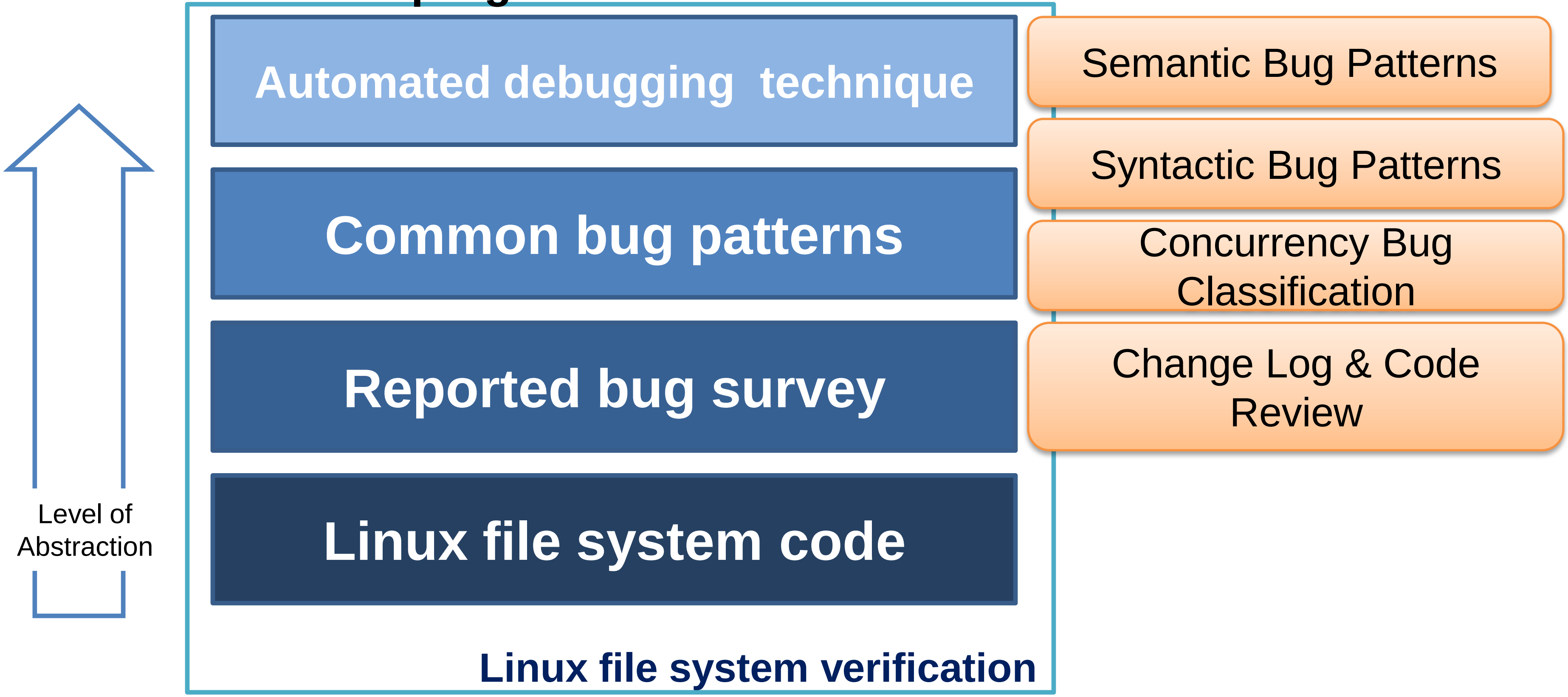
Concurrency Bug Detection through Improved Pattern Matching Using Semantic Information

Hong,Shin

Provable Software Laboratory
CS Dept. KAIST

Overview

- Goal: **Code-based automatic concurrency bug detection for large size concurrent programs**



Automated debugging technique

Common bug patterns

Reported bug survey

Linux file system code

Linux file system verification

Level of Abstraction

Semantic Bug Patterns

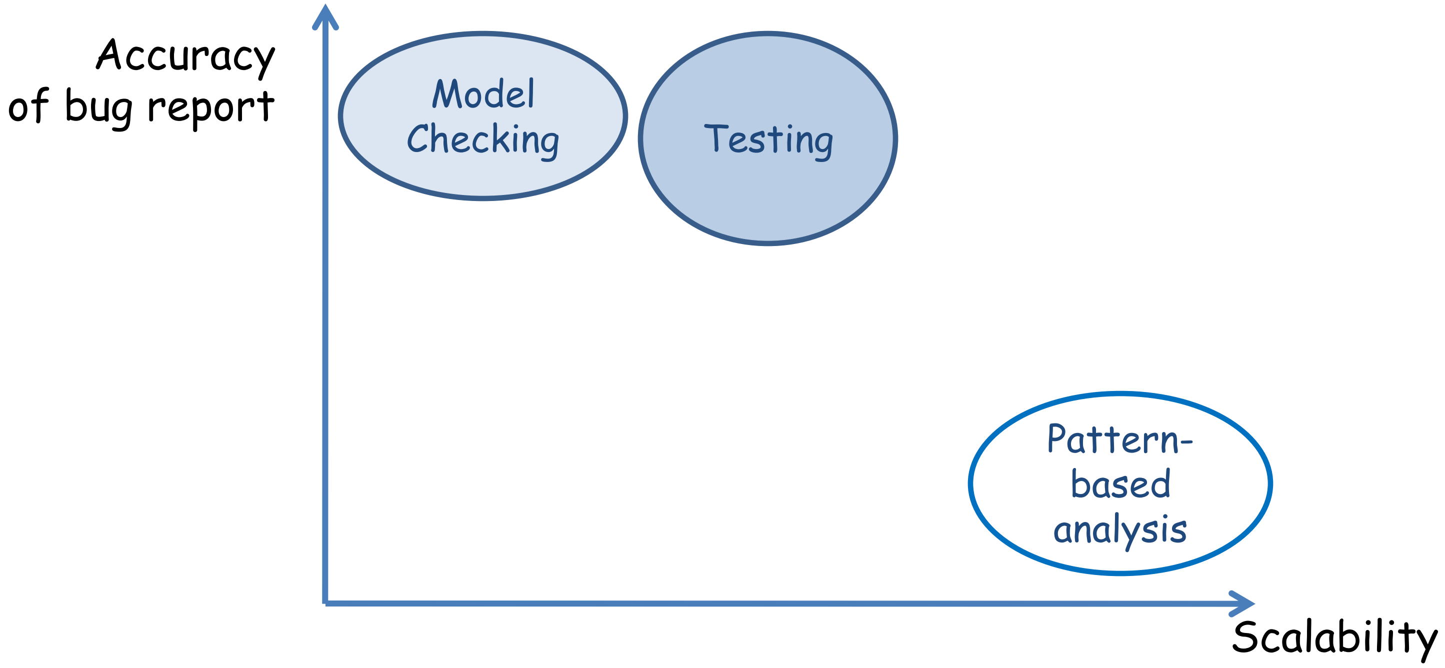
Syntactic Bug Patterns

Concurrency Bug Classification

Change Log & Code Review

- Idea: **Automatically detect common concurrency bugs using both syntactic and semantic code pattern matching**

Introduction



- Pattern-based analysis
  - Pros:
    - No state explosion problem
    - Early detection is possible(High coverage, Architecture-specific codes)
    - No manual abstract model construction
  - Cons:
    - Pattern descriptions
    - False alarm

Concurrency Bug Classification 1/2

- Example – Bug02

Bug02 (reported in Linux change log 2.6.6 /fs/dcache.c)

- Symptom: Data race (inconsistent data structure)

  The statistics information of the number of unused dentry might be smaller than actual number.

- Cause: Inconsistency
- Fault: Violate synchronization idioms

  The programmer assumed that “dentry->d_count” is protected by “dcache_lock”.

  But this assumption is incorrect. “dentry->d_count” is not protected by any lock.

  “denty->d_count” must be protected by atomic_inc_and_test().

- Synchronization mechanism: Atomic instruction
- Lock granularity: Inode

Concurrency Bug Classification 2/2

- Symptom / Cause

Cause Symptom	Performance improvement	Inconsistency	Incompleteness
Data race (Machine exception)		Bug09, Bug04, Bug19	Bug11, Bug15
Data race (Inconsistent data structure)		Bug01, **Bug02** Bug16, Bug18	Bug13, Bug14, Bug21, Bug22, Bug07
Deadlock		Bug24,Bug25	Bug23
Livelock	Bug03		

- Lock granularity / Synchronization mechanism

Lock granularity Sync. Mechanism	Kernel	File system	File	Inode
Instruction	**Bug02**		Bug18, Bug11	
Barrier		Bug14		
Thread creation/join			Bug13	
Mutex	Bug03, Bug15, Bug21, Bug22, Bug23, Bug25	Bug01, Bug07, Bug09		Bug19
Semaphore		Bug24	Bug16	
Readers/writer lock			Bug04	

Concurrency Bug Patterns 1/5

- Based on the analysis result of previously reported concurrency bugs, we construct bug patterns in order to detect similar unrevealed bug automatically.

- We abstract 6 bug patterns:
  (1) Misused of "Test and Test-and-Set"
  (2) Unexpected BKL releasing
  (3) Unlock before I/O operations
  (4) Absence of "get"/"put" function invocations
  (5) Unsynchronized communication at thread creations

- We formalize two bug patterns and then construct automatic bug detection tool using syntactic analyzer(parser) for the two bug patterns.
  - The tools detects suspected bugs in recent Linux file system codes.
  - The tools are built on EDG C/C++ front-end.

Concurrency Bug Patterns 2/5

- **Misused "Test and Test-and-Set" bug pattern**

```
int data;                    /*shared data*/

int func(){
  if (test(data)) {      /*test*/
    lock () ;
    if (test(data)) {   /*test&set*/
      data = newvalue ;
    }
    unlock () ;
   }
}             /* Correct Code */
```

```
int data;              /*shared data*/

int func(){
  if (test(data)){        /*test*/
    lock() ;
    data = newvalue ;
    unlock() ;
  }
}
           /* Buggy Code */
```

- **Related bug in Linux Kernel 2.6.11.10. (Bug07 in the bug list)**

  `/fs/ext3/balloc.c :: void ext3_discard_reservation()`

```
void ext3_discard_reservation(struct inode *inode)
{
      if(!rsv_is_empty(&rsv->rsv_window))
      {
       /*if (!rsv_is_empty(&rsv->rsv_window)) must be here */
            spin_lock(rsv_lock);
            rsv_window_remove(inode->i_sb,rsv);
            spin_unlock(rsv_lock);
      }
}
```

Concurrency Bug Patterns 5/5

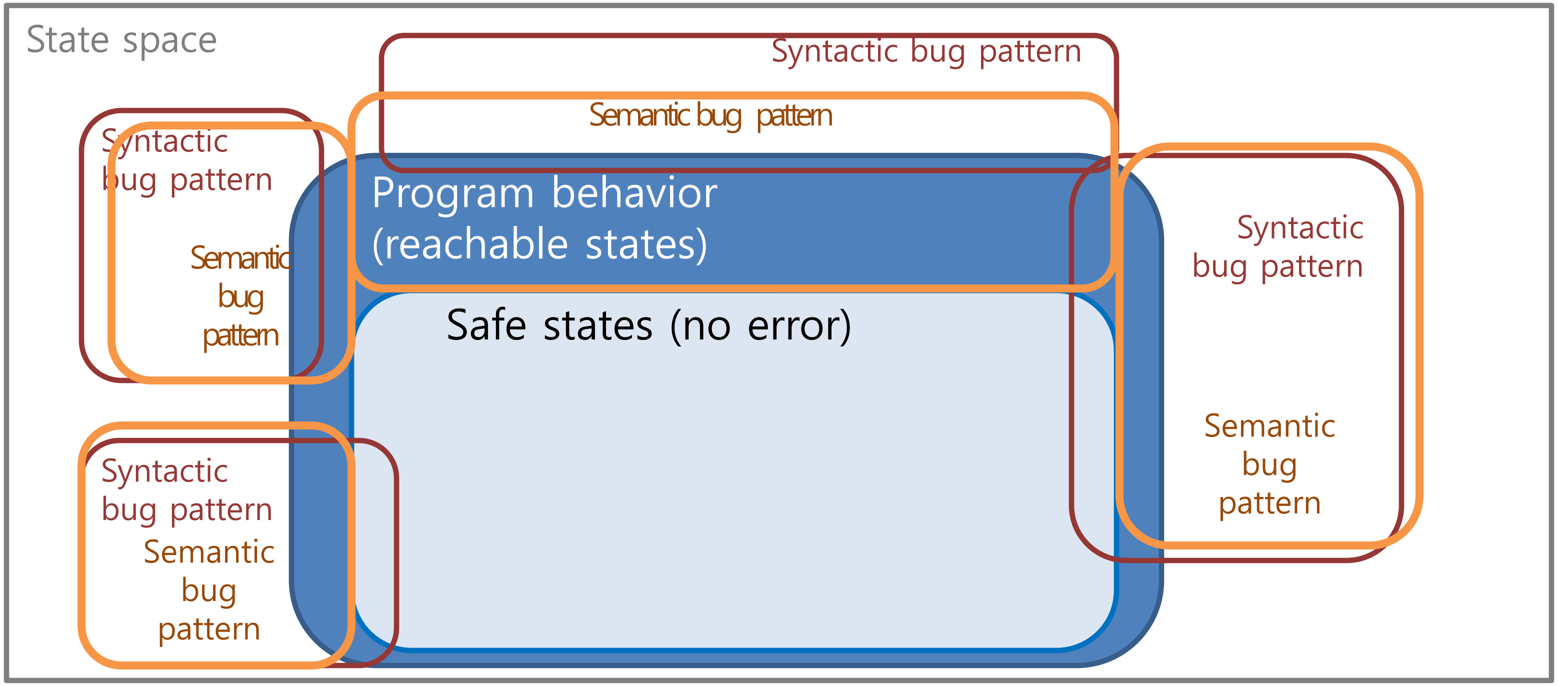
- **Bug detection result (Linux 2.6.26)**
  - We verifies 8 naive file systems in Linux using the bug pattern detection tool. In the verification, we found total 35 bug candidates.

	Ext2	Ext3	Ext4	NFS	ReiserFS	Proc	Sysfs	UDF	Total
# of suspected bugs	**2**	**3**	**6**	**11**	**7**	**1**	**1**	**4**	**35**
# of files	18	24	23	34	26	20	9	17	171
LOC	28K	104K	25K	56K	27K	18K	37K	9K	304K
Time(sec)	2.338	3.849	4.306	6.420	4.454	2.525	1.029	2.689	27.610

```
/* In Linux kernel 2.6.26 /fs/nfs/nfs4state.c */
static void nfs4_drop_state_owner(struct nfs4_state_owner *sp)
{
    if (!RB_EMPTY_NODE(&sp->so_client_node)) {
        struct nfs_client *clp = sp->so_client;
        spin_lock(&clp->cl_lock);
        rb_erase(&sp->so_client_node, &clp->cl_state_owners);
        RB_CLEAR_NODE(&sp->so_client_node);
        spin_unlock(&clp->cl_lock);
    }
}
```

Improved Bug Pattern Matching 1/6

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

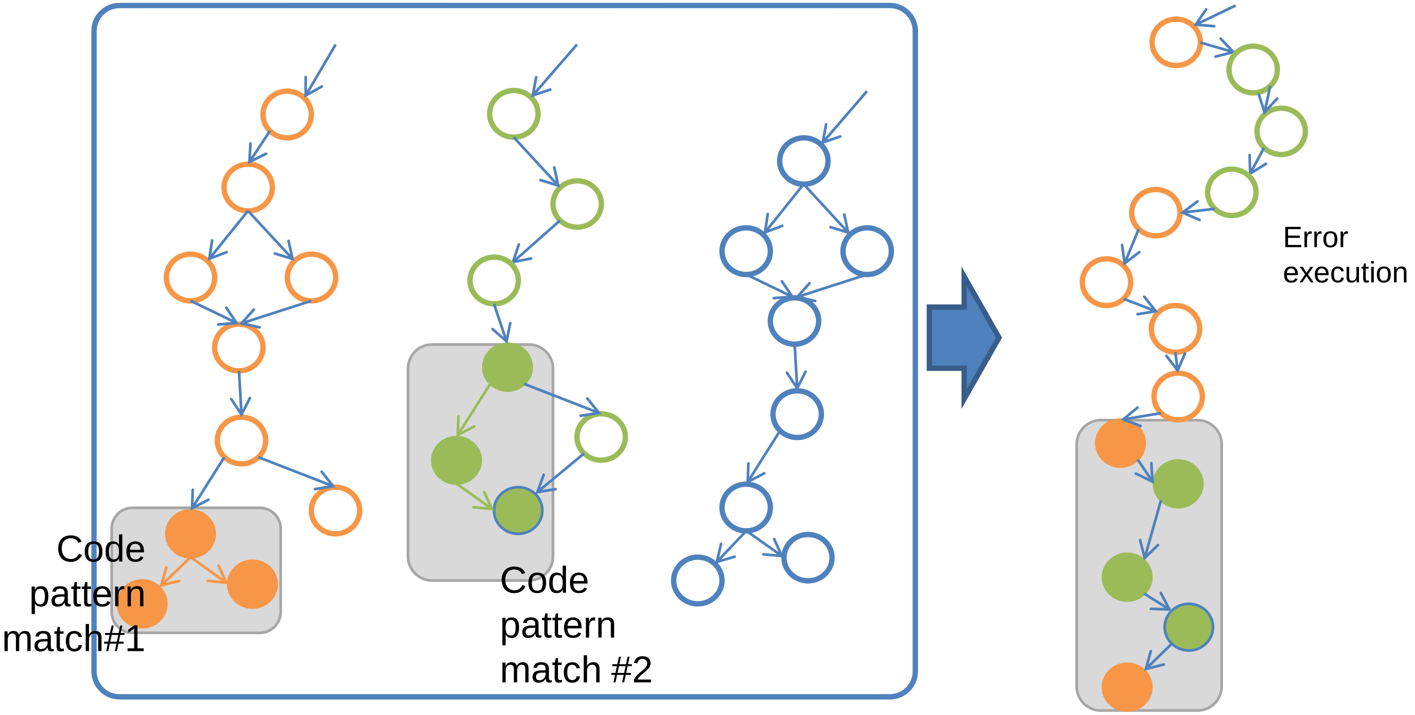


- We validates the bug detection results by syntactic bug pattern matching manually. There are the following main sources of false positives.
  - No parallel thread to race ➔ **Multiple code pattern matching**
  - Synchronized by other locks ➔ **Lock analysis**
  - Shared variable initializations without holding locks ➔ **Simple points-to analysis**

Improved Bug Pattern Matching 2/6

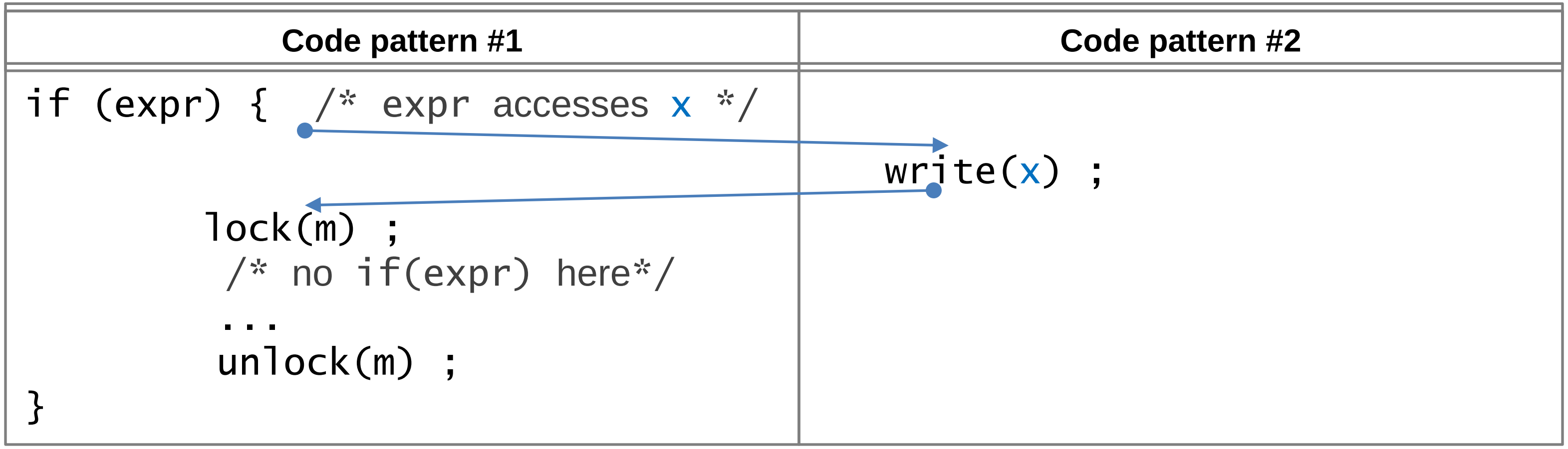
Multiple code pattern matching (thread sensitive analysis)

- There are at least two thread executions in one concurrency error execution.
- We extend the syntactic bug patterns to match multiple code patterns for a bug detection rather than single code pattern matching.
  - The syntactic bug pattern match would be false positive if there is no code which can be executed concurrently to generate concurrency errors.
  - We assume that two pointers of the same type may point to the same memory address.



Improved Bug Pattern Matching 3/6

- Extended misused “Test and Test-and-Set” bug pattern



Code pattern #1

```
if (expr) {  /* expr accesses x */


        lock(m) ;
         /* no if(expr) here*/
         ...
         unlock(m) ;
}
```

Code pattern #2

```
write(x) ;
```

Ex. Linux 2.6.26 /fs/proc

Code pattern #1

- **Match 1-1**

```
proc_get_sb() {
          …
  ei = PROC_I(sb->s_root->d_inode);
  if (!ei->pid) {
    rcu_read_lock();
    ei->pid = get_pid(find_pid_ns(1,ns));
    rcu_read_unlock();
  }
          …
}
```

Code pattern #2

- **Match 2-1**

```
proc_get_sb() {
          …
  if (!ei->pid) ei->pid = find_get_pid(1) ;
```

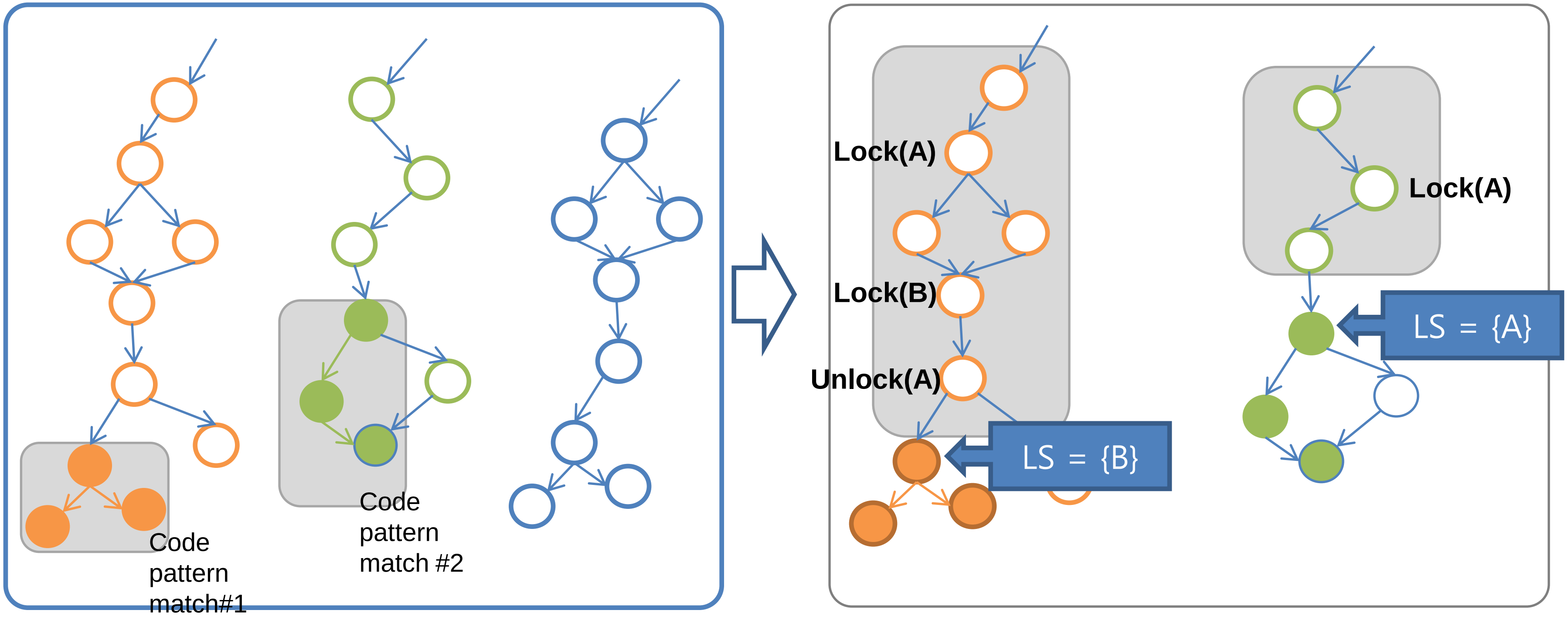
- **Match 2-2**

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

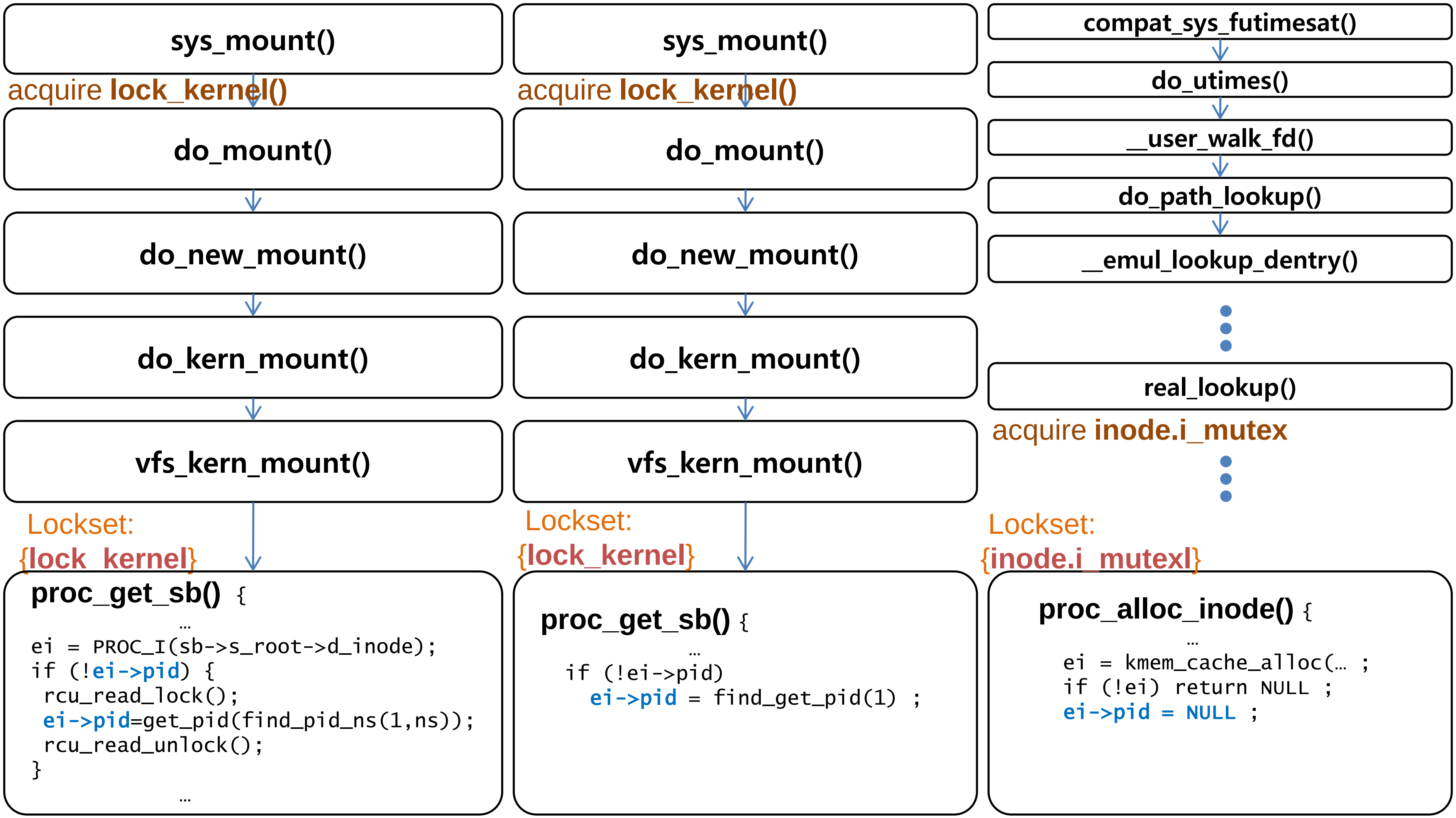
Improved Bug Pattern Matching 4/6

Lock analysis

- We applied lock analysis to compute the set of locks held when a code pattern match is executed. This approach is similar to RacerX approach.
  - Inter-procedural, flow-sensitive, path-insensitive, alias-insensitive analysis
  - We assume that the specification of lock acquiring/releasing functions are specified.
  - We assume that a thread can be started from one of system call routines.
  - We assume that two lock variables of the same type can point to the same memory address.



Improved Bug Pattern Matching 5/6



- Code pattern match 1-1

sys_mount()

acquire **lock_kernel()**

do_mount()

do_new_mount()

do_kern_mount()

vfs_kern_mount()

Lockset:
{lock_kernel}

```
proc_get_sb() {
        …
ei = PROC_I(sb->s_root->d_inode);
if (!ei->pid) {
 rcu_read_lock();
 ei->pid=get_pid(find_pid_ns(1,ns));
 rcu_read_unlock();
}
        …
```

- Code pattern match 2-1

sys_mount()

acquire **lock_kernel()**

do_mount()

do_new_mount()

do_kern_mount()

vfs_kern_mount()

Lockset:
{lock_kernel}

```
proc_get_sb() {
          …
 if (!ei->pid)
   ei->pid = find_get_pid(1) ;
```

- Code pattern match 2-2

compat_sys_futimesat()

do_utimes()

__user_walk_fd()

do_path_lookup()

__emul_lookup_dentry()

⋮

real_lookup()

acquire **inode.i_mutex**

⋮

Lockset:
{inode.i_mutexl}

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

Improved Bug Pattern Matching 4/6

Points-to analysis

- Many programmers initialize a newly allocated heap variable before it becomes shared variable without any synchronization. However, this initialization is recognized as buggy code in the bug pattern matching for the lack of alias-sensitive analysis.

Ex. Match 2-2

```
0:  proc_alloc_inode() {
1:      ...
2:    ei = kmem_cache_alloc(proc_inode_cachep,…;
3:    if (!ei) return NULL ;
4:    ei->pid = NULL  ;
        ...
```

$\texttt{kmem\_cache\_alloc} \in fmalloc$

$\nexists\, j\ (0 < j < 4).\ \texttt{y = ei ;}$

- We apply simple intra-procedural points-to analysis to remove the false positives caused by unshared heap variable initialization.
  - We assume that dynamic memory allocation functions are specified as $fmalloc$ (e.g. `kmalloc()`)
  - We assume that a newly allocated heap variable becomes shared when its address is assigned to other shared variable.
  - The algorithm for checking initialization without locking is as follow.
    For each function $func$ where a code pattern matches, access to a variable x at location l cannot be involved in concurrency error if
    1. There exists $x = alloc()$ where $alloc \in fmalloc$ at location i where $0 \leq i < l$, and
    2. There is no $y = x$ at location j where $i < j < l$ and y is expected to be a shared variable.

Improved Bug Pattern Matching 6/6

Bug detection result

	Ext2	Ext3	Ext4	NFS	ReiserFS	Proc	Sysfs	UDF	**Total**
# of suspected bugs by **syntactic bug pattern matching**	**2**	**3**	**6**	**11**	**7**	**1**	**1**	**4**	**35**
# of suspected bugs by **improved bug pattern matching**	**1**	**3**	**3**	**7**	**2**	**0**	**0**	**3**	**19**
# of files	18	24	23	34	26	20	9	17	171
LOC	546K	619K	708K	737K	718K	442K	490K	553K	4813K

Further improvement- Checking exclusive path conditions

- Extract a set of syntactic path conditions for execution path to each code pattern match
- For two code pattern matches m_1 and m_2 , it may be false positive if

$$\exists c_1, c_2. \quad c_1 \in PC_{m1} \land c_2 \in PC_{m2} \land exclusive(c_1, c_2)$$

- $exclusive$() compares c_1 and c_2 to a set of syntactic templates of exclusive conditions.

e.g. exclusive(x,y) is true for following cases:

(“a == b” , “a != b”), (“!a”, “a”), (“a>b”, “a<b”), (“a>b”, “b>a”),
(“a && b”, “!a”), (“a && b”, “!b”), (“!(a||b)”, “a”), (“!(a||b)”, “b”) , etc.

Related work

- Pattern-based bug detection
  - MetaL (by D. Engler et al. @ Stanford)
  - FindBugs (by W.Pugh et al. @ Univ.Maryland)
  - ConTest (by IBM Haifa Lab.)
  - Learning from mistakes (by Opera group @ UIUC)

- Static concurrency bug detection
  - RacerX (by D. Engler et al. @ Stanford): Static analysis to check lock discipline(data race) and lock ordering constraints (deadlock)
  - RELAY (by R. Jhala et al. @ UCSD): Scalable, lock-sensitive analysis tool for detection data race bugs from concurrent C programs with false alarm restriction heuristics.