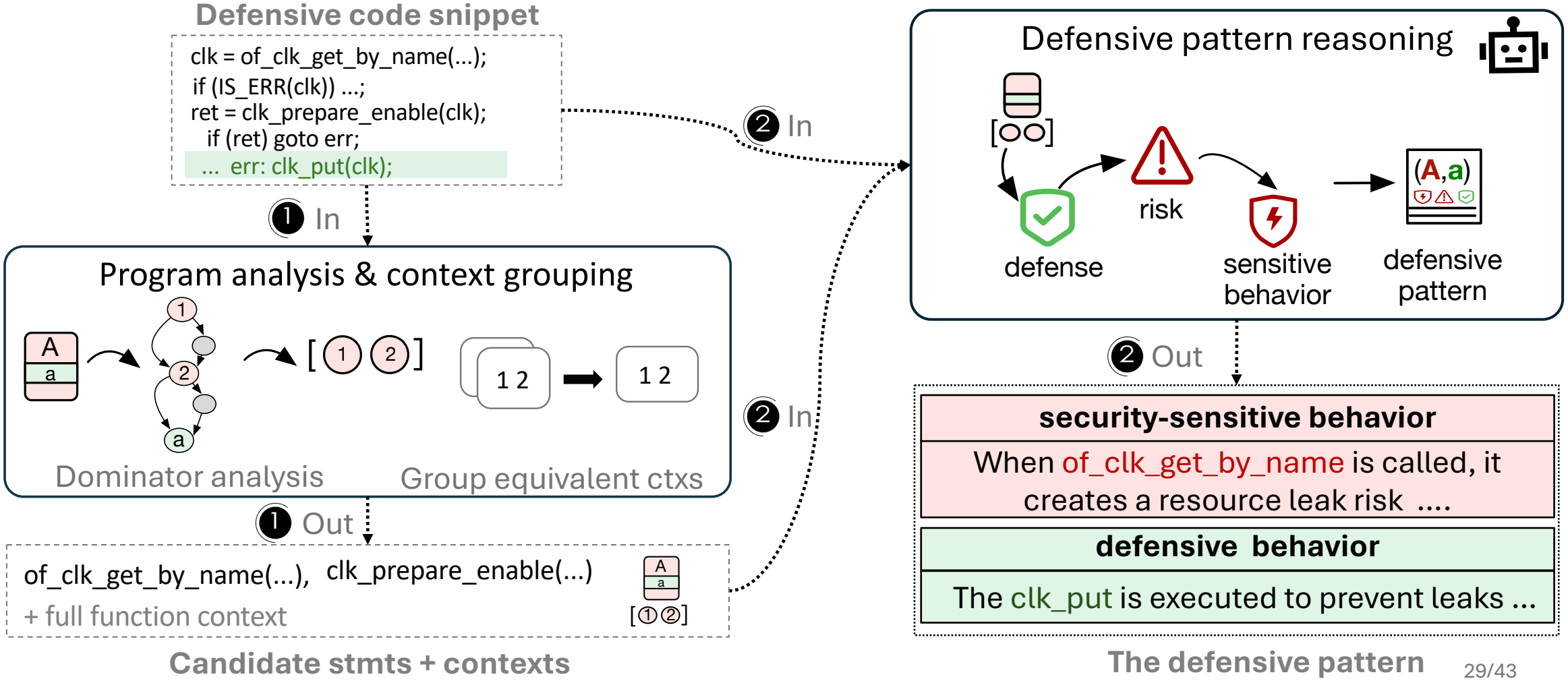
BugAuditor: Defensive pattern reasoning

- Which behavior makes this defense necessary?
- **Abductive reasoning**: Infer security-sensitive behaviors from the observed defense



BugAuditor: Defensive pattern reasoning

- Which behavior makes this defense necessary?
- **Abductive reasoning**: Infer security-sensitive behaviors from the observed defense



BugAuditor: Bug auditing

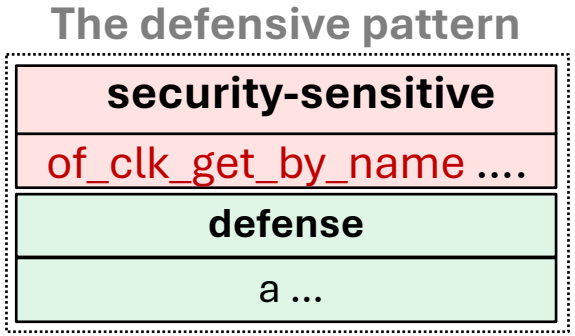
- How do we find real inconsistencies at scale?

BugAuditor: Bug auditing

- How do we find real inconsistencies at scale?
- **Hybrid auditing**: Locate by syntax, check inconsistencies by semantics.

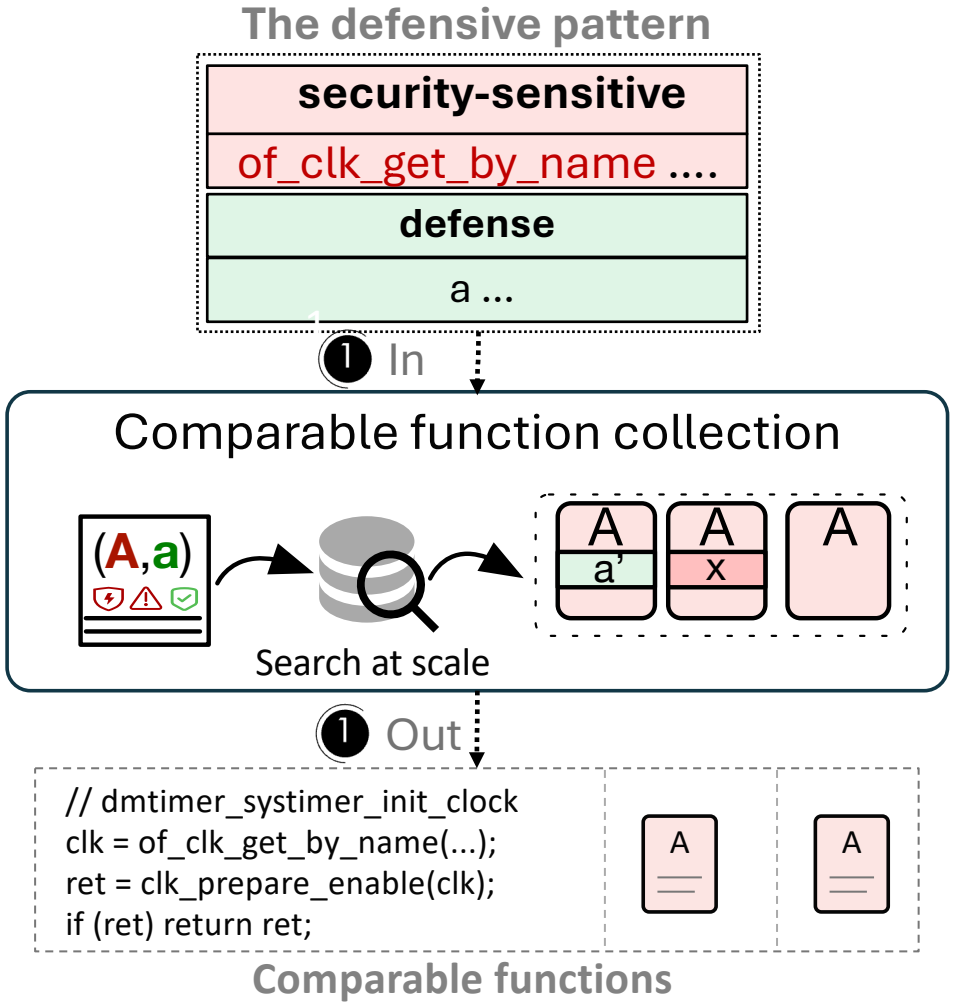
BugAuditor: Bug auditing

- How do we find real inconsistencies at scale?
- **Hybrid auditing**: Locate by syntax, check inconsistencies by semantics.



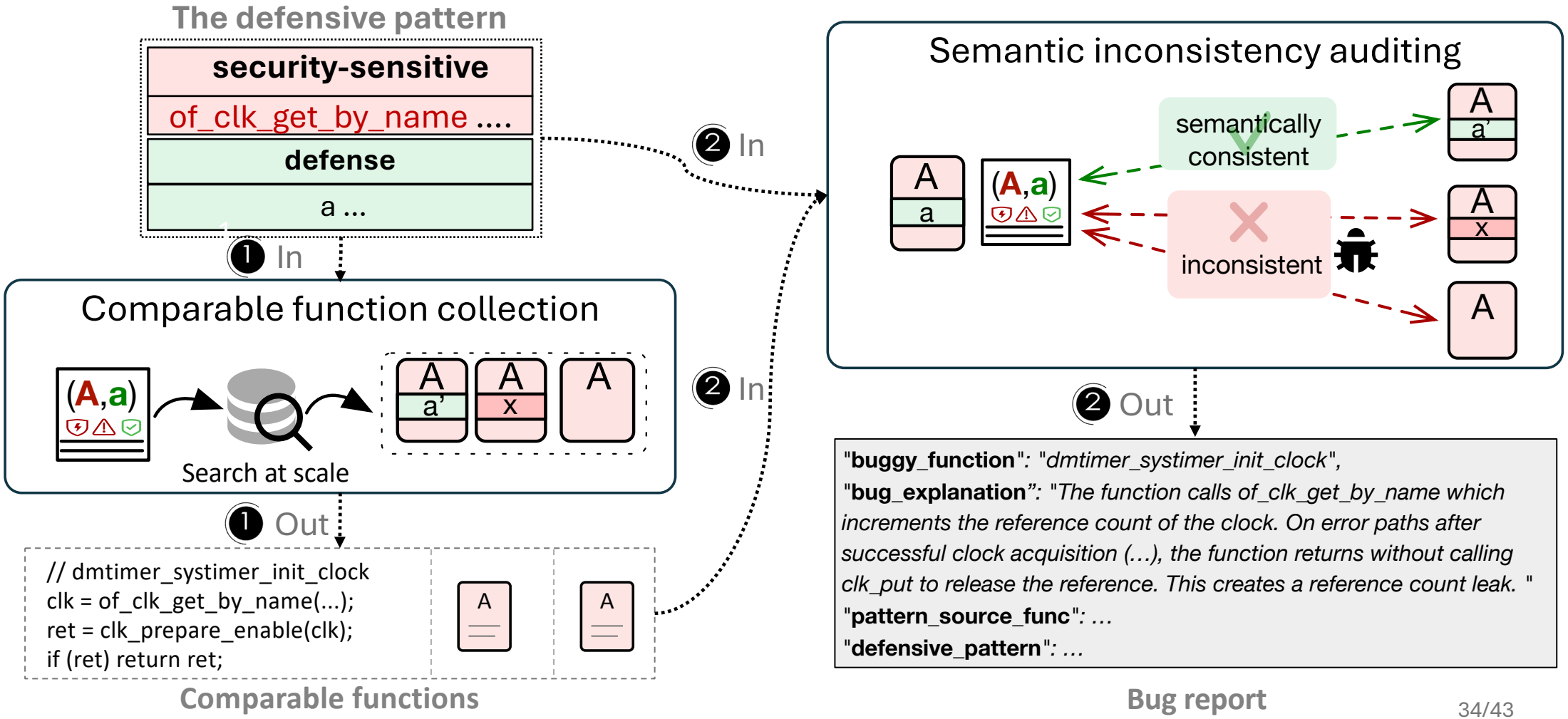
BugAuditor: Bug auditing

- How do we find real inconsistencies at scale?
- **Hybrid auditing:** Locate by syntax, check inconsistencies by semantics.



BugAuditor: Bug auditing

- How do we find real inconsistencies at scale?
- **Hybrid auditing:** Locate by syntax, check inconsistencies by semantics.



Evaluation setup

- **LLM**: DeepSeek-V3.2
- **Program**: the Linux kernel (v6.10-rc4), more than 27 million LoC.
- **Seeds**: six defensive operations from bug patches.

Type	Operation	Code
Resource releases	Memory release	`kfree()`
	Device reference release	`of_node_put()`
	Clock reference release	`clk_put()`
Security checks	Error pointer check	`if (IS_ERR(val))`
	Negative value check	`if (val < 0)`
	NULL pointer check	`if (!val),` `if (val != NULL)`

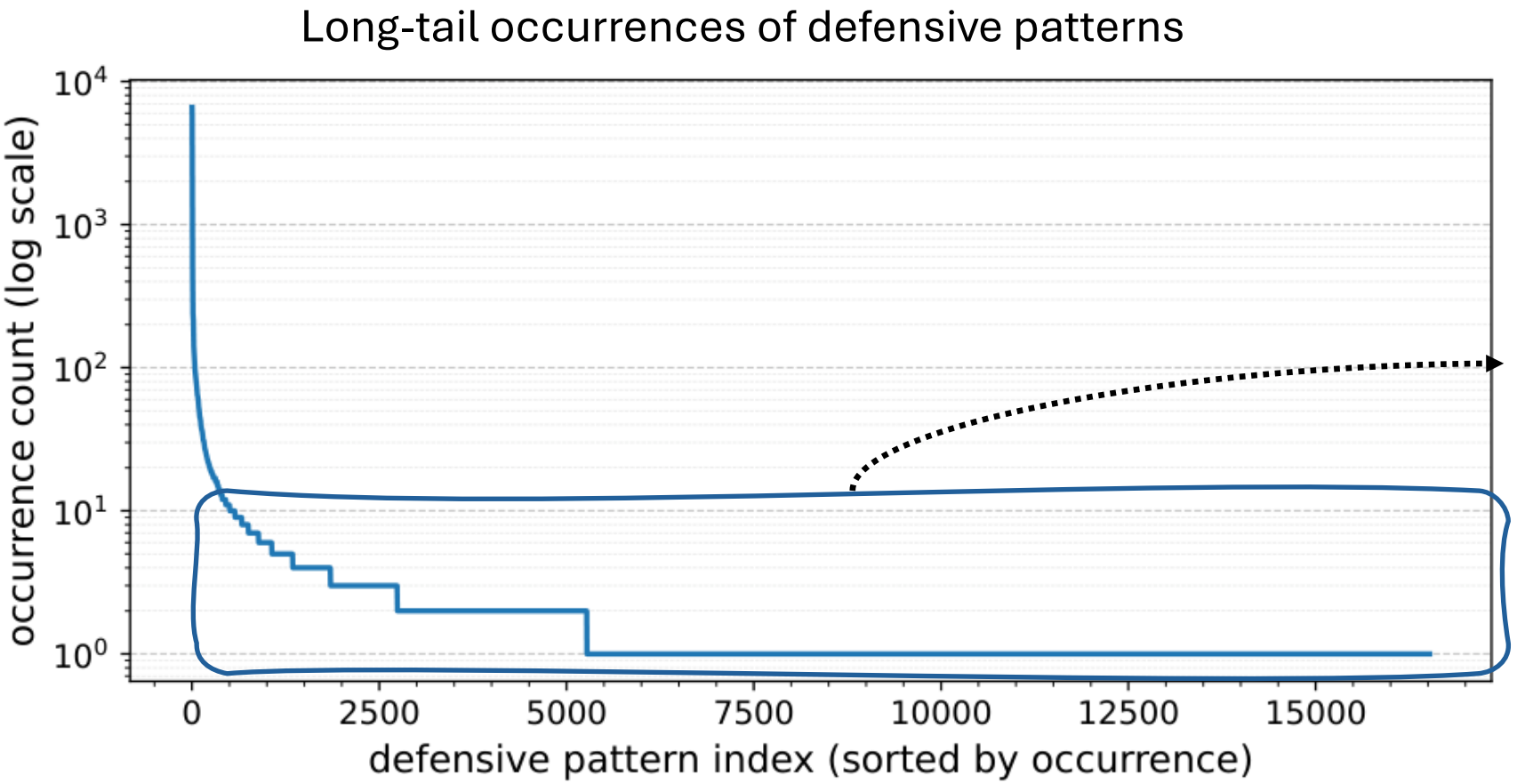
Results: Defensive pattern reasoning

- From **6** seeds, obtain **16508** defensive patterns.
- Pattern accuracy: **92.3%** (random sample)

Operation	# Snippet	#Pattern
kfree	11921	922
of_node_put	1664	102
clk_put	181	8
NULL pointer check	27344	6175
Negative value check	16114	7013
Error pointer check	5208	2288
Total	**62432**	**16508**

Results: Occurrence of mined defensive patterns

- BugAuditor captures patterns without requiring frequent occurrence.



96.97%

occur fewer than 10 times

Results: New bug detection

- 675 reports, **54** confirmed true bugs, 20 fixed to date.
- Bug lifetime: 5.3 years average, **18.4** years longest

Bug type	**#Bug**
Resource Leak	51
NULL/Invalid Pointer Dereference	2
Information Leak	1
Total	**54**

Results: Runtime and token cost

- Linux kernel **27M+** LoC, DeepSeek-V3.2
- The evaluation takes less than **18 hours** and costs **$17.47** in API usage.

Stage	Runtime	Tokens: input / output	API Cost.
Code collecting	40 min	—	—
Pattern reasoning	13 h 36 min	17.7M / 1.4M	$5.54
Bug auditing	3 h 41 min	40.8M / 1.2M	$11.93
Measured total (≤100 functions audited per pattern)	**17 h 57 min**	**58.5M / 2.6M**	**$17.47**
Full-scale estimate (all functions per pattern)	25 h 29 min	142.0M / 5.0M	$41.88

BugAuditor achieves efficient and low-cost bug auditing.

Results: Compared with LLM-only method

Can the LLM-only method detect the 54 bugs found by BugAuditor?

Model	Function-level	File-level
DeepSeek-V3.2 (used by BugAuditor)	7 (13.0%)	4 (7.4%)
GPT-5.4 (a stronger model)	**11 (20.4%)**	**10 (18.5%)**

BugAuditor (with DeepSeek-V3.2) detects bugs missed by GPT-5.4

Results: Generalization across software

Can BugAuditor generalize to other software projects?

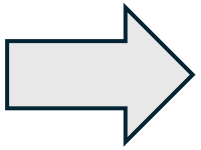
Program	Seed operations	#Usages	#Snippets	#Patterns	Accuracy	New bugs
OpenSSL	NULL check OPENSSL_free	2,922	1,688	**382**	**95%**	**2**
FFmpeg	NULL check av_free	3,782	2,883	**371**	**88%**	**4**

BugAuditor discovers diverse patterns and new bugs across software.

Limitations and future directions

Current

- Few seed operations
- Intra-function snippets
- Search by syntax



Future direction

- Cover more defenses
- Reason across functions
- Search by semantics

BugAuditor

- Observation: defensive code ≈ rich source of implicit specifications
- Key idea: detecting bugs via **inconsistent defensive code auditing.**
- Results
  - From 6 seeds → **16508** patterns (92.3% acc.)
  - **54** new bugs found in Linux (oldest: 18.4 years)

Paper & code



Contact: miaoqian@hku.hk

