Effective Pattern-driven Concurrency Bug Detection for Operating Systems

Shin Hong, Moonzoo Kim*

Computer Science Department, KAIST, South Korea

Abstract

As multi-core hardware has become more popular, concurrent programming is being more widely adopted in software. In particular, operating systems such as Linux utilize multi-threaded techniques heavily to enhance performance. However, current analysis techniques and tools for validating concurrent programs often fail to detect concurrency bugs in operating systems (OS) due to the complex characteristics of OSes. To detect concurrency bugs in OSes in a practical manner, we have developed the COncurrency Bug dETector (COBET) framework based on *composite bug patterns* augmented with *semantic conditions*. The effectiveness, efficiency, and applicability of COBET were demonstrated by detecting ten new bugs in file systems, device drivers, and network modules of Linux 2.6.30.4 as confirmed by the Linux maintainers.

Keywords: Concurrency Bug, Bug Pattern, Static Analysis, Linux

1. Introduction

As multi-core hardware becomes increasingly powerful and popular, operating systems (OSes) such as Linux utilize the cutting-edge multi-threaded techniques heavily to enhance performance. However, current analysis techniques and tools for concurrent programs have limitations when they are applied to operating systems due to the complex characteristics of OSes. In particular, the following three characteristics of OSes make concurrency bug detection on OSes difficult.

- *Various synchronization mechanisms utilized*
  Most concurrency bug detection techniques [1, 2, 3, 4, 5, 6] focus on lock usage, since a majority of user-level applications utilize simple mutexes/critical sections to enforce synchronization. However, OSes exploit

*Corresponding author
Email addresses: `hongshin@kaist.ac.kr` (Shin Hong), `moonzoo@cs.kaist.ac.kr` (Moonzoo Kim)

various synchronization mechanisms (see Table 1) for performance enhancement.

- *Customized synchronization primitives*
  OS developers sometimes implement their own synchronization primitives. Thus, concurrency bug detection tools for standard synchronization mechanisms do not recognize these customized synchronization primitives and produce imprecise results [7].

- *High complexity of operating systems*
  A dynamic analysis (i.e., testing) often fails to uncover hidden concurrency bugs due to the exponential number of possible interleaving scenarios between threads in OSes. In addition, replaying bugs is difficult, since it is hard to manipulate thread schedulers in OSes directly. A static analysis, on the other hand, has limited scalability to analyze OS code due to its high complexity and complicated data structures. Furthermore, the monolithic structure (i.e., tightly coupled large global data structure) of OSes severely hinder modular analyses.

For these reasons, in spite of much research on concurrent bug detection (see Section 6), such techniques have seldom been applied to OS development in practice.

To alleviate the above difficulties, we have developed the COncurrency Bug dETector (COBET) framework, which utilizes *composite bug patterns* augmented with *semantic conditions*. Note that concurrency errors are caused by unintended interference between multiple threads. A salient contribution of COBET is that it utilizes multiple sub-patterns, each of which represents a buggy pattern in one thread, and checking semantic information that determines possible interferences between multiple threads in a precise and scalable manner (see Section 3). In addition, since engineers who use COBET can define various concurrency bug patterns, COBET can detect concurrency bugs that are not targeted by lock-based concurrency bug detection tools.

One drawback of COBET is that a user has to identify and define bug patterns, which requires insight on corresponding bug patterns and knowledge of the target code. However, once such bug patterns are defined, corresponding pattern detectors can be implemented to detect concurrency bugs in (1) subsequent releases of the target program, and/or (2) other modules in a similar domain. It has been frequently observed that although a given bug had been fixed previously, similar bugs often appeared in the subsequent releases or in the different modules of the target program (see Section 5.1 and Section 5.3). Thus, initial efforts to define bug patterns could be sufficiently rewarded by detecting concurrency bugs in rapidly evolving large software systems such as Linux. Furthermore, to lessen the effort to define bug patterns and construct corresponding bug pattern detectors, the COBET framework provides a pattern description language (PDL)(see Section 3.2).

Currently, COBET provides four concurrency bug patterns that are identified based on a review of Linux kernel ChangeLog documents The effectiveness of COBET was demonstrated by detecting ten new bugs in file systems, network modules, and device drivers of Linux 2.6.30.4 (the latest Linux release at the moment of the experiments), which were confirmed by Linux maintainers.

The contributions of this research are as follows:

- We have derived interesting observations on the Linux concurrency bugs from a review of the Linux ChangeLogs documents on Linux 2.6.x releases (Section 2).
- We have developed a pattern-based concurrency bug detection framework, which can define and match various bug patterns. To improve bug detection precision, our framework utilizes composite patterns with semantic conditions in a scalable manner (Section 3).
- Based on previous bug reports, we have defined four concurrency bug patterns with various synchronization mechanisms, which are effective to detect new bugs in Linux that are not targeted by lock-based analysis techniques. (Sections 4-5).

The remainder of this paper is organized as follows. Section 2 describes the characteristics of Linux to show the advantages of pattern-based bug detection approach on Linux. Section 3 overviews the COBET framework. Section 4 explains composite bug patterns with semantic conditions upon the COBET framework. Section 5 reports the evaluation of the COBET framework through the empirical results on Linux kernel. Section 6 discusses related work. Finally, Section 7 concludes the paper.

2. Characteristics of Linux Operating System

In this section, we describe the characteristics of concurrent programming practices used in Linux.

2.1. Synchronization Mechanisms in Linux

Linux utilizes various synchronization mechanisms for enhanced performance. We gathered statistics on the nine standard synchronization mechanisms in the entire kernel code of Linux 2.6.30.4. These nine synchronization mechanisms include atomic instructions, conditional variables, memory barriers, mutexes, read/write semaphores, read/write spin locks, semaphores, spin locks, and thread operations (e.g., thread creation, join, etc). The Linux kernel consists of around 8.67 million lines of C code. Those synchronization mechanisms are identified in target code by the name of the corresponding library function calls.

Table 1 shows the numbers of statements for the nine synchronization mechanisms. Locks, the most popular synchronization mechanism, can be implemented by using spin locks, mutexes, and binary semaphores. Thus, locks take 75∼76% (= 56.1% + 18.9% + 0∼1.0%) of all synchronization statements in the Linux kernel code. Consequently, 24∼25% of synchronization statements cannot be examined by lock-based bug detection techniques.

Table 1: Statistics on the Synchronization Statements in the Linux Kernel 2.6.30.4

	atomic inst.	cond. var.	memory barrier	mutex	rw semaphore	rw spin lock	semaphore	spin lock	thread operation	total
# of stmt.	8926	949	1926	14902	2471	4248	759	44205	460	78846
Ratio	11.3%	1.2%	2.4%	18.9%	3.1%	5.4%	1.0%	56.1%	0.6%	100.0%

2.2. Survey of the Linux Bug Reports

We reviewed 324 ChangeLogs on Linux 2.6.0 to 2.6.30.3 to understand the nature of real concurrency bugs (as Lu et al. [8] did on large application programs) and identify concurrency bug patterns accordingly. We concentrated on the bug reports related to Linux file systems for the following three reasons. First, file systems utilize heavy concurrency to handle multiple I/O transactions simultaneously. Thus, we expected that file systems had many concurrency issues. Second, the Linux file system implementations are referenced in many software systems. Thus, the observations on concurrency bugs for the Linux file systems can be applied to many other programs. Moreover, there are relatively rich reference documents on the Linux file systems, so that it is easy to understand the bug reports and define bug patterns. Third, compared to other parts of Linux, file systems are stable and diverse. As the file system codes are actively evolved and maintained by Linux developers, we expect that there exist rich and quality bug reports. In addition, each file system module exploits unique characteristics. Thus, we expected that the bug reports would not be biased by specific style of program characteristics or programming style. We collected the concurrency bug reports on the Linux file systems by searching related keywords (i.e., ‘lock’, ‘concurrency’, ‘data races’, ‘deadlock’, etc.) as well as manual inspection. Finally, we found 50 concurrency bug reports on the Linux file systems and 27 of them were selected for in-depth review (the remaining 23 bugs were discarded, since these bugs were caused by domain-specific requirement violations or could not be understood concretely). Through the review, we made the following observations:

Observation 1: *Half of the concurrency bugs involved with synchronization mechanisms other than locks.* 12 of the 27 bugs were associated with synchronization mechanisms other than locks (i.e., atomic instructions, memory barriers, thread operations, etc.). In addition, locks were sometimes used in a non-standard manner (e.g., recursive locking, releasing on blocking, etc.). This observation indicates that we need customizable/flexible concurrency bug detection tools that can analyze various synchronization mechanisms, not only standard lock usages.

Observation 2: *Code review was more effective to detect concurrency bugs than runtime testing was.* Linux ChangeLogs reported that, among the 27 concurrency bugs, nine were detected by actual testing and 13 bugs detected by manual code review (the sources of the remaining five bugs were not clear). In general, code review does not reason with concrete input data and schedul-

Table 2: Statistics on the Call Sequences of the Seven Linux File Systems

	`btrfs`	`ext4`	`nfs`	`proc`	`reiserfs`	`sysfs`	`udf`	
Lines of code	41KL	28KL	29KL	8KL	27KL	3KL	9KL	Total 145KL
# of call sequences	2100M	1501M	3394M	12M	13413M	1M	51M	Total 20488M
Max. length of call seq.	88	54	57	33	55	26	43	Avg. 51
Avg. length of call seq.	60	43	39	25	38	23	35	Avg. 38

ing, but by reading code statically. Note that pattern-based concurrency bug detection also has this characteristics of the code review.

Observation 3: *Linux kernel code was updated frequently.* For six years, 324 Linux releases (including major releases and minor releases) have been made. This means that a new Linux kernel has been released on average every week. In addition, on average, 3.83 patches have been applied to Linux 2.6.X releases per hour [9]. Furthermore, the Linux kernel has been constantly growing up from 2.6.11 release (6.6 million lines of code in 17090 files) to 2.6.30 release (11.6 million lines of code in 27911 files) [9]. Thus, we need a light-weight bug detection framework that can analyze a large program quickly and conveniently.

2.3. Complexities of Linux File Systems

To estimate the complexities of the Linux file systems, we counted the number of different call sequences by traversing the inter-procedural control flow graphs (CFG) for the seven Linux file system codes, including `btrfs`, `ext4`, `nfs`, `proc`, `reiserfs`, `sysfs`, and `udf`. The number of call sequences can serve as a measure for the complexity of the file system, since each call sequence represents a unique execution scenario. Each file system is analyzed together with Virtual File System (VFS) code.

Table 2 describes the statistics on call sequences of the seven Linux file systems. To analyze all of these file systems (145KL, i.e., 145 thousand lines of C codes), we had to analyze 20 billion different execution scenarios whose average call depth is around 38 (see the last column of Table 2). Furthermore, due to non-deterministic scheduling, the total number of concurrent execution scenarios is exponential in the number of sequential execution scenarios in Table 2. Thus, it is clear that, due to the huge number of execution scenarios, achieving high coverage by testing and/or model checking is infeasible. Therefore, light-weight analysis techniques should be developed for complex operating systems like Linux.

3. COBET Framework

The observations in Section 2 suggest that a pattern-based concurrency bug detection framework can be a practical solution for Linux. Thus, we have developed the COncurrency Bug dETector (COBET) framework for concurrent C programs based on a pattern matching approach.

3.1. Overview of the COBET Framework

The overall structure of the COBET framework is depicted in Figure 1. Since concurrency errors are caused by unintended interferences among multiple concurrent threads, a concurrency bug pattern should be specified as multiple sub-patterns each of which captures a specific code running on each thread. For this purpose, COBET provides a pattern description language (PDL) to describe the syntactic structure of a bug pattern (see Section 3.2). The COBET synthesizer inputs a user-specified bug pattern description in PDL and generates a corresponding bug pattern detector. A synthesized bug pattern detector contains the following four components:

- syntactic pattern matcher
- semantic condition checker
- semantic analysis engine
- abstract syntax tree (AST) generator

A *syntactic pattern matcher* in a generated bug pattern detector detects segments of a target program code that match sub-patterns in the given PDL description. Then, a *semantic condition checker* checks whether or not these code segments can run concurrently and interfere with each other through a *semantic analysis engine*. For this purpose, the semantic analysis engine performs

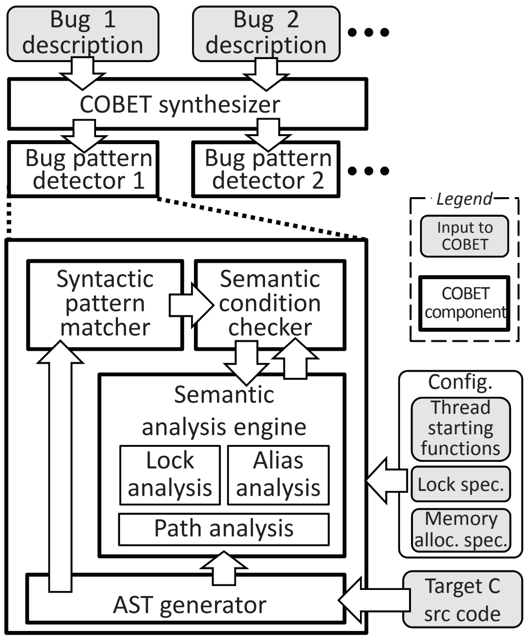


Figure 1: Overview of the COBET framework

path analysis, lock analysis, and alias analysis (see Section 3.3). For these analyses, a user should provide configurations of a target program, which include names of thread starting functions, specifications of lock/unlock operations (e.g., `spin_lock()`, `spin_unlock()`, etc.), and specification of memory allocation operations (e.g., `kmalloc()`, `kmem_cache_alloc()`, etc.). Different target domains may have different configurations. At the lowest layer, the AST generator parses and creates the AST of a target program using the EDG parser [10]. The AST of a target program is used by the syntactic pattern matcher to detect code segments that match bug patterns syntactically.

The COBET framework consists of 4500 lines of C code in 96 functions. The COBET framework uses GCC 4.3.0 to preprocess a target code and EDG C/C++ Front-End 3.1 to parse the preprocessed code.

3.2. Bug Pattern Detectors

The COBET synthesizer constructs bug pattern detector code from a user-given bug pattern specification. The bug pattern detector performs the syntactic pattern matching and calls `sem_cond_checking()` to check relevant semantic conditions.

A *pattern description language (PDL)* is designed to help engineers define bug patterns in a correct and convenient manner. Figure 2 shows the brief grammar of PDL. In PDL, a concurrency bug pattern is described as a set of *sub-patterns* (line 1 of Figure 2), each of which specifies target code running on one thread. A sub-pattern contains one or more *function descriptions* (line 2). A function description consists of *abstract statement descriptions* (line 3). An abstract statement description (lines 4-10) is specified with a keyword (e.g., `if`, `loop`, `lock`, `read`, etc.) which indicates a type of target code statement to match. A bug pattern detector based on PDL searches target code to find matched code statements while ignoring irrelevant code statements. In addition, PDL can describe a bug pattern by using $\backslash\{\text{Stmt}^{+}\}$ (exclusion), which specifies statements that should not appear in pattern matching instances. In PDL, `$`<*name*> is a untyped free variable that binds a corresponding code element in a target C statement. For example, a pattern description `if $cond {write $var;}` can match `if (x<0) {x=f(x); y= x*x; }` and generates the following two pattern matching instances. For the first matching instance, `$cond` and `$var` are bound to `x<0` and `x` in a target code, respectively. For the second instance, `$cond` and `$var` are bound to `x<0` and `y`, respectively. Free variables of PDL are used to describe subtle conditions in a bug pattern.

The syntactic pattern matching of a PDL description to a target code follows the naive tree pattern matching algorithm [11]. The parser generates a target C function into an abstract syntac tree. The COBET synthesizer interprets a PDL description as an ordered tree where a node can have a parameter. The syntactic pattern matcher maps each abstract statement to a C statement and each free variable to a C expression.

To check semantic conditions on a pattern matching instance, pattern detector code invokes `sem_cond_checking()` at every syntactic matching/binding step. As a default, `sem_cond_checking()` checks feasibility of interference among

```
Bug-pattern ::= Sub-pattern+
Sub-pattern ::= pattern constant {Function+}
   Function ::= fun Identifier {Stmt+}
       Stmt ::= if $cond {Stmt*}
              | if $cond {Stmt*}    else {Stmt*}
              | loop $cond {Stmt*}  | break;
              | lock Identifier;     | unlock Identifier;
              | read Identifier;     | write Identifier;
              | call Identifier $args;| \{Stmt+}
              | ...
 Identifier ::=constant | $⟨name⟩
```

Figure 2: Brief grammar of the COBET pattern description language

code segments that match sub-patterns running on multiple threads (e.g., the sets of held locks at different sub-patterns must be disjoint (see Section 3.3)). In addition, a user can add sophisticated semantic condition checking routines to this function, since synthesized pattern detector code is human-readable. For this purpose, COBET provides library functions to check the semantic conditions in a bug pattern matching instance. Figure 6 shows one example of `sem_cond_checking()` for 'misused test and test-and-set' bug pattern (see Section 4.1).

3.3. Semantic Analysis Engine

The COBET semantic analysis engine checks whether or not multiple code segments that match specified sub-patterns can run concurrently and interfere with each other through *path analysis*, *lock analysis*, and *alias analysis*. The semantic analysis engine performs the path analysis first to explore inter-procedural execution paths reaching functions that match at least one syntactic sub-pattern. Then, lock analysis and alias analysis are performed on these execution paths. The path analysis, lock analysis, and alias analysis of COBET are similar to the techniques used in RacerX [2].

For example (see Figure 3), suppose that a bug pattern `b` consists of two sub-patterns, `b.1` and `b.2`. Also, assume that, through syntactic pattern matching, COBET detects that the target program has functions `f1()` and `f2()`, which contain statements $s_{b.1}$ and $s_{b.2}$ matching `b.1` and `b.2`, respectively. Then, the COBET semantic analysis engine statically explores execution paths starting from thread-starting functions to the target functions `f1()` or `f2()` (*path analysis*). Suppose that the COBET analysis engine finds that there is a lock to prohibit concurrent executions of `f1()` and `f2()` in all possible paths (*lock analysis*). Then, COBET concludes that `b` does not occur, since $s_{b.1}$ and $s_{b.2}$ cannot be interleaved and, thus, cannot interfere with each other. If there is no such lock to prevent interference between $s_{b.1}$ and $s_{b.2}$, then COBET checks

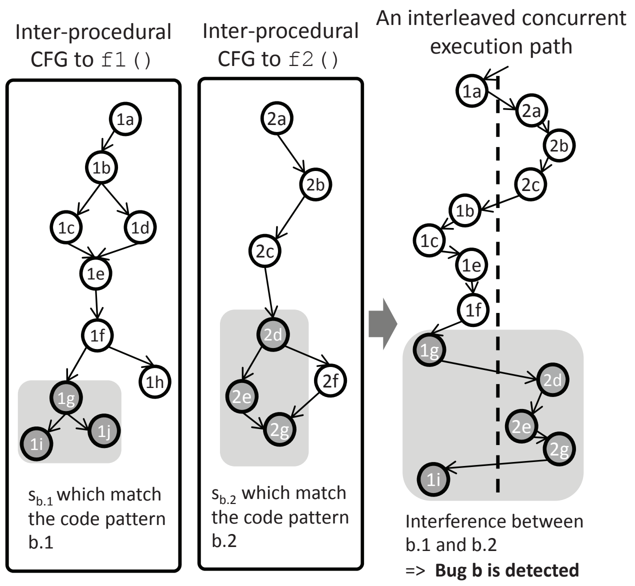

Inter-procedural CFG to f1()
Inter-procedural CFG to f2()
An interleaved concurrent execution path
1a
1b
1c
1d
1e
1f
1g
1h
1i
1j
2a
2b
2c
2d
2e
2f
2g
$s_{b.1}$ which match the code pattern b.1
$s_{b.2}$ which match the code pattern b.2
Interference between b.1 and b.2
=> Bug b is detected

Figure 3: Interference between two sub-patterns causing a concurrency error

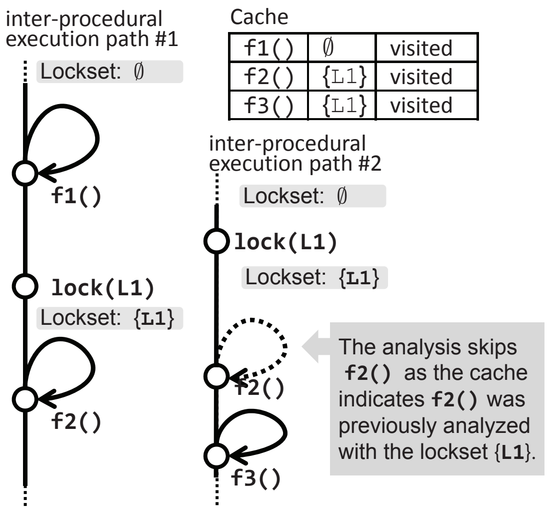
whether or not $s_{b.1}$ and $s_{b.2}$ can access the same variable causing interference (*alias analysis*). If the alias analysis result indicates that $s_{b.1}$ and $s_{b.2}$ can access the same shared variable, COBET reports that `b` is detected in the target program.

Path Analysis: COBET's path analysis generates an inter-procedural CFG from the AST of a target program. Then, the path analysis generates inter-procedural execution paths starting from thread starting functions to the functions that match specified sub-patterns by exploring the interprocedural CFG. Since exploration of all inter-procedural execution paths in the OS consumes a huge amount of time (see Section 2.3), COBET conducts syntactic pattern matching first to prune irrelevant execution paths.

If a target program has many function pointers, which are frequently used to link lower-layer modules to upper-layer modules in layered OS architecture, it is hard to generate an accurate inter-procedural CFG, since we do not know which functions will be called via function pointers. COBET solves this problem using the following heuristics. COBET constructs a function pointer table consisting of pairs of a function pointer and candidate functions which can be invoked through the function pointer, by analyzing assignment statements on global variables of function pointer type. The path analysis refers to the table when it reaches a function call via a function pointer. If there exist multiple candidate functions for a function pointer, the path analysis generates multiple paths to all these candidate functions in a conservative manner.

Lock Analysis: COBET's lock analysis formulates *locksets* (i.e., a set of held locks) at a code location. The lockset information is used to check whether or not two code locations are guarded by the same lock. The lock analysis obtains a lockset of a code location by exploring inter-procedural execution paths while recording lock acquiring operations and lock releasing operations. The lock analysis recognizes lock/unlock operations in a target program based on the lock operation function names specified in an input configuration to COBET. The technique concludes that two lock operations with parameters acquire the same lock if their parameters may alias each other. COBET performs path-insensitive lock analysis due to the high computational cost of path-sensitive lock analysis. For a branching statement, the lock analysis explores both branches and takes the union of the two locksets obtained from both branches as a result of the branching statement. For a loop statement, COBET analyzes only one iteration.

COBET uses an inter-procedural lock analysis. In other words, the technique transfers the lockset of a call site in a caller function to the analysis on the callee function. However, to prevent propagation of incorrect lock analysis results from a callee function, COBET does not reflect the locksets of exit statements in the callee function to the caller function [2].



inter-procedural execution path #1
Lockset: ∅
f1()
lock(L1)
Lockset: {L1}
f2()
Cache
f1() ∅ visited
f2() {L1} visited
f3() {L1} visited
inter-procedural execution path #2
Lockset: ∅
lock(L1)
Lockset: {L1}
f2()
f3()
The analysis skips f2() as the cache indicates f2() was previously analyzed with the lockset {L1}.

Figure 4: Caching in the lock analysis

To avoid redundant inter-procedural lock analysis, COBET semantic analysis engine uses a cache that records the lock analysis result for functions. When the analysis reaches a call site of `f()` with a lockset `LS`, COBET tries to find $\langle$`f()`, `LS`$\rangle$ in the cache. If the cache does not contain such an entry, the analysis continues to the callee function `f()`. Otherwise, the analysis does not go into `f()`. Figure 4 illustrates the cache operations. The lock analysis on the first execution path records $\langle$`f1()`,$\emptyset\rangle$, $\langle$`f2()`, $\{$`L1`$\}\rangle$ in the cache. The lock analysis on the econd execution path skips `f2()`, since the cache already contains $\langle$`f2()`, $\{$`L1`$\}\rangle$. This cache technique saves a large amount of lock analysis time in our experiments, achieving 2∼20 times speedup compared to non-caching lock analysis.

Alias Analysis : COBET's alias analysis statically examines whether or not two expressions may access the same shared variable. COBET utilizes the extended type information on variables (i.e., type of a variable and the type of a structure containing the variable as a field) for the analysis and considers two heap variables of the same type to be aliased.

COBET assumes that the following variables are non-shared: (1) local variables, and (2) dynamically allocated variables through function calls such as `kmalloc()`, `kmem_cache_alloc()`, etc. that are not assigned to global variables yet (i.e., performing 'uniqueness analysis' [12]). The technique traces pointer assignments and considers a variable that is assigned with a non-shared variable to be a non-shared variable, too. At a function call site, if actual parameters of the function are non-shared variables, the analysis utilizes this information during the analysis of the callee function.

4. Composite Bug Patterns with Semantic Conditions

After reviewing the bug reports on Linux file systems (see Section 2.2), we defined the following four bug patterns:

1. Misused test and test-and-set
2. Unsynchronized communication at thread creation
3. Incorrect usage of atomic operations
4. Waiting for an already terminated thread

For each bug pattern, it takes approximately 3 hours for one graduate student with the knowledge on the Linux file systems and the bug pattern to define a corresponding syntactic bug pattern in PDL and implement a bug pattern detector based on the generated template code upon the COBET framework. The following subsections explain the four patterns in detail.

4.1. Misused Test and Test-and-Set

Test and test-and-set programming idioms are used to reduce the number of expensive lock operations required to protect a shared variable.

```
1:if(c) {  // Test and test-and-set idiom
2:  lock l_v;
3:  if(c) {
4:    update v;}
5:  unlock l_v;}
```

Suppose that `c` indicates whether or not a current thread can update a shared variable `v`. Before performing an expensive lock operation (line 2) to update `v` safely (line 4), this idiom checks whether or not `c` is satisfied (line 1). Thus, the lock operation is executed only when `c` is true. If `c` is true (line 1), the above code performs a lock operation (line 2) and checks `c` again (line 3), since `c` might be changed by other threads between lines 1 and 3.

Unfortunately, programmers often omit this second check (line 3), which results in a race condition. For example, Linux ChangeLog 2.6.11.1 reported a data race bug in
`ext3_discard_reservation()` in the `ext3` file system, which was caused by *misused* test and test-and-set idiom. We suspect that similar bugs existed in the subsequent releases or other modules of Linux as this bug pattern could be easily introduced during manual code optimization.

We define the 'misused test and test-and-set' bug pattern [1] as two sub-patterns, `pattern 1` and `pattern 2`, in Figure 5. This bug pattern has the following semantic conditions to check in all pattern matching instances:

1. A lockset at target code that matches `3a` must be disjoint with the lockset at target code that matches `3b`. Otherwise, target code that matches `pattern 1` and `pattern 2` do not interfere with each other.
2. `$w` should be a shared variable.
3. `$cond` should contain a variable that is equal to or aliased to `$w`

```
1a:pattern 1 {
2a: fun $f1 {
3a:  if $cond {
4a:   lock $l;
5a:   \{if $cond { }}
6a:   unlock $l;
7a: }}}
```

```
1b:pattern 2 {
2b: fun $f2 {
3b:  write $w;
4b:  }}
```

Figure 5: Misused test and test-and-set bug pattern

`sem_cond_checking(bug_instance bi)` in Figure 6 checks these semantic conditions. Through the pattern matching, each field of `bug_instance` contains a target code element that matches a corresponding PDL element of a bug pattern. Line 2 in Figure 6 checks whether or not target code that matches `pattern 1` and `pattern 2` is protected by a common lock. `bi._3a` and `bi._3b` represent target code that matches the abstract statements `3a` and `3b` in the PDL description (see Figure 5). `is_lockset_disjoint (bi._3a,bi._3b)` obtains the lock analysis results for the code statements that match abstract statements `3a` and `3b` of the PDL description. Lines 4 and 6 utilize alias analysis to check whether or not matched code statements may access the same shared variable and cause a data race. `bi._w` and `bi._cond` represent the code elements that match `$w` and `$cond` in the PDL description, respectively.

[1]This pattern is different from 'double-checked locking' [13], which consider test and test-and-set idioms as bugs due to the Java memory model. Our bug pattern checks whether or not test and test-and-set idioms are correctly used in general.

```
1: BOOL sem_cond_checking(bug_instance bi) {
2:   if (is_lock_disjoint(bi._3a, bi._3b) == FALSE)
3:     return FALSE;
4:   if (is_shared_var(bi._w) == FALSE)
5:     return FALSE;
6:   if (may_alias(bi._cond,bi._w) == FALSE)
7:     return FALSE;
8: return TRUE; }
```

Figure 6: Semantic condition checking function

```
// Matching with pattern 1
1c:int proc_get_sb(file_system_type *fs_type...){
2c:  ...
3c:  ei = PROC_I(sb->s_root->d_inode);
4c:  if(!ei->pid) {
5c:    rcu_read_lock();
6c:    ei->pid = get_pid(...;

// Matching with pattern 2
1d:int proc_get_sb(file_system_type *fs_type...){
2d:  ...
3d:  if(!ei->pid)
4d:    ei->pid = find_get_pid(1);

// Matching with pattern 2
1e:inode *proc_alloc_inode(super_block *sb){
2e:  ...
3e:  ei = kmem_cache_alloc(...);
4e:  if (!ei) return NULL ;
5e:  ei->pid = NULL ;
```

Figure 7: `proc_get_sb()` (in `fs/proc/root.c`) and `proc_alloc_inode()` (in `fs/proc/inode.c`) of Linux kernel 2.6.30.4

```
// Matching with pattern 1
1f: void htable_put(xt_hashlimit_htable *hinfo){
2f:   if (atomic_dec_and_test(&hinfo->use)) {
3f:     spin_lock_bh(&hashlimit_lock) ;
4f:     hlist_del(&hinfo->node) ;
5f:     spin_lock_bh(&hashlimit_lock) ;

// Matching with pattern 2
1g: xt_hashlimit_htable *htable_find_get(net *net, u_int8_t family) {
2g:   ...
3g:   atomic_inc(&hinfo->use);
```

Figure 8: `htable_put()` and `htable_find_get()` in `net/netfilter/xt_hashlimit.c` of Linux kernel 2.6.30.4

We now illustrate how COBET detects this bug pattern in the example shown in Figure 7. In this example, suppose that COBET detects two syntactically matching instances:

- *Matching instance 1*: `proc_get_sb()` (lines `1c-6c`) and `proc_get_sb()` (lines `1d-4d`) matched `pattern 1` and `pattern 2`, respectively.
- *Matching instance 2*: `proc_get_sb()` (lines `1c-6c`) and `proc_alloc_inode()` (lines `1e-5e`) matched `pattern 1` and `pattern 2`, respectively.

For matching instance 1, to check semantic condition 1 (the lockset at target code that matches `3a` must be disjoint to the lockset at a target code that matches `3b`), COBET checks whether or not there exists a lock to synchronize the target codes that match `pattern 1` and `pattern 2`. COBET finds that the lockset at line `3c` and the lockset on line `3d` always contained `lock_kernel`. This indicates that line `3c` and line `3d` cannot run concurrently, thus cannot interfere each other. Thus, COBET ignores this matching instance.

For matching instance 2, COBET finds that there is an execution path that has no lock to synchronize the target code that match `pattern 1` and `pattern 2`. However, by checking semantic condition 2 (`$w` should be a shared variable), the alias analysis finds that `ei->pid` at line `5e` is not a shared variable, since `ei` was allocated (line `3e`) and had not yet become shared. Therefore, COBET concludes that the matching instance 2 should be rejected as well.

As another example, the following matching instance was found in the `netfilter` network module (see Section 5.3) as shown in Figure 8.

- *Matching instance 3*: `htable_put()` (lines `1f-5f`) and `htable_find_get()` (lines `1g-3g`) matched `pattern 1` and `pattern 2`, respectively.

Matching instance 3 was not filtered out by the semantic analyses, so we reported this result as a suspected bug to a corresponding Linux maintainer. The Linux maintainer in charge of `netfilter` confirmed this bug report and fixed `htable_put()` in Linux 2.6.34 [14].

```
1h:pattern 1 {                     1i:pattern 2 {
2h: fun $f1 {                      2i: fun $f2 {
3h:  call "kthread_run" $a1;       3i:  read $a3;
4h:  write $a2 ;                   4i: }}
5h: }}
```

Figure 9: Unsynchronized communication at thread creation pattern

4.2. Unsynchronized Communication at Thread Creation

The Linux kernel creates a new thread by using `kthread_run()`. For instance, `kthread_run(func, arg, "daemon")` creates a new kernel thread whose name is `"daemon"` and then executes `func(arg)` on the thread. `arg` is a single variable used as a function parameter. Through this variable, a parent thread transfers data used for the initialization of a new thread. In many cases, a parent thread passes a shared memory address through which the parent thread communicates with the child thread. For this type of communication, a parent thread and the child thread should be synchronized. However, programmers often omit synchronization so that concurrent execution of a parent thread and its child thread can exhibit data race errors. Linux ChangeLog 2.6.24 reported such a bug in `GFS2` file system.

We defined this 'unsynchronized communication at thread creation' bug pattern as the two sub-patterns `pattern 1` and `pattern 2` shown in Figure 9. `pattern 1` indicates a parent thread that calls `kthread_run()` and then assigns some value to the shared memory (line 4h). `pattern 2` describes a function executed by a child thread. `$f2` reads data passed from its parent thread through a pointer of the function parameter (line 3i). Since PDL does not specify expression-level conditions, a user needs to add additional condition checking code to the synthesized bug detector code (i.e., the first element of `$a1` should be same to `$f2`). This bug pattern has the following semantic conditions:

1. The lockset at `4h` must be disjoint with the lockset at `3i`.
2. `$a2` should be a shared variable.
3. `$a3` should contain a variable which is equal to or aliased to `$a2`

The 'unsynchronized communication at thread creation' bug detector found a new bug in `btrfs`, as shown in Figure 10. The child thread (`worker_loop()`) can access `worker` and read an invalid value (line `4k`), since the parent thread (`btrfs_start_workers()`) may not have assigned a proper value to `worker` (line `4j`) yet. We reported this bug to the kernel developers and they made the patch immediately (see Linux ChangeLog 2.6.31).

4.3. Incorrect Usage of Atomic Operations

Most computer architectures provide special instruction sets for synchronization operations called *atomic instructions*. 'test-and-set' and 'compare-and-swap' are popular forms of these instructions. The distinguishable feature of

```
// Matching with pattern 1
1j:int btrfs_start_workers(btrfs_workers *workers,
   int num_workers) {
2j: ...
3j: worker->task = kthread_run(worker_loop,worker,
    "btrfs-%s-%d",worker->name,worker->num_workers+i);
4j: worker->workers = workers;

// Matching with pattern 2
1k:int worker_loop(void *arg) {
2k:  btrfs_worker_thread *worker = arg ;
3k:  ...
4k:  work->worker = worker;
```

Figure 10: `btrfs_start_workers()` and `worker_loop()` at `fs/btrfs/async-thread.c`, Linux 2.6.30.4

```
1l:pattern 1 {                      1m:pattern 2 {
2l:  fun $f1 {                      2m:  fun $f2 {
3l:    call $atomic1 $a1 ;          3m:    call $atomic2 $a3 ;
4l:    if ($a2) { }                 4m:  }}
5l:    }}
```

Figure 11: Use atomic instructions in non-atomic ways bug pattern

atomic instructions is that computer architectures guarantee the atomic execution of read and consequent update on operands. These instructions are utilized for implementing higher synchronization mechanisms such as lock or semaphore, or used directly by programmers for light-weight synchronizations.

Linux kernel provides special variable type `atomic_t` and the library functions for atomic instructions (e.g., `atomic_read`, `atomic_set`). These are used for lock free synchronization on shared variables for simple and frequent updates. frequent and simple updates. Many counter variables on shared data structures are implemented by `atomic_t` and its associated operations.

Atomic instructions synchronize two threads by executing read and update operations atomically. For this reason, the programmers should use proper combined atomic instructiosn for two related read and write operations. However, programmers mistakenly use two separate atomic operations instead of one combined instruction, and it may lead to data race bugs.

We characterize this incorrect atomic instruction usages as a bug pattern shown in Figure 11. `pattern 1` represents two consecutive atomic instructions whose executions are intended to be atomic. `pattern 2` expresses another thread which can be scheduled between two atomic instructions inf `pattern 1` and interfere their executions.

The semantic condition checking examines the following conditions:

1. The set of held locks at `3l` must be exclusive to the set of held locks at `3m`.

```
 1: unsigned int ip_vs_in(unsigned int hooknum,
 2:  sk_buff *skb, net_device *in, net_vice *out,
 3:  int(*okfn)(sk_buff *)) {
 4:  ...
 5:  atomic_inc(&cp->in_pkts);
 6:  if (af == AF_INET &&
 7:      (ip_vs_sync_state & IP_VS_STATE_MASTER)
 8:      && (((cp->protocol != IPPROTO_TCP ||
 9:       && cp->state == IP_VS_TCP_S_ESTABLISHED)
10:        (atomic_read(&cp->in_pkts) %
11:         sysctl_ip_vs_sync_threshold[1]
12:         == sysctl_ip_vs_sync_threshold[0])) ||
13:       ((cp->protocol == IPPROTO_TCP) &&
14:        (cp->old_state != cp->state) &&
15:        ((cp->state == IP_VS_TCP_S_FIN_WAIT) ||
16:         (cp->state == IP_VS_TCP_S_CLOSE_WAIT) ||
17:         (cp->state == IP_VS_TCP_S_TIME_WAIT)))))
18:   ip_vs_sync_conn(cp);
19:   cp->old_state = cp->state ;
20:   ...
```

Figure 12: Use atomic instructions in non-atomic ways bug detected at `net/netfilter/ipvs/ip_vs_core.c`, Linux 2.6.30.4

Otherwise, `pattern 1` and `pattern 2` do not interfere with each other.

2. `$a1` and `$a3` should be shared variables and may alias to each other.
3. `$a2` should contain a variable which is equal to or aliased to `$a1`.
4. `$atomic1` is an atomic instruction which updates `$a1` (e.g., `atomic_inc`, `atomic_set`, etc). And `$atomic2` is an atomic instruction which updates `$a3`.
5. `$a2` should contains a function call to an atomic instruction which reads `$a1` (e.g, `atomic_read()`, etc).

We refine the bug pattern detector code manually to supplement expression level conditions. The bug pattern detector refers the type of the associated variables to check whether or not related variables are `atomic_t` variables. We provide the function names of library functions for atomic instructions and the bug pattern detector utilizes these information at checking the fourth and the fifth semantic conditions.

Using the bug pattern detector, we found undiscovered bugs in Linux kernel, one of them is shown in Figure 12. This code updates `&cp->in_pkts` at line 5 and then examines its value at line 10 and these might not be executed atomically. We repored this issue according to the result from the bug pattern detector, and the maintainers immediately patch the code that the two separate atomic instructions are replaced to a proper combined atomic instruction.

```
1n:pattern 1 {
2n:  fun $f1 {
3n:    call "kthread_run" $a1 ;
4n:    call "kthread_stop" $a2 ;
5n:    }}
```

```
1p:pattern 2 {
2p:  fun $f2 {
3p:    loop $c1 {
4p:      if $c2 {
5p:        break ; }
6p:  }}}
```

Figure 13: Waiting Already Finished Threads

4.4. Waiting Already Finished Thread

A Linux kernel can create and execute a child thread by calling `kthread_run()` and terminate the child thread by calling `kthread_stop()`. `kthread_stop()` sends a special message to a child thread and waits until the child thread is terminated. A child thread should regularly call `kthread_should_stop()` which returns true if the message is received and terminates respondingly. Otherwise, the parent thread waits indefinitely at `kthread_stop()` as no other thread can make the handshaking with the parent thread.

One programming pattern of a child thread is a function with a loop conditioned by `kthread_should_stop()`. The child thread processes task until `kthread_should_stop()` returns true. In this situation, the child thread should not terminate although the task is finished or the task may not proceed for errors, because the parent thread may invoke `kthread_stop()` after the child thread terminates and it results in indefinite waiting of the parent thread.

However, if a child thread has a path which escape the loop regardless of `kthread_should_stop()`, the child thread can terminate earlier than the parent thread's `kthread_stop()` and result in the deadlock situation. We observe an actual bug of this type reported at Linux Change Log 2.6.28, where the loop is escaped by `break` statement for an error condition.

We specify this bug into a PDL pattern as shown in Figure 13. `pattern 1` represents a parent thread which creates a child thread and invokes `kthread_stop()` for the child thread. `pattern 2` specifies the function for child thread which has a loop with `kthread_should_stop()`. Line ? represents the branch which escape the loop illegally. We manually supplemented the detailed conditions related to expressions. For this pattern, the semantics condition checking does not have any additional condition.

In Linux 2.6.30.4, we found the undiscovered bugs from `btrfs` file system code, one of them shown in Figure 14. The child thread unintentionally terminated when the error handling branch is taken (line 5q). The `btrfs` developer confirmed these bugs. And after Linux 2.6.32, the Linux kernel developers modified the semantics of `kthread_stop()` not to wait if the child thread is already finished, so that the bug of this categories may not induce error.

```
static int cleaner_kthread(void *arg) {
  btrfs_root *root = arg;
  do {
    smp_mb() ;
      if (root->fs_info->closing)
        break ;
      ...
  } while(!kthread_should_stop()) ;
  return 0; }
```

Figure 14: Waiting already finished thread bug detected at `fs/btrfs/async-thread.c` Linux 2.6.30.4

5. Empirical Results

To investigate the effectiveness, efficiency, and applicability of the COBET framework, we performed the following three empirical evaluations on Linux 2.6.30.4, the latest version at the time of this empirical study.

- To determine whether pattern-driven bug detectors based on the old bug reports can detect new concurrency bugs in subsequent releases, we applied the four bug pattern detectors (based on the bug reports on the file systems in Linux 2.6.0 to 2.6.30.3) to the file systems in Linux 2.6.30.4. We reported our bug detection results to Linux maintainers and validated the bug detection results by their feedback (Section 5.1).
- We evaluated the effectiveness and efficiency of the three semantic analyses (path analysis, lock analysis, and alias analysis) of the COBET semantic analysis engine. We measured the improvement in bug detection precision and the additional time cost associated with each semantic analysis (Section 5.2).
- To investigate the applicability of the COBET framework, we applied the four bug pattern detectors not only to file systems, but also to other modules. We applied the four bug pattern detectors to the device drivers and the network modules, and then evaluated the bug detection capability (Section 5.3).

All empirical studies were performed on 64-bit Fedora Linux 9 equipped with a 3.6 GHz Core2Duo processor and 16 GBytes memory.

5.1. Bug Detection Result on File Systems

We applied the four bug pattern detectors (Section 4) to the seven Linux file systems (`btrfs`, `ext4`, `nfs`, `proc`, `reiserfs`, `sysfs` and `udf`). Since Linux file systems are tightly coupled with virtual file system layer (VFS), we analyzed each file system together with VFS. We specified all system call handling functions in VFS as the thread starting points.

Table 3: Bug Detection Results on Linux File Systems

	`btrfs` (41KL)	`ext4` (28KL)	`nfs` (29KL)	`proc` (8KL)	`reiserfs` (27KL)	`sysfs` (3KL)	`udf` (9KL)	`vfs` (48KL)	Total (193KL)
Misused test and test-and-set	3/0	3/0	4/0	2/0	3/0	1/0	2/0	10/0	28/0
Unsync. comm at thread creation	2/1	0/0	0/0	0/0	0/0	0/0	0/0	0/0	2/1
Incorrect usage of atomic operations	5/0	0/0	1/0	0/0	7/0	0/0	0/0	3/0	18/0
Waiting already terminated thread	3/3	0/0	0/0	0/0	0/0	0/0	0/0	0/0	3/3
Total	12/4	3/0	6/0	2/0	5/0	1/0	2/0	11/0	51/4
Time(sec)	1.72	1.90	1.20	0.81	1.26	0.66	0.80		8.35

Table 3 describes the number of detected bugs for each bug pattern and for each file system. The first row shows the sizes of the file systems before pre-processing (for example, `btrfs` is 41000 lines long (the second column)). The first number in a cell of Table 3 indicates the number of new bugs detected by COBET. The second number indicates the number of real bugs among the detected bugs that were confirmed by the Linux maintainers. For example, COBET detected two 'unsynchronized communication at thread creation' bugs in `btrfs` file system (third row, second column). We reported these two suspected bugs to a `btrfs` maintainer, who confirmed that one was a real bug, but the other was a false alarm. COBET took only nine seconds to apply these four bug pattern detectors to the seven file systems (see the last row and the last column of Table 4).

Another observation is that relatively new file systems such as `btrfs` have several concurrency bugs. For example, `btrfs` was introduced in the Linux 2.6.29 release in March 2009. It has three 'waiting already terminated thread' bugs and one 'unsynchronized communication at thread creation' bug, which were confirmed by the Linux maintainers. If we can generalize this result, COBET can detect bugs in recently revised modules more effectively. Considering that Linux evolves rapidly (see Section 2.2), a light-weight bug detection tool such as COBET can be a practical aid to detect concurrency bugs in the OS.

5.2. Evaluation of Semantic Analysis Techniques

To investigate the effectiveness and efficiency of the COBET semantic analyses, we measured the false alarm reduction rate through the semantic analyses and additional time cost for the analyses. For this purpose, we performed four series of studies for each bug pattern detector with different combinations of semantic analyses (see Section 3.3).

1. The first series of studies detected one main sub-pattern without semantic analysis (see the second column of Table 4). This series of studies was similar to the studies with conventional pattern-based bug detection tools (e.g., MetaL [15] and FindBugs [16]).

2. The second series of experiments detected multiple sub-patterns of a bug pattern, but still without semantic analyses.
3. The third series extended the second series by performing path analysis and lock analysis as well. Note that the lock analysis depends on the path analysis and cannot be performed separately.
4. The fourth series extended the third by performing alias analysis as well. This series utilizes all semantic analyses by the semantic analysis engine.

Table 4: Effectiveness and Efficiency of the Semantic Analyses for the Linux File Systems

	Syn. matching (single sub-pattern)		Syn. matching (multiple sub-patterns)		Syn. matching + path analysis + lock analysis		Syn. matching + path analysis + lock analysis + alias analysis	
	Bug	Time	Bug	Time	Bug	Time	Bug	Time
Misused test and test-and-set	51	1.38	36	2.55	32	4.21	28	4.23
Unsync. comm. at thread creation	2	0.86	2	1.00	2	1.28	2	1.30
Incorrect usage of atomic operations	21	0.90	18	1.06	18	1.55	18	1.59
Waiting already terminated thread	3	0.64	3	0.74	3	1.01	3	1.23
Total	77	3.78	59	5.35	55	8.05	51	8.35

Table 4 describes the numbers of bugs detected and corresponding analysis time on the seven file systems in total. This table shows that the false alarms are reduced as additional analysis techniques are employed. For example, 'misused test and test-and-set' bugs (second row of Table 4) are reduced from 51 to 36, 32, and 28 as multiple pattern matching, path/lock analyses, and path/lock/alias analyses are applied respectively; finally 45% (=(51-28)/51) of the 'misused test and test-and-set' bugs were filtered out through these techniques.

The time costs for these analysis techniques were not burdensome. For example, the four bug pattern detectors spent 8.35 seconds in total to analyze the seven file systems with `vfs` with multiple sub-pattern matching and all semantic analyses (last row and last column of Table 4) while they required 3.78 seconds with syntactic analysis for a single sub-pattern only. The maximum memory consumption was less than 50 MBytes in the all experiments.

5.3. Bug Detection Results on Device Drivers and Network Modules

To investigate the general applicability of COBET, we applied the four pattern detectors for Linux file systems to other Linux modules. We targeted the seven modules in total including three Linux device drivers (`bluetooth`, `ieee1494`, and `mtd`) and four network modules (`atm`, `ax25`, `netfilter`, and `rds(ib)`). These target programs were implemented as loadable kernel module objects. Thus, the thread starting points of these modules are the function pointers registered at the module initializations.

Table 5: Bug Detection Result on Linux Device Drivers and Network Modules

	Device drivers			Network modules				Total
	`bluetooth` (11KL)	`ieee1394` (25KL)	`mtd` (15KL)	`atm` (8KL)	`ax25` (7KL)	`netfilter` (27KL)	`rds(ib)` (9KL)	(100KL)
Misused test and test-and-set	0/0	1/0	0/0	1/1	4/1	1/1	1/0	8/3
Unsync. comm. at thread creation	0/0	0/0	1/1	0/0	0/0	0/0	0/0	1/1
Incorrect usage of atomic operations	0/0	0/0	0/0	0/0	0/0	1/1	3/1	4/2
Waiting already terminated thread	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
Total	0/0	1/0	1/1	1/1	4/1	2/2	4/1	13/6
Time (sec)	5.90	7.29	7.85	0.46	1.06	24.65	1.64	48.85

Table 5 shows the bug detection results. The first number in each cell indicates the number of new bugs detected by COBET. The second number indicates the number of bugs that were confirmed as "real" by Linux maintainers. The three bug pattern detectors ('misused test and test-and-set', 'unsynchronized communication at thread creation', and 'incorrect usage of atomic operations') detected 13 bugs while 'waiting for an already terminated thread' detected no bug. Six bugs among these 13 bugs were confirmed as real ones by Linux maintainers (last column, last row).

Although the scope of this empirical study is limited, these results suggest that the bug patterns defined in one domain can be applied effectively to other domains and can help OS developers in practice. The following quotation is part of a response from a Linux maintainer to our 'misused test and test-and-set' bug report on `netfilter`.

> *Nice catch, this does indeed look like a bug. The entire locking concept seems a bit strange, we neither need an atomic_t for the reference count nor two locks to protect the list...* [2]

This bug report was immediately followed by the corresponding kernel patch. We received similar positive responses from other Linux maintainers regarding our bug reports and it indicates that the COBET approach can help kernel developers to detect subtle concurrency bugs in a practical manner.

5.4. Comparison with Coverity Prevent

We detect concurrency bugs from the Linux modules in case study by Coverity Prevent to compare COBET with a mature lock-based concurrency bug detection technique. Coverity Prevent is a pattern-driven bug detector which has been demonstrated successful bug detections in many real-world software projects for last ten years [17]. Coverity Prevent has the following five predefined bug pattern regarding concurrency: ATOMICITY, LOCK, MISSING_LOCK,

[2]This quotation is from an e-mail from Patrick McHardy on Jan 13th, 2010. The full text and patch information can be found at [14]

ORDER_REVERSAL, SLEEP. These patterns mainly specify five incorrect lock usages. The basic descriptions of LOCK and SLEEP patterns are presented in [18] and MISSING_LOCK and ORDER_REVERSAL in [2]. ATOMICITY detects a bug when a shared variable defined while holding a locking may be used after releasing the lock.

We applied the five bug detectors to compare the bug detection capabilities of COBT and Coverity Prevent and the time performance of COBET and Coverity Prevent. For fair comparison, we configure Coverity Prevents to (1) analyzes the same Linux module as COBET experiments, (2) enable only the five concurrency related bug patterns and disable all others, (3) enable function call via function pointers, (4) set bug sensitivity high so to report as many bugs as possible, and (5) acknowledge domain-specific lock operations used in the COBET configurations. We run Coverity Prevents on Intel(R) Core 2 Duo CPU E 8500 at 3.0 GHz, the same processor used in COBET experiment.

Table 6: Bug Detection Result by Coverity Prevent on Linux file systems

	`btrfs` (98KL)	`ext4` (85KL)	`nfs` (98KL)	`proc` (75KL)	`reiserfs` (61KL)	`sysfs` (75KL)	`udf` (71KL)	`vfs` (58KL)	Total (621KL)
ATOMICITY	45	1	1	3	5	3	0	4	62
LOCK	7	5	0	0	0	0	0	1	13
MISSING_LOCK	3	1	1	1	1	1	0	3	11
ORDER_REVERSAL	0	0	0	0	0	0	0	0	0
SLEEP	0	0	0	0	0	0	0	0	0
Total	55	7	2	4	6	4	0	8	86
Time (sec)	47.72	51.37	75.80	14.56	11.53	14.58	27.93	9.37	252.86

Table 7: Bug Detection Result by Coverity Prevent on Linux device drivers, and network modules

	Device drivers			Network modules				Total
	`bluetooth` (70KL)	`ieee1394` (87KL)	`mtd` (63KL)	`atm` (65KL)	`ax25` (62KL)	`netfilter` (89KL)	`rds` (70KL)	total (??KL)
ATOMICITY	0	4	14	0	0	0	2	20
LOCK	0	1	0	4	4	1	1	11
MISSING_LOCK	0	2	0	5	7	2	5	21
ORDER_REVERSAL	0	0	0	0	0	0	0	0
SLEEP	0	0	0	0	0	0	0	0
Total	0	7	14	9	11	3	8	52
Time (sec)	9.12	29.78	17.27	13.23	4.86	15.89	9.18	99.33

Table 6 and 7 shows the number of bug detection reported by Coverity Prevents and the execution time for each modules. For checking file systems, we examine each naive file systems together with VFS as we did with COBET. The code size in the table represents the pre-processed one, whereas the code size in Table 3 and 5 is the original one. COBET also uses pre-processed codes so that the amount of analyzed code is similar in COBET and Coverity Prevent. The number appears on the third to eighth rows refers the number of detections produced by corresponding checkers. We do not validate whether a detection is a real bug or false alarm, since we do not have sufficient domain knowledge and asking developers to validate detections will require lots of efforts.

In order to compare the bug detection capability, we first checked how many real bugs detected by COBET is also detected by Coverity Prevent. We found that Coverity Prevent does not detect any of the ten real bugs uncovered by COBET. As mentioned in Coverity manual [19], Coverity Prevent handles only standard lock and recursive lock operations as synchronization operations and does not have capability to model diverse synchronization operations. For this reason, it fails to examine concurrency codes that exploit other synchronization operations. This observation supports our claim that COBET can complement lock-based concurrency bug detection techniques as COBET targets orthognal domain of bug patterns.

The experiment results shows the significant difference in the computation time. From this observation, we presume that COBET does much cost-effective analysis than Coverity Prevents. For example, Coverity Prevents consumes 30x more time than COBET for checking the same file systems. Since we cannot control Coverity Prevent internal analysis engine, we cannot indicate by which aspect of COBET brough efficeincy over Coverity Prevent. Nonetheless, we have the following conjectures for the reason for better time performance. Our first conjecture is that COBET is faster than Coverity Prevent because COBET does path-insensitive analysis whereas Coverity Prevent does path-sensitive analysis. For checking bug patterns against linear execution paths, Coverity Prevents may internally generate all possible linear executions from a target program code, and then examine each linear execution. COBET performs pattern matching agains code structure so that it does not differentiate all execution paths.

Our second conjecture for the performance difference is that COBET performs the syntactic pattern matching prior to full semantic analysis which remove irrelevant codes from the analysis in an early phase. COBET bug detector applies syntactic pattern matching so to identify portions of a target program ahead to at least one syntactic bug pattern matching. The following semantic analysis does not explore irrelevant codes, since the semantic information only associated with syntactic pattern matching instances will be used in the further step. As this technique is not utilized for Coverity Prevent, we suspect that the technique is a major factor for efficiency of COBET analysis.

We also observed that `btrfs` at which COBET found many bug detections Coverity Prevent also reports the more bug detections than other file systems in the experiments. This result advocates our observation that relatively new modules may contain more bugs than old modules (Section 5.1).

6. Related Work

Pattern based techniques [18, 15, 16, 20] can analyze large programs quickly, since these techniques perform pattern matching on a target program without sophisticated analyses. Engler et al. [15] used a high-level state-machine language MetaL to specify system rules (i.e., programming idioms) over linear execution paths. They applied system rules such as a 'holding lock' rule (i.e., the acquired locks should be released before a function exits) to several operating systems and found bugs [18]. However, they target sequential errors related to synchronization operations while COBET targets complex concurrency errors caused by thread interactions. Hovemeyer et al. [16] defined frequently observed Java concurrency bug patterns and analyzed the bytecode of the target Java program through code pattern matching. They found several concurrency bugs in Sun JDK 1.5 and an open source J2SE library. The false alarm ratio of simple bug patterns such as 'double check' that target sequential errors related to lock operations was less than 20%. However, the false alarm ratio of complex concurrency bug patterns (e.g. 'wait not in loop') was high, since [16] does not check thread interactions or semantic conditions. COBET targets complex concurrency bugs by utilizing multiple sub-patterns and semantic conditions. In addition, COBET helps engineers build bug detectors in a semi-automatic manner using PDL.

Otto et al. [20] propose a bug pattern matching technique with semantic analyses on locks for finding concurrency bugs in Java programs. The idea of utilizing semantic information for better accuracy is similar to COBET. However, they do not provide a pattern description language, or support multiple sub-patterns.

Lock based techniques concentrate on lock usages. Lock based techniques effectively detect deadlocks [21, 2, 3] and low-level data races [1, 2, 22, 5, 6] which occur only when no lock synchronizes multiple threads which read and update one shared variable. However, these techniques share the limitation when they are applied to OS codes which utilize various synchronization mechanisms other than lock.

Concurrency bug detection techniques that analyze stateful behavior of a target program detect violations of user-specified properties (i.e., assertions, invariants, or temporal logic formulae) by analyzing executions state by state, either by model checking [23, 24, 25] or by systematic testings [26, 27]. Nonetheless, the scalability of these approaches is still limited due to the state explosion problem and exponential numbers of execution scenarios. Thus, these approaches are still not capable of analyzing OS kernels in practice.

7. Conclusion

We have developed a pattern-based COncurrency Bug dETector (COBET) framework for operating systems. To target complex concurrency bugs, COBET utilizes composite bug patterns and associates semantic information with code structures in bug pattern matching. While most concurrency bug detection techniques concentrate on lock usages, COBET targets various concurrency bug patterns specified by a user, so as to detect complex bugs. The effectiveness, efficiency, and applicability of COBET were illustrated by detecting ten new bugs in the file systems, device drivers, and network modules of Linux 2.6.30.4

with a modest cost. Although the bug detection of COBET is neither sound nor complete, the empirical results indicate that the COBET approach can detect concurrency bugs in large and complex programs practically.

References

[1] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, M. Sridharan, Efficient and precise datarace detection for multithreaded object-oriented programs, in: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, PLDI '02, 258–269, 2002.

[2] D. Engler, K. Ashcraft, RacerX: Effective, Static Detection of Race Conditions and Deadlocks, in: Proceedings of the nineteenth ACM Symposium on Operating Systems Principles, SOSP '03, 2003.

[3] M. Naik, C. S. Park, K. Sen, D. Gay, Effective Static Deadlock Detection, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, 386–396, 2009.

[4] A. Raza, G. Vogel, RCanalyzer: A Flexible Framework for the Detection of Data Races in Parallel Programs, in: Proceedings of the 13th Ada-Europe international conference on Reliable Software Technologies, Ada-Europe '08, Springer-Verlag, 226–239, 2008.

[5] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson, Eraser: A Dynamic Data Race Detector for Multi-threaded Programs, ACM Transactions on Computer Systems 15 (4) (1997) 391–411.

[6] J. W. Voung, R. Jhala, S. Lerner, RELAY: static race detection on millions of lines of code, in: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC-FSE '07, 205–214, 2007.

[7] W. Xiong, S. Park, J. Zhang, Y. Zhou, Z. Ma, Ad hoc synchronization considered harmful, in: Proceedings of the 9th USENIX conference on Operating systems design and implementation, OSDI '10, 1–8, 2010.

[8] S. Lu, S. Park, E. Seo, Y. Zhou, Learning from mistakes: a comprehensive study on real world concurrency bug characteristics, in: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems, ASPLOS XIII, 329–339, 2008.

[9] G. Kroah-Hartman, J. Corbet, A. McPherson, Linux Kernel Development: How fast it is going, who is doing it, what they are doing, and who is sponsoring it: An August 2009 Update, Tech. Rep., the Linux Foundation, 2009.

[10] E. D. Group, The C++ Front End, `http://www.edg.com`, 2011.

[11] M. Dubiner, Z. Galil, E. Magen, Faster tree pattern matching, Journal of ACM 41 (1994) 205–213.

[12] P. Pratikakis, J. S. Foster, M. Hicks, LOCKSMITH: Practical static race detection for C, ACM Transactions on Programming Languages and Systems 33 (2011) 3:1–3:55.

[13] D. Hovemeyer, W. Pugh, Finding bugs is easy, in: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, OOPSLA '04, 132–136, 2004.

[14] Linux kernel mailing list of netfilter, `http://www.spinics.net/lists/netfilter-devel/msg11823.html`, 2010.

[15] S. Hallem, B. Chelf, Y. Xie, D. Engler, A system and language for building system-specific, static analyses, in: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, PLDI '02, 69–82, 2002.

[16] D. Hovemeyer, W. Pugh, Finding Concurrency Bugs in Java, in: In Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs, 2004.

[17] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, D. Engler, A few billion lines of code later: using static analysis to find bugs in the real world, Communications of the ACM 53 (2010) 66–75.

[18] D. Engler, B. Chelf, A. Chou, S. Hallem, Checking system rules using system-specific, programmer-written compiler extensions, in: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4, OSDI'00, USENIX Association, Berkeley, CA, USA, 2000.

[19] Coverity, Coverity 5.4 Checker Reference, 2011.

[20] F. Otto, T. Moschny, Finding synchronization defects in Java programs: extended static analyses and code patterns, in: Proceedings of the 1st international workshop on Multicore software engineering, IWMSE '08, 41–46, 2008.

[21] R. Agarwal, L. Wang, S. D. Stoller, Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring, in: Proceedings of the Parallel and Distributed Systems: Testing and Debugging, Springer-Verlag, 191–207, 2005.

[22] J. Erickson, M. Musuvathi, S. Burckhardt, K. Olynyk, Effective data-race detection for the kernel, in: Proceedings of the 9th USENIX conference on Operating systems design and implementation, OSDI'10, 1–16, 2010.

[23] M. Musuvathi, S. Qadeer, Iterative context bounding for systematic testing of multithreaded programs, in: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, PLDI '07, 446–455, 2007.

[24] H. Post, C. Sinz, W. Küchlin, Towards automatic software model checking of thousands of Linux modules - a case study with Avinux, Software Testing Verification and Reliability 19 (2009) 115–172.

[25] S. Qadeer, D. Wu, KISS: Keep It Simple and Sequential, in: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation, PLDI '04, 14–24, 2004.

[26] E. Farchi, Y. Nir, S. Ur, Concurrent Bug Patterns and How to Test Them, in: Proceedings of the International Parallel and Distributed Processing Symposium, 286b, 2003.

[27] P. Joshi, M. Naik, C.-S. Park, K. Sen, CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs 5643 (2009) 675–681.