Pattern-driven Concurrency Bug Detection for Operating System Kernel

Hong, Shin

Provable Software Laboratory
CS Dept. KAIST

Motivation

- Concurrent programs are widely spread in these days.

- However, assuring correctness of an industrial-sized concurrent program is difficult.
  - State explosion problem

- Operating system kernel

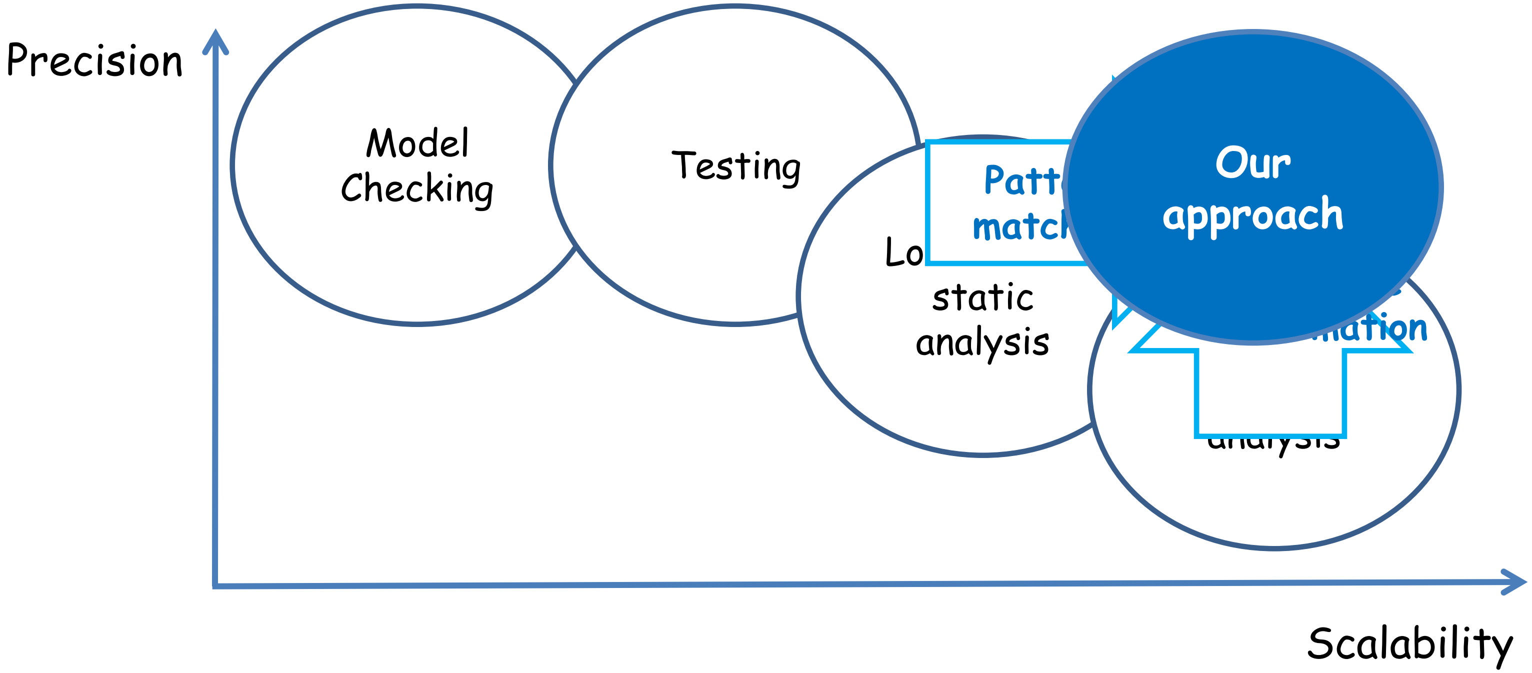
- High concurrency
- Large size program
- Complex data structures
- Various synchronization mechanisms
  - Barriers, Instructions, etc
  - 30~40% synchronization operations are not conventional binary locks.

Ex. A function from Linux MTD/UBI device driver in ver. 2.6.27.22 (Simplified)

```
int ubi_thread(void * u) {
 for (;;) {
  if (kthread_should_stop())
   break ;
  spin_lock(&ubi->wl_lock) ;
  if (list_empty(&ubi->works) ||
    ubi->ro_mode ||
    !ubi->thread_enabled) {
   ...
   spin_unlock(&ubi->wl_lock) ;
   schedule() ;
   continue ;
  }
  spin_unlock(&ubi->wl_lock) ;
  err = do_work(ubi) ;
  if (err) {
   if (failures++ > WL_MAX_FAILURES)
    break ;
  }
  cond_resched() ;
 }
}
```

Approach

- Verification techniques



- Related works
  - Lock-based static analysis techniques
    : RacerX, RELAY ➔ Lock discipline, Partial order among locks
  - Pattern-based bug detection
    : MetaL, FindBugs ➔ Low precision (Too many false alarms)

Classification of Concurrency Bugs

- We survey previous concurrency bugs from Linux file systems
  - Search Linux Change Log 2.6.1 ~ 2.6.28
  - Keyword: concurrency, data race, deadlock, livelock, file system, ext, etc.
  - In almost 300 documents, we found 40 bug reports (patches) related to both file system and concurrency bugs.

- We construct concurrency bug classification to analyze the bug reports.
  - 5 different aspects
    - Symptom:
      - Data race (machine exception), Data race (Faulty state), Deadlock, Livelock.
    - Fault :
      - Design decision violations, Incorrect use of synch. idioms, Program logic error
    - Resolution:
      - {Insert, Remove, Change, Reorder} × {Sync. operation, Data operation, Control operation}
    - Related synchronization mechanism:
      - Instruction, Barrier, Thread operations, Conditional variable, Lock, Complex lock, Semaphore
    - Synchronization granularity:
      - Kernel-level, File system-level, File-level, Inode-level
  - 27 bugs are classified

Concurrency Bug Patterns

- Based on the bug analysis result by the classification, we define the 10 concurrency bug patterns in order to detect unrevealed bugs automatically by code pattern matching.

\Symptom Sync. mechanism	Race condition	Deadlock	Livelock
Barrier	• No Memory Barrier After Object Initializations		• Busy-waiting on variable without memory barrier
Instruction	• Use atomic instructions in Non-atomic ways		
Thread operation	• Unsynchronized Data Passing to Child Thread	• Waiting Already Finished Thread	
Conditional variable		• Waiting with Lock Held	
Binary Lock	• Buggy Test and Test-and-Set • Unlock before I/O Operations	• Releasing and Re-taking Outer Lock	
Complex lock	• Unintended Big Kernel Lock Releasing		

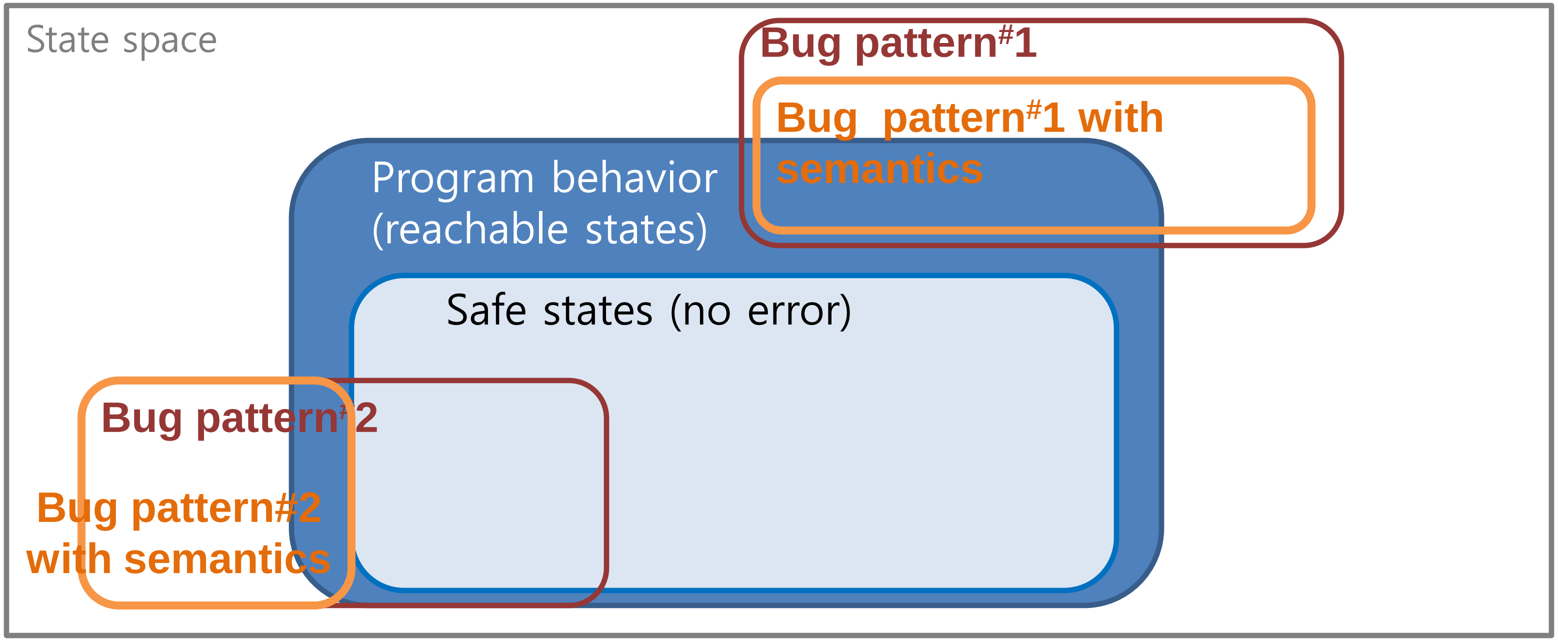
Semantics Augmented Pattern Matching

- Syntactic bug pattern matching results false positives for the following reasons:
  - No parallel thread to be scheduled
  - Synchronized by other locks
  - Shared variable initializations without holding locks

Thread sensitive analysis

Lock analysis

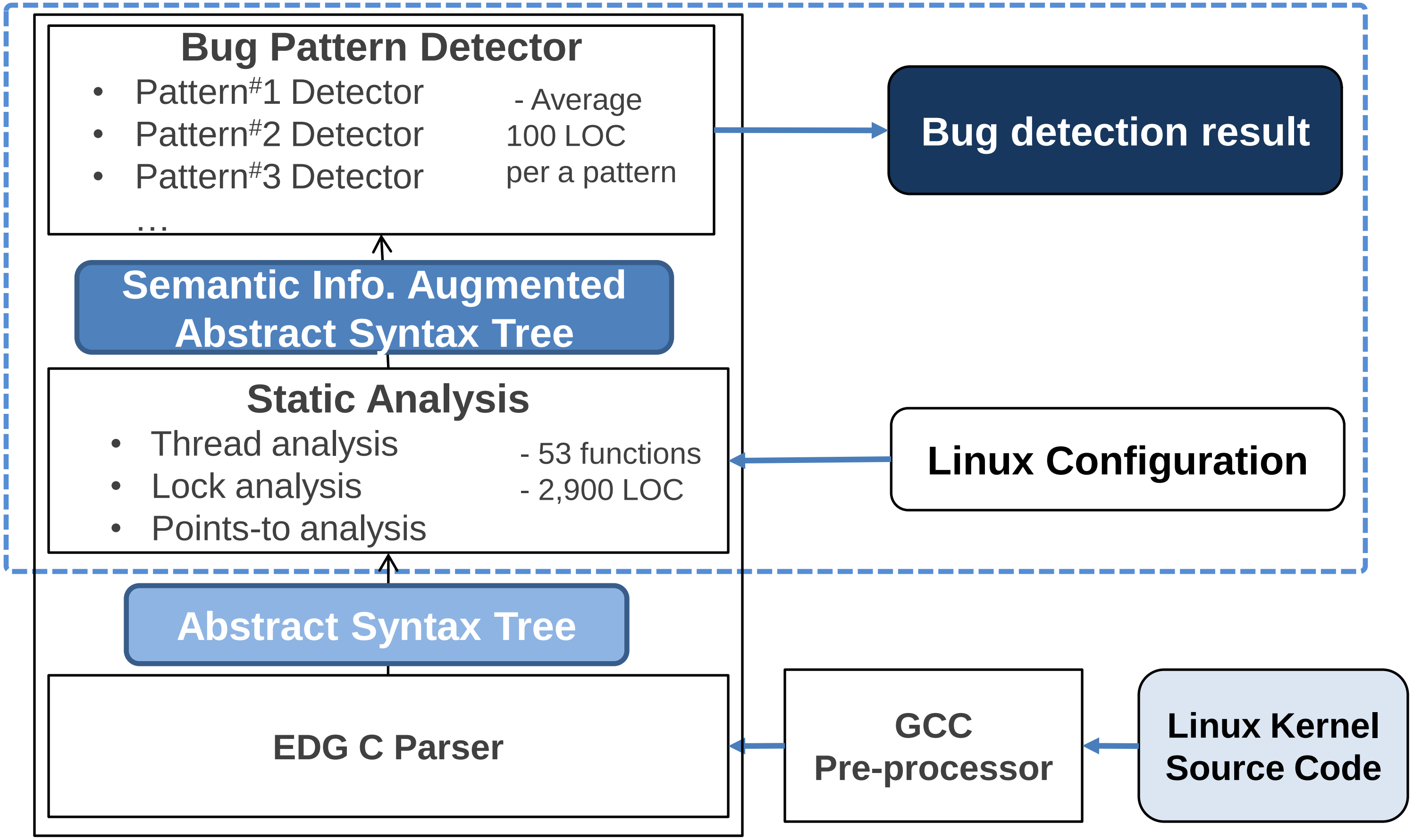
Simple points-to analysis


State space
Bug pattern#1
Bug pattern#1 with semantics
Program behavior (reachable states)
Safe states (no error)
Bug pattern#2
Bug pattern#2 with semantics

- We improve the bug pattern matching using semantic information to refine the bug detection results.

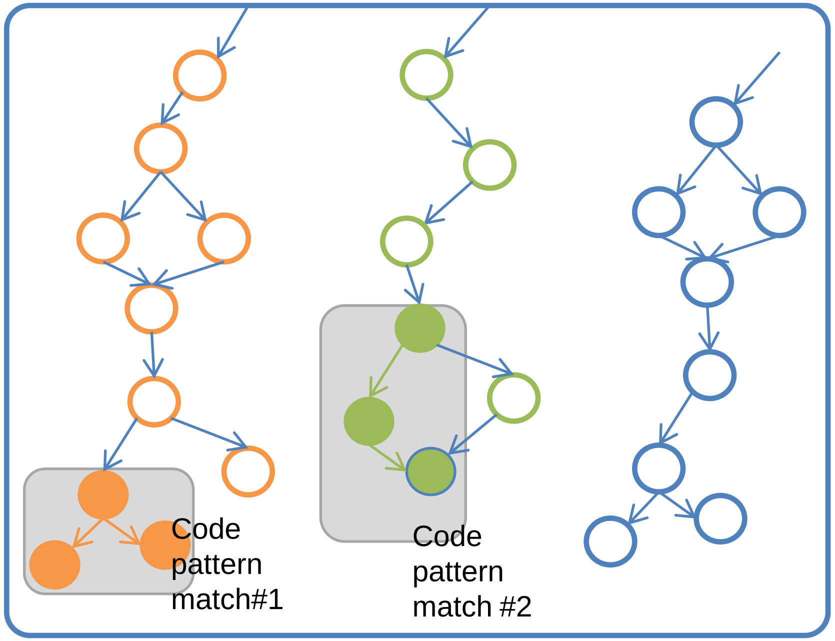
COBET Framework

- We build a COncurrency Bug pattErn maTching framework (COBET) to support programming template for effective bug pattern detector generation upon EDG C/C++ parser.

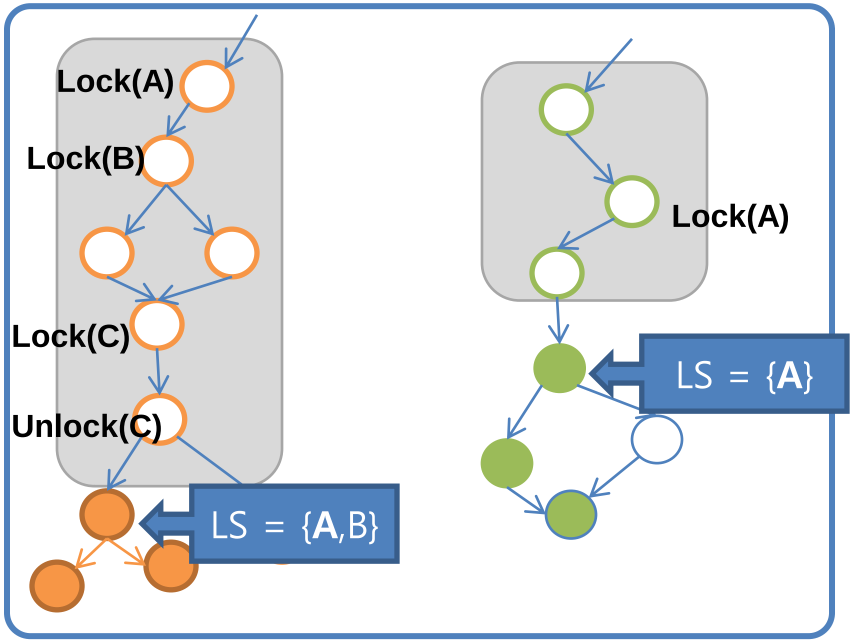


Semantic Information

- **Thread sensitive analysis**



- **Lock analysis**



- **Simple points-to analysis**

```
proc_alloc_inode() {
    ...
  ei = kmem_cache_alloc( ...
  if (!ei) return NULL ;
  ei->pid = NULL ;
    ...
```

Non-shared: {}

Non-shared: {ei}

Non-shared: {ei}

No other thread can access ei->pid.

Experiment Result

- Bug pattern matching result: Buggy Test and Test-and-Set pattern
  - 9 file systems in Linux kernel 2.6.30.4.
  - Every file system code is analyzed together with virtual file system code

	Ext2	Ext3	Ext4	NFS	Reiser FS	Proc	SysFS	UDF	BtrFS	Total	
Syntactic	13	19	18	15	18	18	12	15	17	145	
+ Multiple	9	14	14	9	13	11	7	10	14	101	**70%**
+ Multiple + Lockset	9	11	11	9	13	11	7	10	13	94	**65%**
+ Multiple + Points-to	6	11	11	6	10	8	4	7	11	74	**51%**
+ Multiple + Lockset + Points-to	6	8	8	6	10	7	4	7	9	65	**45%**

Conclusion

- Current progress
  - We detect and confirmed 8 unrevealed bugs from a recent Linux

Location	Bug patterns
Device Drivers (mtd/ubi)	Unsynchronized Data Passing to Child Thread
File Systems (btrfs)	Unsynchronized Data Passing to Child Thread
Device Drivers (scsi/qla4xxx)	Use Atomic Instructions in Non-atomic Ways
Network Stacks (atm)	Buggy Test and Test-and-Set
Network Stacks (ax25)	Buggy Test and Test-and-Set
Network Stacks (netfilter/ipvs)	Use Atomic Instructions in Non-atomic Ways
Network Stacks (rds)	Use Atomic Instructions in Non-atomic Ways
File Systems (btrfs)	Waiting Already Finished Threads

- Further works
  - Formal pattern description defining over both syntax and semantics
  - Utilizing analysis results for further programming and analysis
  - Apply mining techniques to associate similar bug reports in order to assist pattern extraction.