Kronecker Algebra-based Deadlock Analysis in the Linux Kernel

Yoojin Park

Technical Report TR-0003
Embedded Systems Languages and Compilers Lab
Department of Computer Science
Yonsei University

December 2016

Abstract

Multi-core technology has moved concurrent programming to the forefront of computer science. The C programming language explicitly supports concurrent programming through POSIX threads. Threads execute in parallel and communicate via shared objects that can be locked using synchronized access to achieve mutual exclusion. However, with concurrent programming comes a new set of problems that can hamper the quality of the software. Deadlocks are category from such a problem.

The Linux kernel currently deploys on many different architectures. The Linux kernel source now has 12 million lines of code. The code of Linux kernel is mostly written in C. As of now, The Linux kernel source-code defines 45 kernel threads. For several kernel threads, multiple threads in sharing are executing in parallel at runtime.

The Linux kernel-level synchronization primitives are software mechanisms provided by the operating system for the purpose of supporting kernel thread synchronization. Examples are memory barriers, atomic operations, mutexes, spinlocks. Erroneous use of synchronization primitives is difficult to detect by testing. It leads to deadlocks in the Linux kernel on multi-core processor architectures.

This report describes a static analysis for C-code to detect deadlocks due to erroneous use of spinlocks. The intended application is the detection of deadlocks in the Linux kernel. A major obstacle is the sheer size of the Linux kernel source-code. We analyze the entire source code of all 45 concurrent threads defined in the Linux kernel. The sizes of the control flow graphs (CFGs) of the underlying kernel threads are intractable for static analysis. We thus devise a reduction mechanism to eliminate CFG nodes which are not relevant for deadlock detection. We define graph rewrite rules to make problem sizes tractable.

Deadlocks constitute a locking hierarchy violation. In our analysis, we show that spinlocks which can be shown to adhere to a given locking hierarchy are irrelevant for deadlock detection. This allows us to omit such locks and thus further reduce the problem size. Kronecker algebra is a matrix calculus that has been proven useful for analysing multi-threaded programs. Our analysis for deadlock detection among Linux kernel threads is based on Kronecker algebra. We employ Kronecker algebra on the entire Linux kernel source-tree to detect deadlocks.

1 Introduction

Now we are in the explicit parallelism multi-core processor era. As a result, the computing platform is parallel multi-core architectures. To obtain performance increase in the multi-core processor, developers implement parallel applications instead of sequential ones. Task parallelism is a form of parallelization of computer code across multi-core processors. Using threads are one of solution for task parallelism. Threads execute in parallel and communicate via shared objects. Shared objects must be locked using synchronization primitives to achieve mutual exclusion.

The C programming language is one of popular language and explicitly supports concurrent programming through POSIX threads. Multi-core programming with POSIX threads and locks introduces potential concurrency bugs. Deadlocks are one of potential concurrency bugs and the hardest bug to find by a developer. As an example, it is sufficient to consider two threads which take each other locks. Given two locks A and B, and two threads, if thread 1 takes lock A then B while still under A, and thread 2 takes lock B then A while still under B. The deadlock occurs if thread 1 takes lock A, then thread 2 takes lock B before thread 1 does.

Deadlocks are related with all possible thread interleavings using locks. To analyze deadlock from concurrent programs by threads, Analysis must consider all possible thread interleavings.

Existing techniques do not scale for large code-bases. One of simple example for deadlock analysis is Dijkstra's Dining Philosophers which has 20 lines of code and 20 threads only. Existing techniques show deadlock analysis using a simple example like Dining Philosophers. Real-world source codes are bigger that example one. The number of threads in real-world source codes can be grown exponentially.

The Linux operating system is deployed on many architectures. The Linux operating systems are now everywhere from mobile phones to factory systems. Almost all embedded devices we use are controlled by the Linux operating system. The Linux kernel is a major part of the Linux operating system. The Linux kernel source now 12 million lines of code, 23 thousand functions, 45 district kernel thread functions and more thread instances. The code of Linux kernel is mostly C. There are some debugging mechanisms that can be activated in the kernel that can detect deadlocks at runtime. To enable these mechanisms, the kernel has to be complied in debugging mode with watchdog functions and monitors included. They are also often not used simply because they require recompiling the whole kernel and extra drivers are in an inconvenient. And of course, the mechanisms only detect deadlocks that occur, not those that do not occur during the testing [1]. The deadlock is relatively easy to evoke via coding errors, because different authors may decide to take two common spinlocks in a different order.

Programs that we want to analyze are large codebase like the Linux kernel. If We analyze the Linux kernel, we can analyze other legacy source codes written by C.

This report makes the following contributions.

1) We model Linux kernel threads for use in the Kronecker matrix calculus. We compile Linux kernel files using the LLVM/CLANG compiler and get intermediate representation (IR) files from C files. LLVM-link is a tool to link IR files into one IR file. We link generated IR files into one IR file using LLVM-link. We traverse linked IR file and find thread functions. The static spinlocks are declared statically in the kernel source code. We find static spinlocks as a synchronization primitive. CFGs of threads consists of basic blocks and relation between them which models the flow of control. We generate CFGs of all Linux kernel threads. We identify the Linux kernel threads in section same as our root nodes on which we perform function inlining to create complete CFG of a kernel thread.

2) We define graph rewrite rules which eliminate CFG nodes (basic blocks) that are irrelevant for deadlock detection. We link kernel IR files into one IR file. The linked IR file has 25,623

[1] According to E. Dijkstra's famous quote: "Testing can only proof the presence of errors, not their absence". [https://en.wikiquote.org/wiki/Edsger_W._Dijkstra]

functions. We inline all procedure calls into root function and create complete CFG the of kernel thread. Not all nodes of a CFG are relevant for deadlock detection. Shrinking CFG is required to ensure tractable problem size. Nodes and edges of complete CFG are shrunk by defined graph rewrite rules.

3) We provide novel technique to detect the violations of locking hierarchy base on Kronecker algebra. The locking hierarchy is an order of semaphores depicting how threads acquire locks for a critical section. The violation of the locking hierarchy makes a deadlock in the system. To find existing deadlocks in the system, We find semaphores which violate the locking hierarchy base on Kronecker algebra.

2 Background

We introduce background and basic notation required in this report. We start with introducing our semiring and continue with CFGs. We introduce Kronecker product and Kronecker sum forms the so-called Kronecker algebra. We define both operations, properties and give examples on matrix and automata level. We give deadlock analysis example using Kronecker algebra.

2.1 Semiring

In this section, we define our semiring. Properties and definitions of the semiring are from [16, 17].

Our semiring consists of a set of labels $\mathcal{L}$ which is defined by $\mathcal{L} = \mathcal{L}_\mathrm{V} \cup \mathcal{L}_\mathrm{S}$, where $\mathcal{L}_\mathrm{V}$ is the set of non-synchronization labels, and $\mathcal{L}_\mathrm{S}$ is the set of labels representing synchronization primitive calls, e.g., p_i and v_i are referred to the operation p and v of semaphore i. The sets $\mathcal{L}_\mathrm{V}$ and $\mathcal{L}_\mathrm{S}$ are disjoint.

Semiring $\langle \mathcal{L}, +, \cdot, 0, 1\rangle$ consists of a set of labels $\mathcal{L}$, two binary operations $+$ and $\cdot$, and two constants 0 and 1 such that

1) $\langle \mathcal{L}, +, 0\rangle$ is a commutative monoid,

2) $\langle \mathcal{L}, \cdot, 1\rangle$ is a monoid,

3) left and right distributivity of $\cdot$ over $+$:
   - $\forall l_1, l_2, l_3 \in \mathcal{L} : l_1 \cdot (l_2 + l_3) = l_1 \cdot l_2 + l_1 \cdot l_3$ and
   - $(l_1 + l_2) \cdot l_3 = l_1 \cdot l_3 + l_2 \cdot l_3$ hold, and

4) constant 0 is an absorbing element concerning the semiring operation $\cdot$:
   $\forall l \in \mathcal{L} : 0 \cdot l = l \cdot 0 = 0$.

Intuitively, our semiring is a unital ring without subtraction. For each $l \in \mathcal{L}$ the usual rules are valid, e.g., $l + 0 = 0 + l = l$ and $1 \cdot l = l \cdot 1 = l$. Also we equip our semiring with the unary operation $*$. For each $l \in \mathcal{L}$, l^* is defined by

- $l^* = \sum_{j \geq 0} l^j$, where $l^0 = 1$ and $l^{j+1} = l^j \cdot l = l \cdot l^j$ for $j \geq 0$.

2.2 Kronecker Algebra

Kronecker algebra calculus encodes finite automata as adjacency matrices. Kronecker product and Kronecker sum are so-called Kronecker algebra. In this section, we define both operations, state properties, and give examples on matrix.

2.2.1 Kronecker Product

Definition 1 (Kronecker Product). *Given an m-by-n matrix A and a p-by-q matrix B, their Kronecker product $A \otimes B$ is a mp-by-nq block matrix defined by*

$$A \otimes B = \begin{pmatrix} a_{1,1} \cdot B & \cdots & a_{1,n} \cdot B \\ \vdots & \ddots & \vdots \\ a_{m,1} \cdot B & \cdots & a_{m,n} \cdot B \end{pmatrix}.$$

Kronecker product is simultaneous execution of automata A and B. The operation $\otimes$ can be used to synchronize automata. In the following, we give a Kronecker product example.

Example 1. *We use the following matrices A and B.*

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \text{ and } B = \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix}.$$

The Kronecker product $A \otimes B$ is a 6 by 6 matrix defined by

$$A \otimes B = \begin{pmatrix} a_{1,1} \cdot b_{1,1} & a_{1,1} \cdot b_{1,2} & a_{1,1} \cdot b_{1,3} & a_{1,2} \cdot b_{1,1} & a_{1,2} \cdot b_{1,2} & a_{1,2} \cdot b_{1,3} \\ a_{1,1} \cdot b_{2,1} & a_{1,1} \cdot b_{2,2} & a_{1,1} \cdot b_{2,3} & a_{1,2} \cdot b_{2,1} & a_{1,2} \cdot b_{2,2} & a_{1,2} \cdot b_{2,3} \\ a_{1,1} \cdot b_{3,1} & a_{1,1} \cdot b_{3,2} & a_{1,1} \cdot b_{3,3} & a_{1,2} \cdot b_{3,1} & a_{1,2} \cdot b_{3,2} & a_{1,2} \cdot b_{3,3} \\ a_{2,1} \cdot b_{1,1} & a_{2,1} \cdot b_{1,2} & a_{2,1} \cdot b_{1,3} & a_{2,2} \cdot b_{1,1} & a_{2,2} \cdot b_{1,2} & a_{2,2} \cdot b_{1,3} \\ a_{2,1} \cdot b_{2,1} & a_{2,1} \cdot b_{2,2} & a_{2,1} \cdot b_{2,3} & a_{2,2} \cdot b_{2,1} & a_{2,2} \cdot b_{2,2} & a_{2,2} \cdot b_{2,3} \\ a_{2,1} \cdot b_{3,1} & a_{2,1} \cdot b_{3,2} & a_{2,1} \cdot b_{3,3} & a_{2,2} \cdot b_{3,1} & a_{2,2} \cdot b_{3,2} & a_{2,2} \cdot b_{3,3} \end{pmatrix}$$

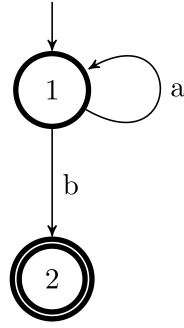
Example 2. *In this example, we show how we calculate all possible simultaneous execution of the two automata represented by the matrices*

$$C = \begin{pmatrix} a & b \\ 0 & 0 \end{pmatrix} \text{ and } D = \begin{pmatrix} 0 & a \\ 0 & b \end{pmatrix}.$$

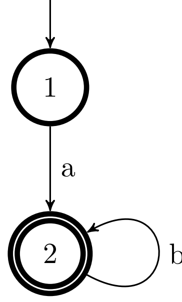
The corresponding automata are Figure 1(a) and 1(b). The Kronecker product $C \otimes D$ is a 4 $\times$ 4 matrix defined by

$$C \otimes D = \begin{pmatrix} 0 & a \cdot a & 0 & b \cdot a \\ 0 & a \cdot b & 0 & b \cdot b \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

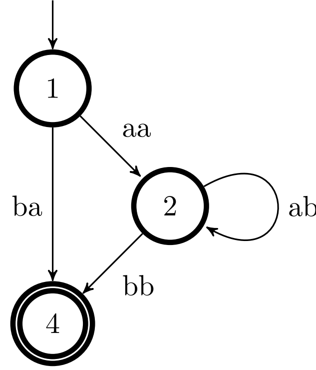
The corresponding automaton of $C \otimes D$ is Figure 1(c). In Figure 1(c), there are two input automata on each edge. This means the both automata perform a single step at the same time.



(a) Graph for C



(b) Graph for D



(c) Graph for $C \otimes D$

Figure 1: Simultaneous execution of Kronecker product $C \otimes D$

2.2.2 Properties

In the following, we introduce basic properties of the Kronecker product. For example, Let A, B and C be matrices. The Kronecker product distributes over the + operation [15], i.e.,

$$A \otimes (B + C) = A \otimes B + A \otimes C,$$
$$(A + B) \otimes C = A \otimes C + B \otimes C,$$
$$(A + B) \otimes (C + D) = A \otimes C + B \otimes C + A \otimes D + B \otimes D.$$

2.2.3 Kronecker Sum

Definition 2 (Kronecker Sum)**.** *Given a matrix A of order m and a matrix B of order n, their Kronecker sum $A \oplus B$ is a matrix of order mn defined by*

$$A \oplus B = A \otimes I_n + I_m \otimes B$$

where I_m and I_n denote identity matrices of order m and n, respectively.

The Kronecker sum calculates all interleavings of two concurrently executing automata A and B. Operation $\oplus$ can be used to model concurrency. In the following, we give a Kronecker sum example.

Example 3. *We show how to calculate all possible interleavings of the two automata represented by the following matrices*

$$A = \begin{pmatrix} 0 & a & 0 \\ 0 & 0 & b \\ 0 & 0 & 0 \end{pmatrix} \text{ and } B = \begin{pmatrix} 0 & c & 0 \\ 0 & 0 & d \\ 0 & 0 & 0 \end{pmatrix}.$$

The corresponding automata are Figure 2(a) and 2(b). The Kronecker sum $A \oplus B$ is a 9×9 matrix defined by
$A \otimes I_3 + I_3 \otimes B =$

$$\begin{pmatrix} 0 & a & 0 \\ 0 & 0 & b \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & c & 0 \\ 0 & 0 & d \\ 0 & 0 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 0 & 0 & a & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & a & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & a & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & b & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & b & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & b \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & c & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & d & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & d & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & c & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & d \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & c & 0 & a & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & d & 0 & a & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & a & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c & 0 & b & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & d & 0 & b & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & b \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & c & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & d \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

The corresponding graph is depicted in Figure 2 (c).

2.2.4 Example: Deadlock Analysis using Kronecker Algebra

We use a Dijkstra's Dining Philosophers problem with two threads and two semaphores. To illustrate Kronecker algebra-based deadlock analysis, we assume that thread T_1 acquires semaphore S_1 then semaphore S_2 while still holding semaphore S_1, and thread T_2 acquires semaphore S_2 then semaphore S_1 while still holding semaphore S_2. The deadlock occurs if thread T_1 acquires semaphore S_1, then thread T_2 acquires semaphore S_2 before thread T_1 does. Figure 3 (a) and (b) contain the CFGs of threads T_1 and T_2. Figure 3 (c) and (d) contain the CFGs of semaphore S_1 and S_2. Semaphores are used in T_1 and T_2.

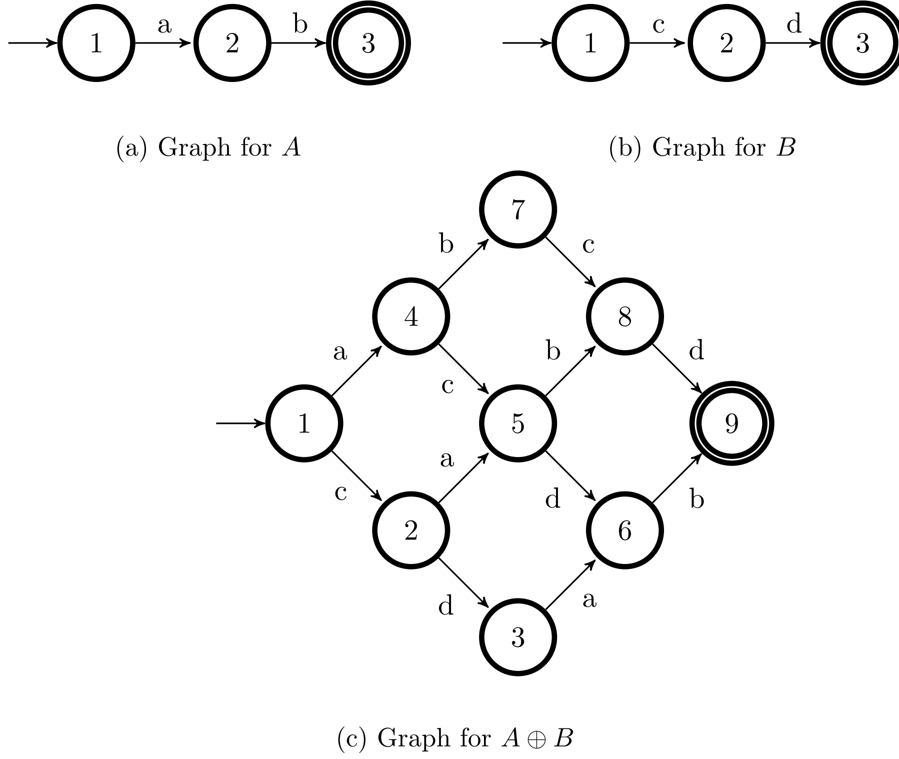
In the following, we show how to analyze deadlock using Kronecker product and Kronecker sum in this example.

Example 4. *We model CFGs of T_1, T_2, S_1 and S_2 depicted in Figure 3 as follows.*

$$T_1 = \begin{pmatrix} 0 & p_1 & 0 & 0 \\ 0 & 0 & p_2 & 0 \\ 0 & p_1 & 0 & v_1 \\ v_2 & 0 & 0 & 0 \end{pmatrix} \text{ and } T_2 = \begin{pmatrix} 0 & p_2 & 0 & 0 \\ 0 & 0 & p_1 & 0 \\ 0 & p_1 & 0 & v_2 \\ v_1 & 0 & 0 & 0 \end{pmatrix}$$

$$S_1 = \begin{pmatrix} 0 & p_1 \\ v_1 & 0 \end{pmatrix} \text{ and } S_2 = \begin{pmatrix} 0 & p_2 \\ v_2 & 0 \end{pmatrix}$$

Let T_i for $i = 1, ..., n$ be concurrently executing threads and S_j for $j = 1, ..., k$ be semaphores. Then the example consisting of threads $T_i (i = 1, ..., n)$ and semaphores $S_j (j = 1, ..., k)$ can be modeled by



(a) Graph for A

(b) Graph for B

(c) Graph for $A \oplus B$

Figure 2: All interleavings of Kronecker sum $A \oplus B$

$$\bigoplus_{i=1}^{n} T_i \ \otimes \ \bigoplus_{j=1}^{k} S_j = (T_1 \oplus T_2) \otimes (S_1 \oplus S_2).$$

The corresponding concurrent program graph (CPG) of two Dining Philosophers is Figure 4. Clearly, Node 24 cannot reach to Node 1 which is the final node. Thus, the deadlock is detected. The deadlock occurs if philosopher 1 picks up fork 1 and before he picks up fork 2, philosopher 2 picks up it. There is a second way for the deadlock to occur if philosopher 2 picks up fork 2 and before he picks up fork 1, philosopher 1 picks up it.

2.2.5 Skeleton of CFG for Kronecker Algebra.

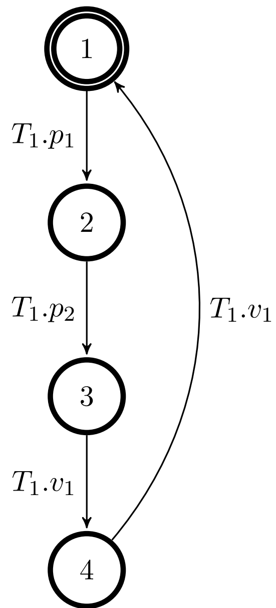
CFG includes all inputs and nodes of a program. However, Not all inputs and nodes are used for Kronecker algebra analysis in CFG. We eliminate additional inputs and nodes in CFG.

The steps to eliminate additional inputs and nodes in CFG depicted in Figure 5. We define graph rewrite rules to eliminate CFG nodes that are irrelevant for deadlock detection in this report and reduce CFG nodes in “Liner probing” step. “Liner probing” step creates probed CFG as an input and makes CFG size tractable for “Quadratic probing”. The skeleton CFG contains the inputs and nodes needed for Kronecker algebra calculus only. The skeleton CFG is the result of “Quadratic probing” step.

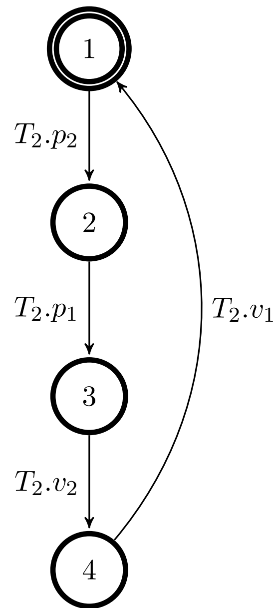
We eliminate additional inputs and nodes in “Quadratic probing” as follows. The first step to eliminating additional inputs from CFG is to replace the corresponding edge labels with “1”. See Figure 6 (a). The additional input “y”. It is replaced to “1” and depicted in Figure 6 (b). The next step is to eliminate the “1” edges and follows below algorithm.

1) If there are no “1”-edges, the algorithm terminates, otherwise go to 2.

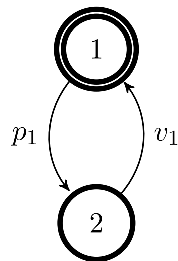
2) Remove all “1”-self-loop. If there are no more “1”-edges, the algorithm terminates, otherwise go to 3.



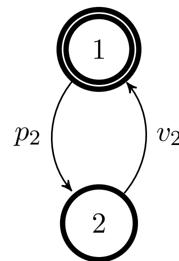
(a) CFG for Thread T_1



(b) CFG for Thread T_2



(c) CFG for Semaphore S_1



(d) CFG for Semaphore S_2

Figure 3: Example: Dining Philosophers with two threads and semaphores

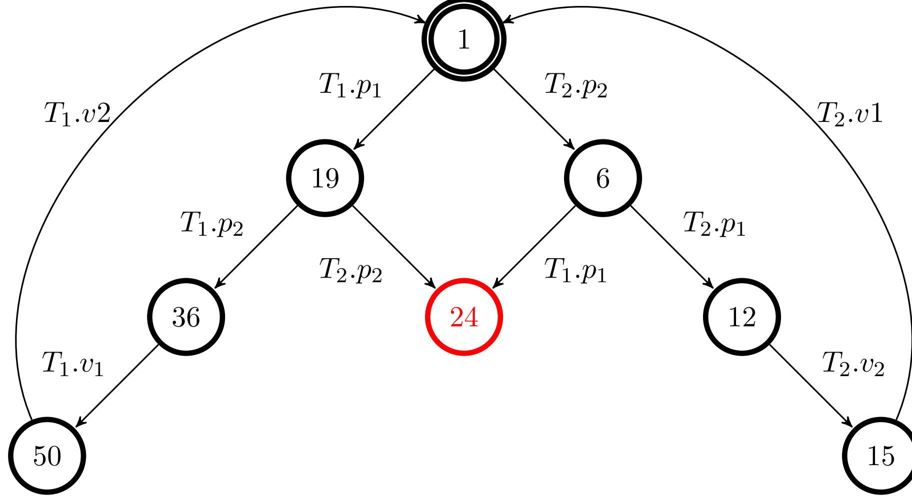


Figure 4: CPG of the two Dining Philosophers example

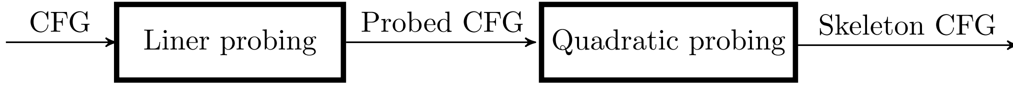
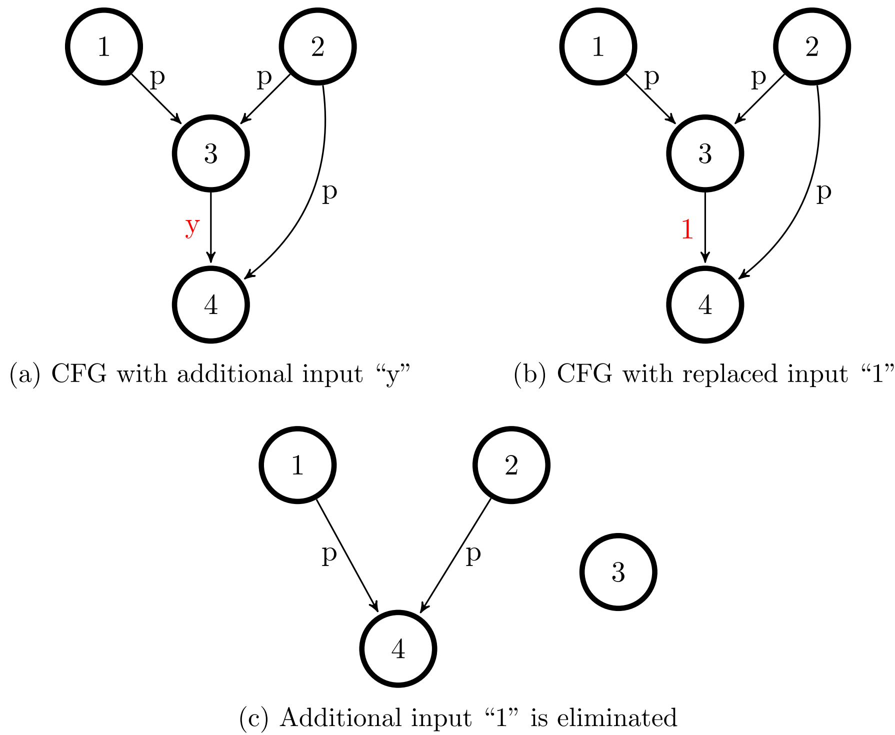


Figure 5: CFG reduction steps

3) Pick a node n with at least one outgoing "1"-edge, $(n \to m)$. Let $Pred(n)$ denote the set of all predecessors of n. For all edges from a node $p \in Pred(n)$ to n, add an edge $(p \to m)$. Remove the "1"-edge $(n \to m)$. Do this for all "1"-edges rooted at node n. Finally, go to 1. Do this for all "1"-edges rooted at node n. Finally, go to 1.

The result of algorithm about Figure 6 (b) is depicted in Figure 6 (c).

We simulate user scenario depicted in Figure 7 (a) and file usage system depicted in Figure 7 (b). User scenario in Figure 7 (a) contains the "c", "o" and "r" as the inputs. Compare to user scenario Figure 7 (a), file usage system Figure 7 (b) contains "a", "b", "d", "e", "f", "g", "h" and "i" as additional inputs. After applying above algorithm to Figure 7 (b), we get the result CFG depicted in Figure 8. The final step is that all nodes from which no final node can be reached are deleted from Figure 8. Figure 9 is the result of the final step and skeleton CFG of Figure 7 (b). The additional inputs "a", "b", "d", "e", "f", "g", "h" and "i" and Nodes "1", "3", "4", "5", "7" and "9" are eliminated from Figure 7 (b).



(a) CFG with additional input “y”

(b) CFG with replaced input “1”

(c) Additional input “1” is eliminated

Figure 6: CFG of skeleton

2.3 Control-flow Graphs

Modelling software systems can be done with finite state machines (FSMs). FSMs consist of a finite number of states. States in FSMs are changed from one state to another when a condition is changed.

CFG consists of basic blocks and of a relation between them which models the flow of control. Basic blocks in CFG are a finite number of states. Basic blocks in CFGs are changed from one basic block to another according to a flow of control in a program. CFGs are used as FSMs to model software systems.

We represent threads and synchronization primitives in the form of CFGs. CFG is a directed labeled graph defined by $G = < V, E, n_e, V_f >$, where V is a set of nodes, edge $E \subseteq V$ x V, n_e is an entry node and $V_f \subseteq V$ is a set of final nodes.

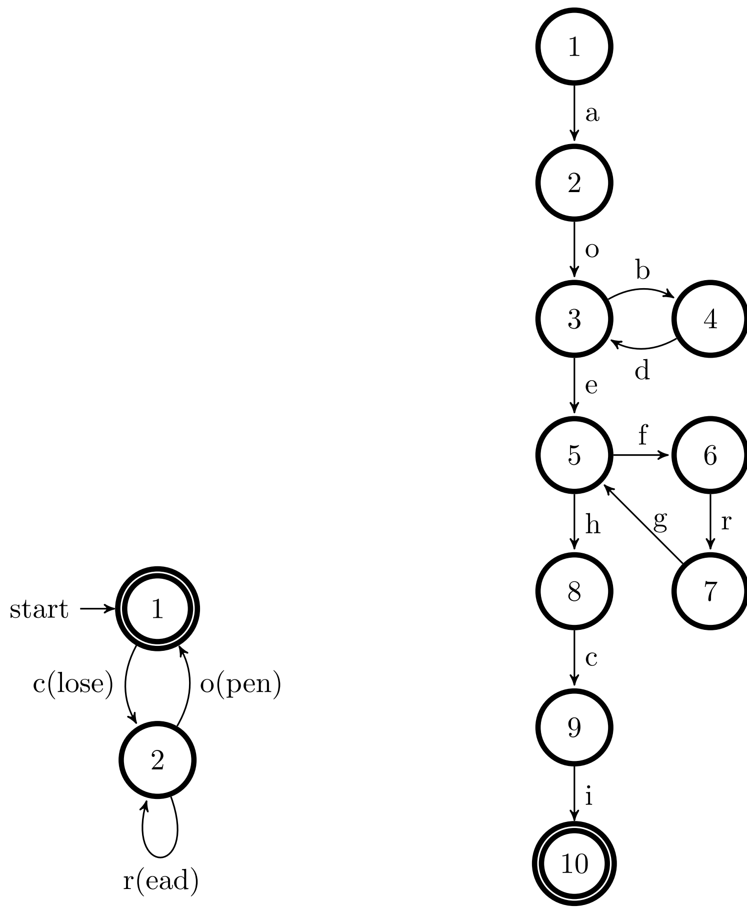
The CFG nodes represent basic blocks. The edges represent the transfer of control between the basic blocks. Each edge $e \in E$ is assigned to basic blocks. Figure 10 is one of function in Linux kernel named “usblog_flush”.

3 Related Work

3.1 Kronecker Algebra

The most closely related work is [8]. It worked on the static analysis of Ada programs with protected objects. The plain Kronecker product is used to simulate the execution of concurrent threads with automata. The plain Kronecker sum is used to model all interleavings of concurrent threads with automata. Ada protected objects are modeled as CFGs. We use same definitions of Kronecker algebra as proposed from [8]. We extend the Kronecker algebra to check locking hierarchy and find deadlocks in multi threads. Our approach is capable of analyzing huge programs efficiently.

Another closet work was done by [7]. It worked on generating reachability sets in composed



(a) CFG of file user scenario

(b) CFG of file usage system

Figure 7: CFGs to make skeleton

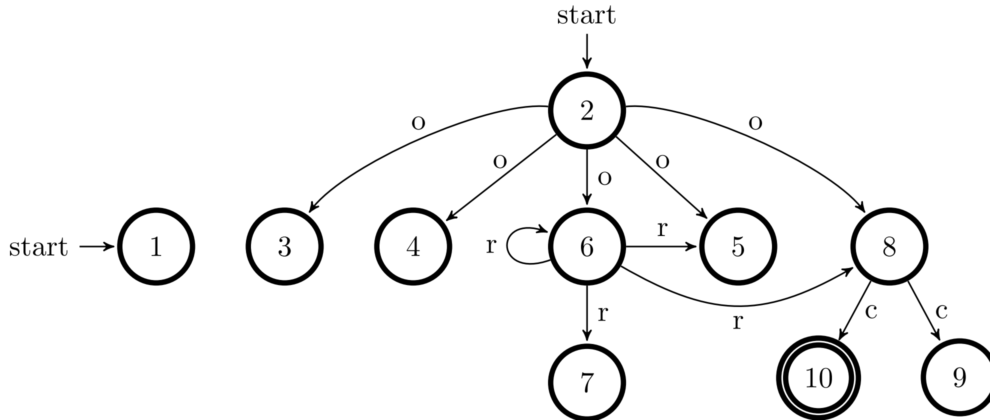


Figure 8: Result CFG of eliminating additional inputs in file usage system

automata. However, it differs from our work as follows. It used boolean matrices. Our matrices are labels from a semiring.

Different approaches are data flow analysis by [4], [5], [23], [14] and symbolic analysis [6].

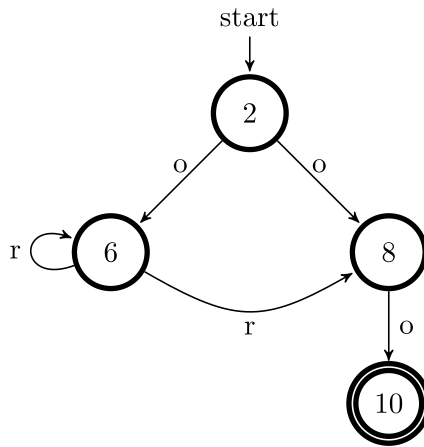


Figure 9: Skeleton CFG of file usage system

3.2 Control-flow Graph Reduction

Graph reduction and splitting approach are presented from [13]. Node splitting is the method to make irreducible graph reducible. From [11], normalization is proposal to make irreducible graph reducible in the front-end of a compiler.

3.3 Deadlock Analysis

We divide deadlock detection techniques into two categories as static and dynamic. A static technique examines the text of a program without executing it. A dynamic technique detects deadlocks at run-time. Static techniques are presented from [8]. Procedures of a program are modeled as CFGs. All possible thread interleavings are simulated by Kronecker algebra. The approach from [12] focus on dynamic techniques for deadlock detection. Static deadlock analyzer finds deadlock candidates including false positives and false negatives. The starting point for deadlock detection is deadlock candidates provided by a static analyzer. Deadlock candidates are monitored and reported to analyzer when a deadlock occurs.

3.4 Static Analysis of Barriers

Barriers are employed in various parallel programming models from [18], [19] and [20]. In the following, we compare some of the work done in these areas to our work. From [1], the same number of barriers are executed in all threads to ensure the structural correctness. Static analysis is used to determine that same number of barriers are executed in all threads.

From [3], [25], the focus is that which portions of the program execute in parallel without verification of the correctness of barrier synchronization.

To larger set of barrier scenarios, the barrier matching is introduced by [24], [1]. It allows to prove the correctness of a larger set of barrier scenarios.

From [10], static and dynamic barriers are verified on the fork and join programs. From [2], several barrier scenarios are introduced and verified.

3.5 Modeling Concurrent Programs

To model concurrent programs, [21] and [22] are used widely. They model the whole systems instead of individual threads. Petri nets are constructed from scratch and cannot be generated from source code efficiently. To indicate a certain state, Petri nets use markings which define all possible states. In contrast Kronecker algebra models concurrent systems with the more and less independent components, namely threads and synchronization primitives, using CFGs. CFGs

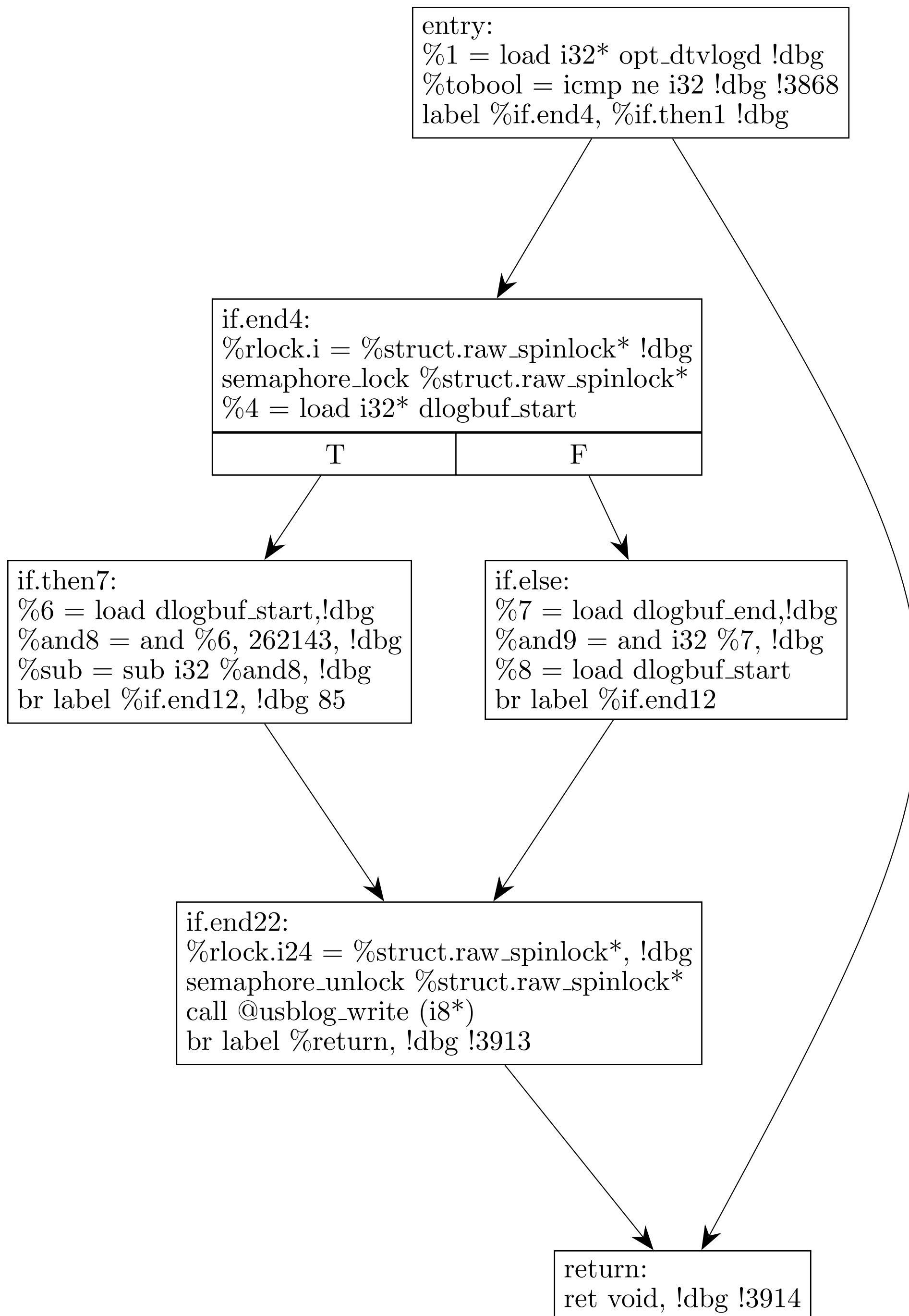


Figure 10: CFG of "usblog_flush" function in the Linux kernel

are frequently used for data structures, and Kronecker algebra is used to construct a global system view in a fully automated way.

4 Modeling Linux Kernel Threads for the Kronecker Matrix Calculus

In the Linux kernel, The *"kthread_create_on_node"* and *"kernel_thread"* functions are used to create kernel threads. The arguments of these functions are the thread name and the thread function. We use these functions to identify Linux kernel threads in the kernel source code as follows.

Clang is a compiler for C-code and generates intermediate representation (IR) files from C files. We compile the Linux kernel files using the Clang and link generated IR files to one IR file. We traverse the linked IR files and find 45 thread functions. These 45 functions are root nodes in the call graph of Linux kernel functions.

Linux kernel uses the macro definition to declare synchronization primitives. We add the annotation to macro definition by Clang compiler and find synchronization primitives.

To generate a finite automaton of kernel threads, we use CFG. We model Linux kernel threads in the kernel source code as follows.

4.1 Building Linux Kernel C Files.

We use Kronecker algebra to find deadlock in the Linux kernel threads. Kronecker algebra requires a finite automaton for each kernel thread. We use CFG to generate a finite automaton of the kernel thread. CFG is a directed and labeled graph with entry node and set of final nodes. A basic block is a single-entry and single-exit sequence of statements. Edges represent the flows of basic blocks in the program.

LLVM IR contains CFG information. We compile the Linux kernel C files by Clang compiler and get 1,305 IR files as follows.

4.1.1 Build-script for the LLVM compiler

The GNU Compiler Collection (GCC) is the compiler used with Linux kernel. GCC compiles Linux kernel C files and generate object files with building configurations. Linux kernel has the building configurations for GCC already.

To generate LLVM IR files of Linux kernel C files, we run GCC and Clang compiler at the same time. GCC compiles all C files in Linux kernel and passes the building configurations to Clang compiler.

GCC uses compile options for Linux kernel in building configuration. Examples are "-Os", "-mcpu" and "-mfpu". Not all compile options of GCC are supported by Clang compiler. Unsupported compile options by Clang compiler make errors of compilation. To avoid compile error of Clang compiler by compile options supported by GCC only, we use the script. The script collects compile options for Clang compiler from GCC compile options. The script is depicted in Listing 1. All compile options passed by GCC are stored to variable "GCC_CONFIGS". The function "Is_support_by_Clang" returns "true" if variable "option" is supported by Clang compiler. Unless returns "false". The variable "cfile" has the name of C file for compilation. All supported compile options by Clang compiler are collected to variable "Clang_options". Clang compiler compiles Linux kernel C file with variable "Clang_options" and generates LLVM IR files.

Listing 1: Build-Script for the LLVM compiler

```
1 GCC_CONFIGS = arges[@]
2 for( i=0; i<$GCC_CONFIGS; i++); do
```

```
3     option=${args[$i]}
4     if [Is_support_by_Clang(option)]
5     then
6         Clang_options+="$option"
7 clang $Clang_options $cfile
```

4.2 Synchronization Primitives in the Linux Kernel for Deadlock Analysis

Kernel level synchronization primitives are software mechanisms provided by the operating system for the purpose of supporting kernel thread synchronization. Examples are memory barriers, atomic operations and spinlocks.

Static spinlocks are declared statically in the kernel source code. See Listing 2 for a statically declared spinlock. Linux kernel uses the macro "DEFINE_SPINLOCK" to initialize the spinlock variable. The *"static"* keyword makes the variable local to the containing file. Accessing the static spinlock from outside is only possible via the functions declared in the file. A static spinlock uses a static variable as a locking variable. All concurrent threads access static memory to acquire this locking variable. Kernel threads share static spinlocks to avoid race conditions. We use static spinlocks as a synchronization primitive for deadlock analysis in Linux kernel.

Listing 2: Statically declared "my_lock" variable as spinlock in Linux kernel.

```
1 #define DEFINE_SPINLOCK(x) spinlock_t x = __SPIN_LOCK_UNLOCKED(x)
2 static DEFINE_SPINLOCK(my_lock);
3 int my_kernel_func(void)
4 {
5     spin_lock(&my_lock);
6     {
7      //critical section
8     }
9     spin_unlock(&my_lock);
10    return 0;
11 }
```

4.3 Adding Annotations to Synchronization Primitives

To find static spinlocks in LLVM IR file, we use annotations. An annotation is a form of metadata by Clang compiler. "__attribute__" is the function in Clang compiler to annotate following string with "annotate" into a variable or function as a property during compilation. See Listing 3. We add annotation string "spinlock_annotation" to the macro of spinlock definition.

Listing 3: Adding annotation to spinlock macro

```
1 #define DEFINE_SPINLOCK(x) \
2         __attribute__((annotate(``spinlock_annotation''))) \
3         spinlock_t x = __SPIN_LOCK_UNLOCKED(x)
```

We compile Linux C source codes and generate LLVM IR file included annotation. Figure 11 (a) is the declaration of static spinlock variable "plock" and source code of function "cal" which uses the "plock". Figure 11 (b) is the LLVM IR of compiled static spinlock variable "plock" and function "cal". Compiled LLVM IR has annotation "spinlock_annotation" for "plock" variable as a property.

```
1 static \
2 DEFINE_SPINLOCK(plock);
3
4 int cal(struct m *pt)
5 {
6  spin_lock(plock);
7  add(&pt->list, head);
8  spin_unlock(plock);
9  return 0;
10 }
```

(a) Source codes of static spinlock “plock”

```
@.str35 = “spinlock_annotation\00”
@llvm.global.annotations = @plock
define int @cal (%struct.pt* %m) #0
{
store %struct.spinlock @plock,%lock.addr.i;
call void @spin_lock(%lock.addr.i);
call @add(...);
call void @spin_unlock(%lock.addr.i);
return 0;
};
```

(b) Compiled “plock” variable in LLVM IR

Figure 11: Annotation “spinlock_annotation” is added to “plock” in LLVM IR file

4.4 Stand-alone C++ Tool that Uses the LLVM Pass

To create CFGs of Linux kernel threads, we use stand-alone C++ tool. We get statements of the program from the interface of “ModulePass” in the stand-alone tool. See Listing 4. “main” function is executed and get LLVM IR file name as an argument. The “getFileOrSTDIN” is the function to save LLVM IR contents into a memory buffer. The “parseBitcodeFile” parses LLVM IR and returns all statements of LLVM IR. The “for” loops iterate functions, basic blocks and instructions to get information of CFG.

Listing 4: Stand-alone C++ tool

```
1 void main(int argc, char*argv[]) {
2  MemoryBuffer Buffer = getFileOrSTDIN(argv[1]);
3
4  Module* M = parseBitcodeFile \
5                (Buffer.get()->getMemBufferRef(), context);
6
7  Interator F, bb, inst;
8  for(F = M->begin(), E = M->end(); F != E; ++F)
9  {
10   // Get function information
11   for(bb = F->begin(), E = F->end(); bb != E; ++bb)
12   {
13    // Get basic block information
14    for(inst = bb->begin(), E = bb->end(); inst != E; ++inst)
15    {
16      //Get instruction informatoin
17    }
18   }
19  }
20 }
```

4.5 Create Control-flow Graphs of Linux Kernel Threads

In the Linux kernel, The “kthread_create_on_node” and “kernel_thread” function are used to create kernel threads. The arguments of these functions are the thread name and the thread function. We traverse merged LLVM IR file and find root function of the kernel thread. We find 45 functions as root functions of the kernel threads.

In merged LLVM IR file, there are all instructions and variables consisting Linux kernel. Not all instructions and variables in basic blocks are relevant for CFG. To create CFG of each function, we collect related information with CFG from basic blocks in merged LLVM IR file. A function consists of one or more basic blocks. If function consists of several basic blocks, they have information of predecessors. See Figure 12, keyword "preds =" represents basic block name of the predecessor of each basic block.

```
if.end: ;preds = %entry
%14 = load %struct.vdfs_xattrtree_key** !dbg !3068
%object_id6 = getelementptr inbounds 1, !dbg !3068
br i1 %cmp8, %if.then9, label %if.end10, !dbg !3071
```

Figure 12: Basic block "if.end" has predecessor basic block "entry".

We traverse all basic blocks of functions in Linux kernel and collect the basic block name, function name, predecessors of each basic block, static spinlocks and procedure calls. We store collected information to the structure depicted in Listing 5.

Listing 5: Structure of collected information for CFG

```
1 struct cfg_infomation
2 {
3     char  *lock;
4     char  *procedure_call;
5     char  *function_name;
6     char  *basic_block_name;
7     char  *predecessors;
8 };
```

Figure 13 is CFG of function "usblog_flush" in Linux kernel. It is generated by LLVM optimizer and includes all instructions and variables in "usblog_flush". Not all instructions and variables in basic blocks are needed for deadlock detection of "usblog_flush". We traverse basic locks of function "usblog_flush" and collect information from each basic blocks. Red texts in Figure 13 are collected information for deadlock detection in function "usblog_flush". We regenerate CFG of function "usblog_flush" using collected information.

See Figure 14. We regenerate CFG of function "usblog_flush" using collected information from basic blocks for deadlock analysis. Regenerated CFG of "usblog_flush" has basic block names, static spinlocks, procedure call and flows of basic blocks. Regenerated CFG makes problem size tractable.

5 Control-flow Graph Reduction

We link kernel IR files into one IR file. The linked IR file has 25,623 functions. The 45 kernel threads identified in section are the root nodes on which we perform function inlining to create the complete CFG of a kernel thread.

The CFGs of large code base may result in intractable problem size for Kronecker-based deadlock analysis. The CFG of a program is a directed graph $G = (N, E)$ where N is the set of nodes and E is the set of edges. Not all N of a CFG are relevant for deadlock detection. Shrinking CFGs is required to ensure tractable problem sizes. We define graph rewrite rules which eliminate CFG nodes that are irrelevant for deadlock detection as follows.

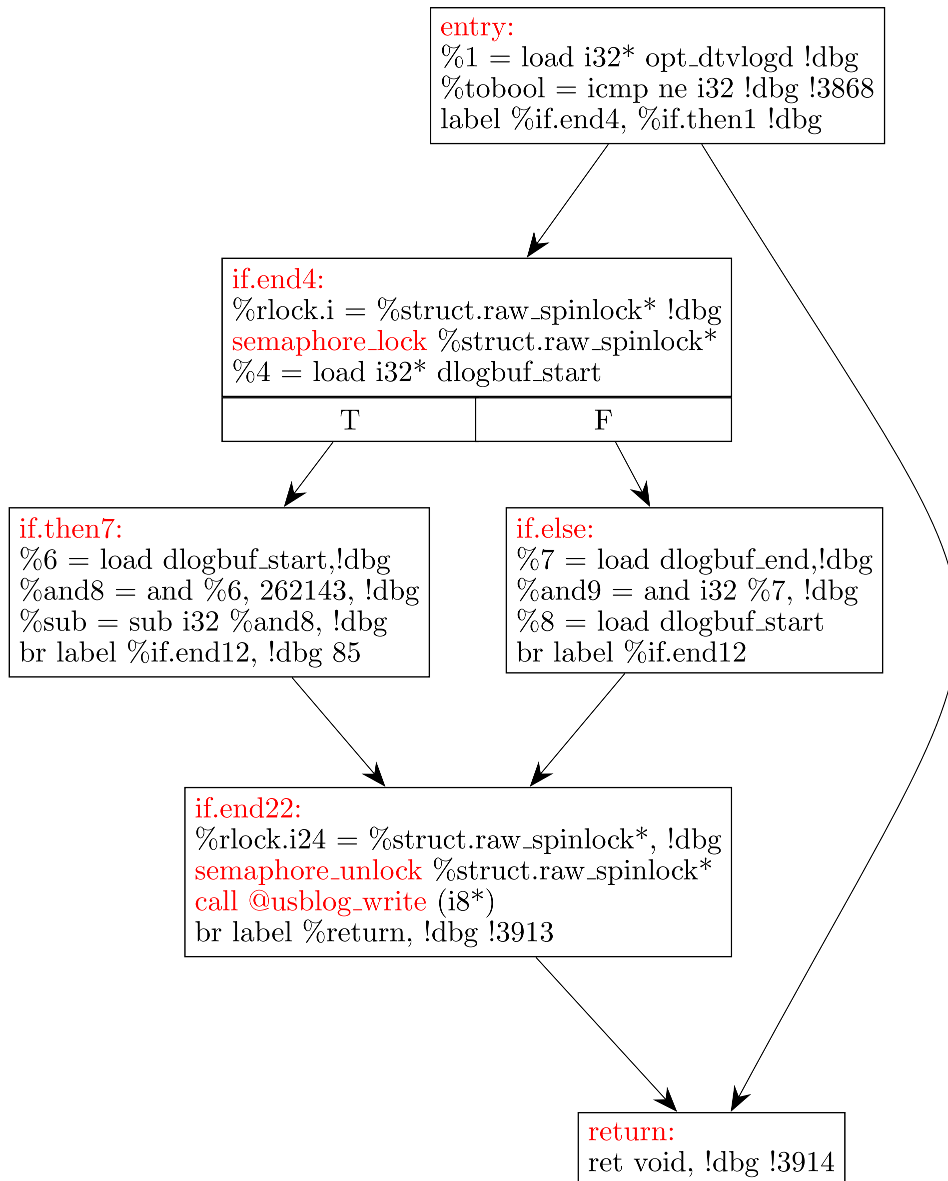


Figure 13: CFG of function “usblob_flush”

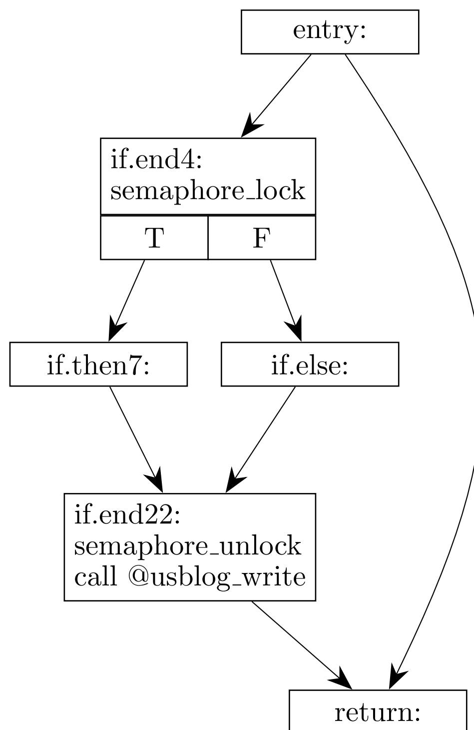


Figure 14: Regenerated CFG of function "usblob_flush" for deadlock analysis.

5.1 Graph Reduction Rules

Rule 1. *Let G=(N,E) be a CFG. Let node $v \neq u$ have a single predecessor u. If v has no synchronization primitive, the transformation is the consumption of node v by node u. The successor edges of node v become successor edges of node u. The original successor edges of node u are preserved except for the edge to node v.*

See Figure 15 for an example of this rule [13].

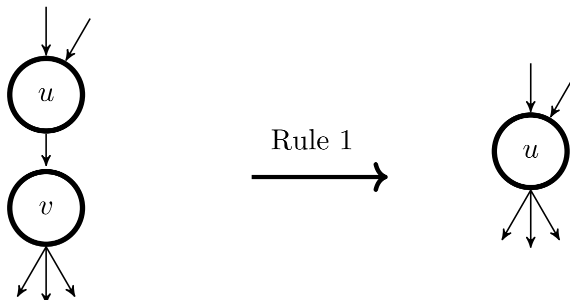


Figure 15: Graphical representation of Rule 1

Rule 2. *Let $u \in N$. This transformation removes the edge $(u, u) \in E$, which is a self-loop. If this edge exists and u has no synchronization primitive, self-loop can be eliminated.*

See Figure 16 for an example of this rule [13].

We perform the CFG reduction of **Rule 1** and **Rule 2** during inlining of the kernel thread functions. We give example as follows.

Example 5. *We perform the CFG reduction to Figure 17. Node 4 contains synchronization primitive in the CFG.* **Rule 1** *is performed to Node 1, Node 2 and Node 3.* **Rule 2** *is performed to Node 6. Performed result is depicted in Figure 18 (a).* **Rule 1** *is performed to Node 5 and Node 6. Performed result is depicted in Figure 18 (b).*

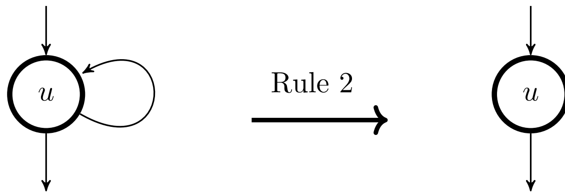


Figure 16: Graphical representation of Rule 2

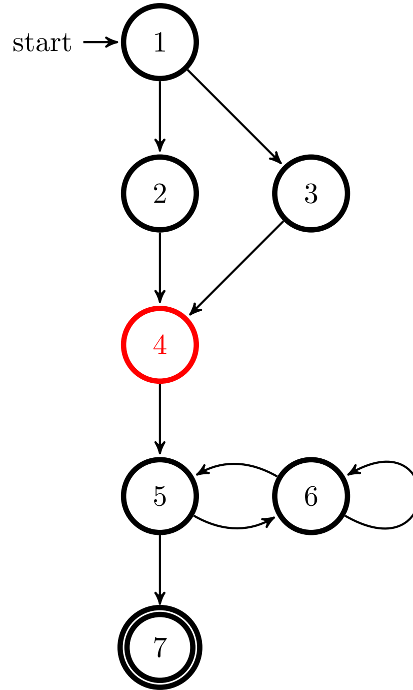
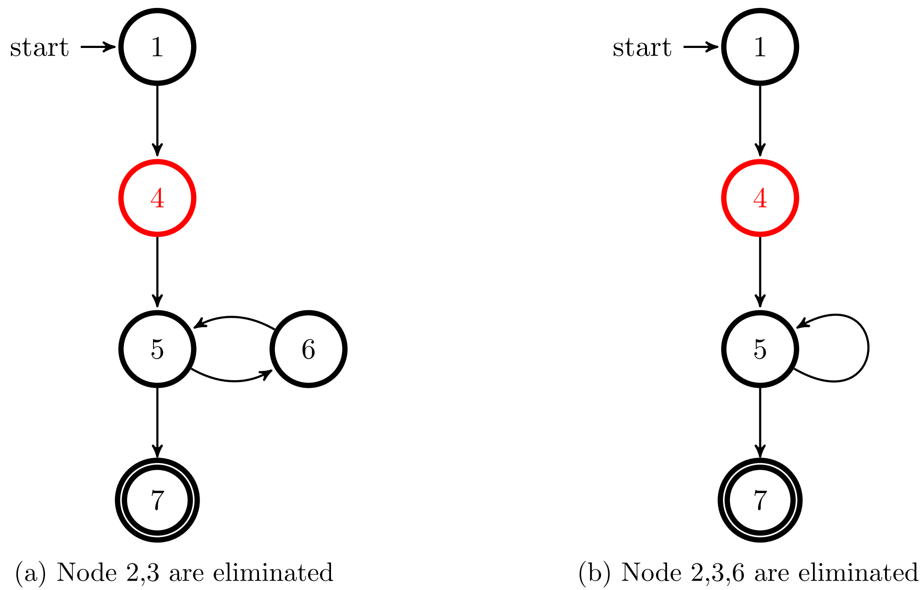


Figure 17: CFG to perform graph reduction



(a) Node 2,3 are eliminated

(b) Node 2,3,6 are eliminated

Figure 18: The intermediate result of CFG reduction

The **Rule 1** and **Rule 2** are performed to Node 5. Final result of CFG reduction is depicted

in Figure 19. In the CFG, Node 2,3,4,5,6 are irrelevant nodes with deadlock detection and eliminated. Node 4 contains synchronization primitive and is the relevant node with deadlock detection.

We perform the CFG reduction of **Rule 1** and **Rule 2** during inlining of the kernel thread function to make problem size tractable for Kronecker algebra calculus.

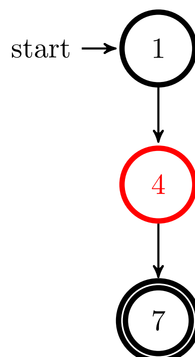


Figure 19: The result of CFG reduction

6 Inlining Called Functions into Linux Kernel Threads

We identified 45 concurrent kernel threads. Modeled CFGs of kernel threads contain procedure calls. We inline all procedure calls in Linux kernel threads as follows.

6.1 Inlining of Procedure Calls in CFGs.

Figure 20 (a) contains the source code of the "main" function and procedure call to "sub" function. Figure 20 (b) contains the source code of the "sub" function called by the "main" function. The "sub" function is inlined to the "main" function.

```
1 int main(void)
2 {
3   int a = 4, b = 10;
4   int result = 0;
5   if(a>b)
6     result = sub(a,b);
7   else
8     result = sub(b,a);
9   return 0;
10 }
```

(a) Source codes of "main" function

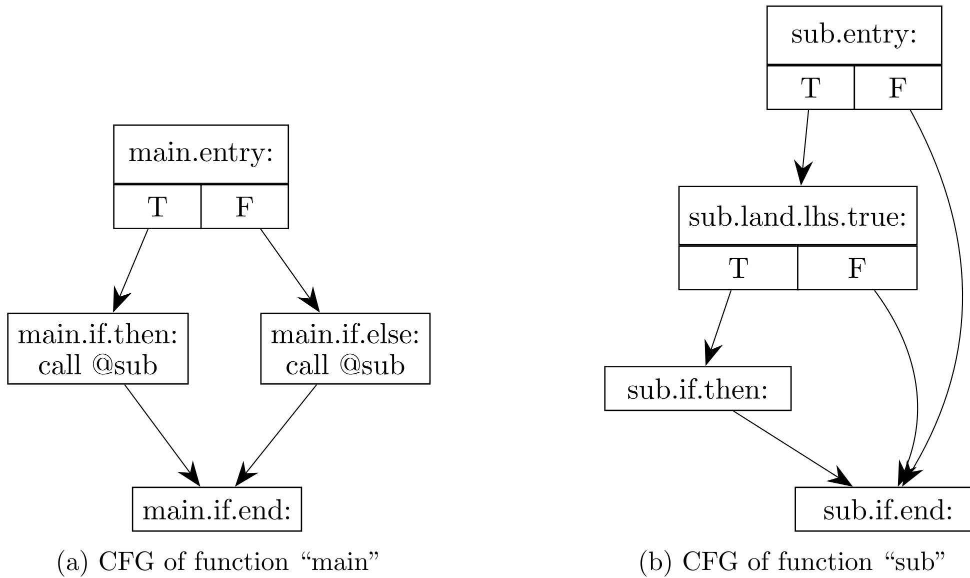
```
1 int sub(int a, int b)
2 {
3   int result = -1;
4   if(a > 0 && b > 0)
5     result = a-b;
6
7   return result;
8 }
```

(b) Source codes of "sub" function

Figure 20: Source code of "main" function which calls the function "sub".

The CFGs of "main" and "sub" functions are depicted in Figure 21. See Figure 21 (a). The "@sub" in the nodes is a procedure call to function "sub" in the "main" function. To inline function "sub", The successor of nodes which have the "@sub" in "main" function is changed to entry node of CFG of function "sub". See Figure 21 (b). The final node of function "sub" has successor node, which is the successor of procedure call node in "main" function.

See Figure 22 for the inlined result of the procedure call. The CFG of function "sub" is inlined into the CFG of the "main" function. Inlined CFG of function "sub" is shown with red colour boxes in Figure 22.



(a) CFG of function "main"

(b) CFG of function "sub"

Figure 21: CFG of "main" function which calls function "sub"

6.2 Inlining Strongly Connected Components.

A directed graph is called strongly connected if there is a path in each direction between each pair of nodes of the call-graph. A strongly connected component (SCC) of a call-graph is a subgraph that is strongly connected.

The circular wait is a closed chain of processes, and each process holds at least one resource needed by the next process in the chain. The circular wait is deadlock condition. SCC is a closed chain of a procedure calls and can make circular wait. Recursive call in SCC can hold one resource under the same resource holding by the previous call.

We compute the SCC in the call-graph and find 31 SCCs in Linux kernel To find circular wait in SCC, we inline the SCC twice. Inlinings twice are sufficient to detect deadlock related with circular wait by Kronecker algebra calculus.

Figure 23 (a) contains the source code of "multiply" function. Figure 23 (b) contains the source code of "divide" function. The function "multiply" calls the function "divide". The function "divide" calls the function "multiply". The function "multiply" and "divide" are SCC.

Figure 24 (a) contains the CFG of "multiply" function. Figure 24 (b) contains the CFG of "divide" function. The "@multiply" is a procedure call to function "multiply" in the "divide" function. The "@divide" is a procedure call to function "divide" in the "multiply" function.

See Figure 25. We inline function "multiply" into "divide". Inlined "multiply" function contains procedure call "@divide" in inlined CFG. We inline "divide" function to procedure call "@divide" in inlined CFG. We inline "multiply" function to procedure call "@multiply" in inlined CFG. As a result, "divide" and "multiply" functions are inlined twice and depicted in Figure 26.

7 Deadlock Detection: Finding Locking Hierarchy Violations

A locking hierarchy is an order on how threads acquire and release the synchronization primitives for a critical section to prevent deadlocks. To prevent a deadlock between concurrent threads, synchronization primitives must be acquired and released in the same order in locking hierarchy. Figure 27 contains locking hierarchy of concurrent threads. One of the threads must acquire the "Lock A" and then acquire the "Lock B" or "Lock C". If one of the threads uses the "Lock B" then needs the "Lock A", "Lock B" should be released before acquiring the "Lock A" to avoid the deadlock.

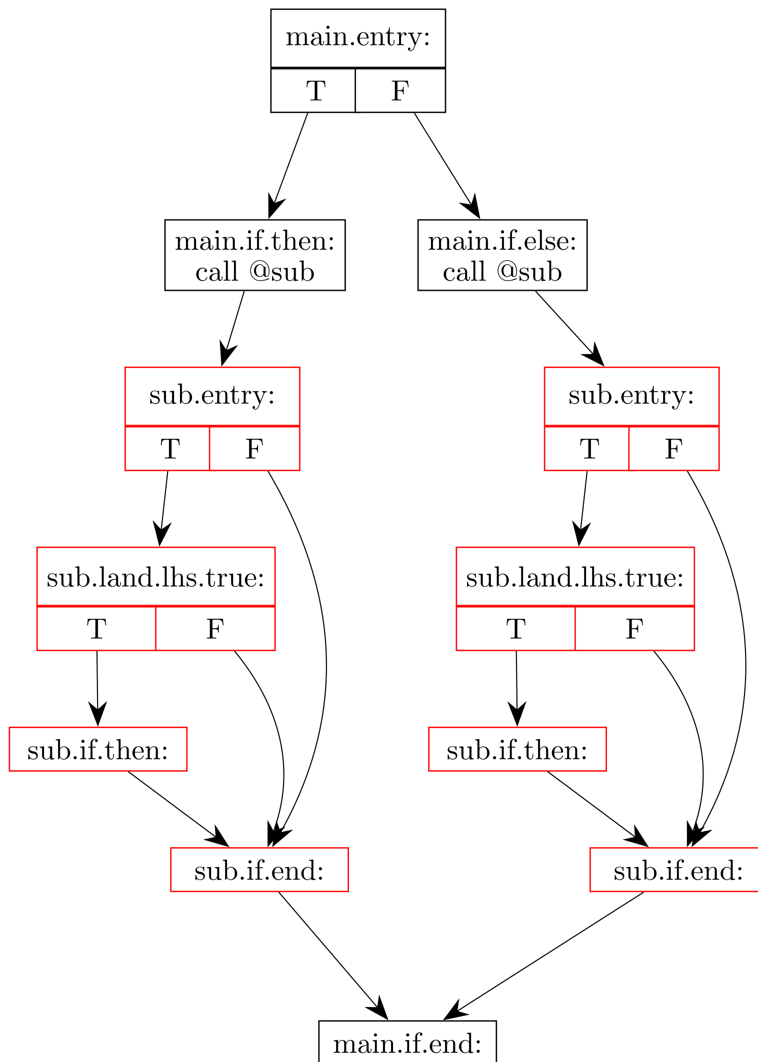


Figure 22: Inlined CFG of main function with function “sub”.

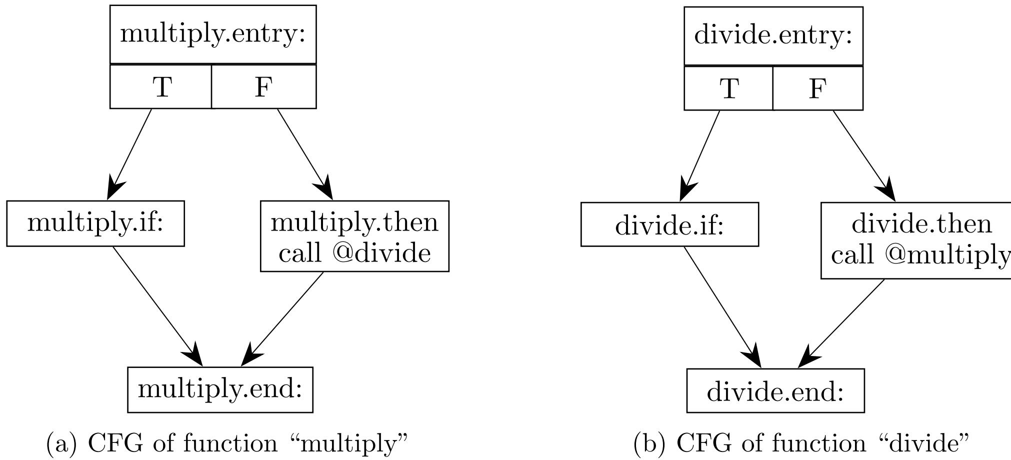
```
1 int multiply(int in)
2 {
3   if(in > LIMITE)
4   {
5     in = INIT;
6   }
7   else
8   {
9     divide(in*3);
10  }
11 }
```

(a) Source codes of “multiply” function

```
1 void divide(int in)
2 {
3   if(in <= 0)
4   {
5     printf("Check input value.");
6   }
7   else
8   {
9     multiply(in/2);
10  }
11 }
```

(b) Source codes of “divide” function

Figure 23: Function “multiply” and “divide” constitute an SCC.



(a) CFG of function “multiply”

(b) CFG of function “divide”

Figure 24: CFGs of “multiply” and “divide” constitute an SCC.

Finding the violation of locking hierarchy in concurrent threads is deadlock detection. We find the violation of locking hierarchy by Kronecker algebra as follows.

7.1 Innermost Synchronization Primitives

To avoid a deadlock, Synchronization primitives should be acquired and released in the same order in locking hierarchy in all kernel threads.

Not all synchronization primitives in kernel thread are relevant with a deadlock. Innermost synchronization primitive is not relevant with a deadlock. Figure 28 contains the innermost locks. The “C.lock()” and “C.unlock()” are used in critical section. No other lock operation is used before “C.unlock()”. “C.lock()” is innermost synchronization primitive in the threads and not the violation of locking hierarchy.

The innermost synchronization primitives can be eliminated from program. The “C.lock()” and “C.unlock()” are eliminated from program. The “B.lock()” and “B.unlock()” are used and no other lock operation is used before “B.unlock()” in “Thread1”. However, The “A.lock()” is used before “B.unlock()” in the “Thread2”. The “B.lock()” is not innermost synchronization primitives. The “A.lock()” and “A.unlock()” are used and “B.lock()” is used before “A.unlock()” in “Thread1”. The “A.lock()” is not innermost synchronization primitive. “Thread1” and “Thread2” have a potential deadlock because of “A.lock()” and “B.lock()”. We find innermost synchronization primitives in kernel threads using Kronecker algebra as follows.

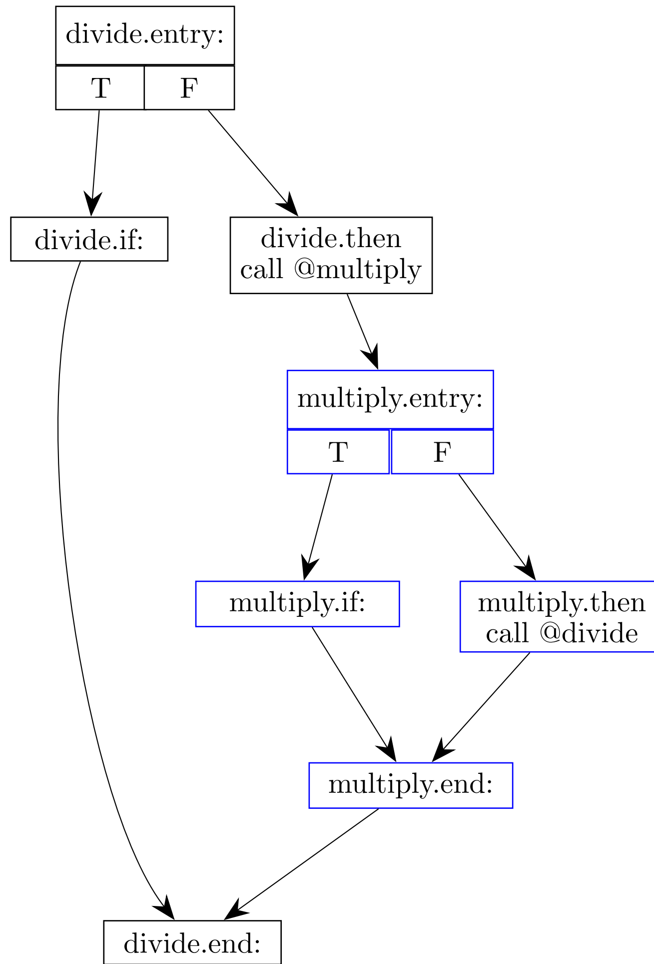


Figure 25: Inlined CFG of “multiply” function into “divide”function.

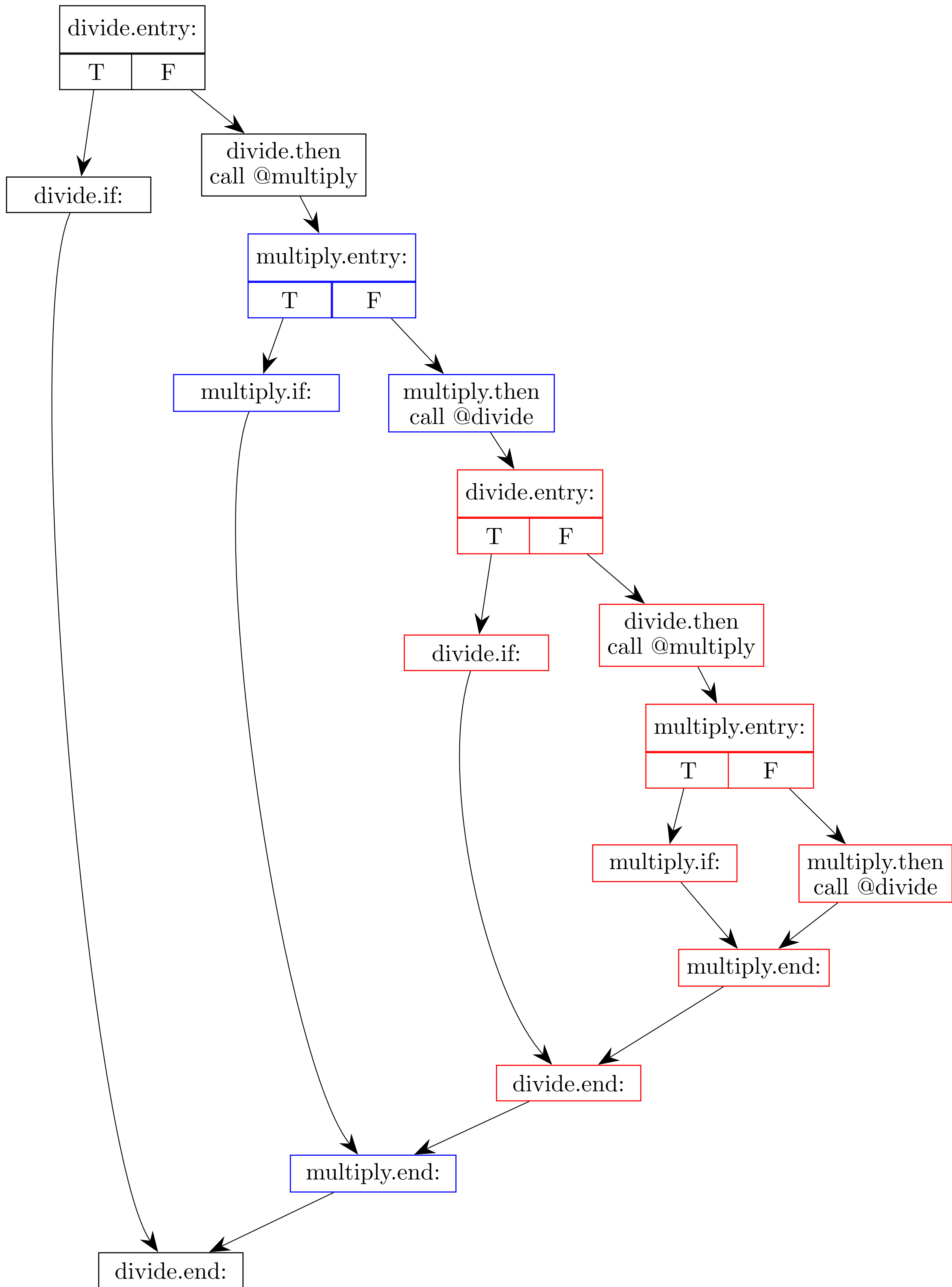


Figure 26: Inlining result of SCC

Lock A $\rightarrow$ Lock B $\rightarrow$ Lock C

Figure 27: Locking hierarchy

```
1 void Thread1(void)
2 {
3   A.lock();
4   B.lock();
5   C.lock();
6   // Critical section.
7   C.unlock();
8   B.unlock();
9   A.unlock();
10 }
```

(a) Locking order in "Thread1" function

```
1 void Thread2(void)
2 {
3   B.lock();
4   A.lock();
5   C.lock();
6   // Critical section.
7   C.unlock();
8   A.unlock();
9   B.unlock();
10 }
```

(b) Locking order in "Thread2" function

Figure 28: A potential deadlock between "Thread1" and "Thread2".

7.2 Regular Expressions

Regular expression of innermost synchronization primitive is depicted in Figure 29. and represented by

$$\Big(\big(\sum_{\substack{j \in J \\ j \neq i}} p_j + \sum_{j \in J} v_j \big)^* p_i \big(\sum_{\substack{j \in J \\ j \neq i}} v_i \big)^* v_i \Big)^*$$

Let i is a selected synchronization primitive to check whether i is innermost synchronization primitive. The J is set of synchronization primitives in the program. The j is the element of J.

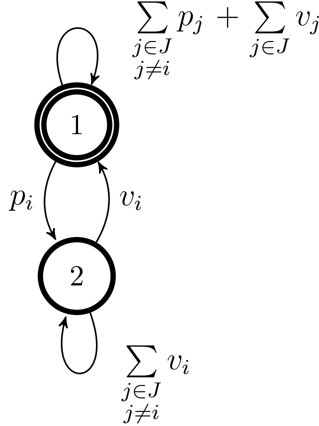


Figure 29: Regular expression of an innermost synchronization primitive

7.3 Example: Finding Innermost Synchronization Primitives

We model Linux kernel to CFGs and inline procedure calls in kernel threads. Kronecker product is the simultaneous execution of the program. Finding Innermost synchronization primitives in kernel thread can be done by Kronecker product.

We use example thread with three semaphores depicted in Figure 30. Figure 30 contains the CFG of thread T_1 and semaphores p_1, p_2 and p_3.

Example 6. *We represent matrix Th_1 of CFG of thread T_1.*

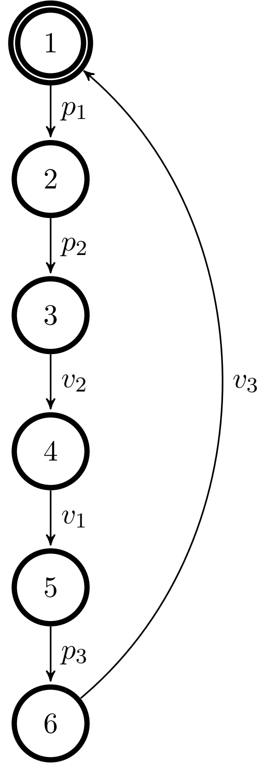


Figure 30: Thread T_1 has semaphore p_1, p_2 and p_3.

$$Th_1 = \begin{pmatrix} 0 & p_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & p_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & v_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & v_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & p_3 \\ v_3 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

First, we check whether semaphore p_1 is the innermost semaphore in thread T_1. Let $i = 1$ and $J = \{p2, p3, v2, v3\}$ in regular expression and represent matrix S_1.

$$S_1 = \begin{pmatrix} p2 + p3 + v2 + v3 & p_1 \\ v_1 & v2 + v3 \end{pmatrix}.$$

Matrix S_1 can be represented to matrix $K_1 + K_2 + K_3 + K4$, where

$$K_1 = \begin{pmatrix} p_2 & p_1 \\ v_1 & v_2 \end{pmatrix},\ K_2 = \begin{pmatrix} p_3 & 0 \\ 0 & v_3 \end{pmatrix},\ K_3 = \begin{pmatrix} v_2 & 0 \\ 0 & 0 \end{pmatrix},\ K_4 = \begin{pmatrix} v_3 & 0 \\ 0 & 0 \end{pmatrix}.$$

We simulate Th_1 and S_1 by Kronecker product as follows.

$$Th_1 \otimes S_1 = Th_1 \otimes (K_1 + K_2 + K_3 + K_4) = Th_1 \otimes K_1 + Th_1 \otimes K_2 + Th_1 \otimes K_3 + Th_1 \otimes K_4$$

$$
= \begin{pmatrix}
0 & 0 & 0 & p_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & p_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & v_2 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & v_2 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & v_1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & p_3 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
v_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & v_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
$$

In graph theory, an isomorphism of two graphs is a bijection between the vertex sets of two graphs. If an isomorphism exists between two graphs, then the graphs are called isomorphic.

The result of $Th_1 \otimes S_1$ is depicted in Figure 31 We find the result of $Th_1 \otimes S_1$ and Th_1 is not isomorphic. As a result, The p_1 is not innermost semaphore in thread T_1.

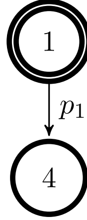


Figure 31: Kronecker product of Th_1 and S_1

The S_2 is matrix about the regular expression of p_2 and v_2. The S_3 is matrix about the regular expression of p_3 and v_3. The S_2 and S_3 are represented as follows.

$$
S_2 = \begin{pmatrix} p1 + p3 + v1 + v3 & p_2 \\ v_2 & v1 + v3 \end{pmatrix}, \; S_3 = \begin{pmatrix} p1 + p2 + v1 + v2 & p_3 \\ v_3 & v1 + v2 \end{pmatrix}.
$$

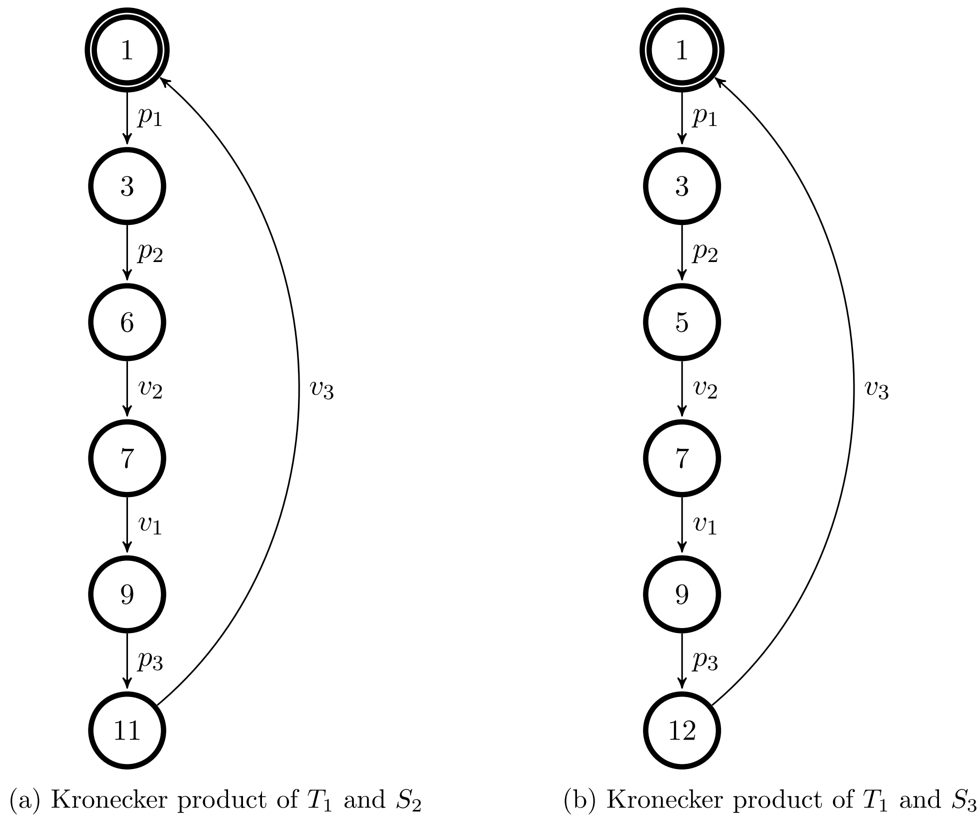
The result of $T_1 \otimes S_2$ and $T_1 \otimes S_3$ are depicted in Figure 32. The result of $T_1 \otimes S_2$ and T_1 are isomorphic. The result of $T_1 \otimes S_3$ and T_1 are isomorphic. As a result, The p_2 and p_3 are innermost semaphores in thread T_1.

The CFG of kernel thread is the large graph. We use static spinlocks as a synchronization primitive for deadlock analysis. To check isomorphism between the kernel thread and static spinlock, we use a tool named Grail+ [9]. Grail+ is symbolic computation environment for finite-state machines and regular expressions. Grail+ provides a function to check the isomorphism between two graphs. We use the function of Grail+ to check isomorphism of kernel thread and static spinlock.

8 Experimental Results

8.1 Locking Hierarchy Violations in the Linux Kernel

We use Linux kernel version 3.10.28. We compile Linux kernel files using Clang compiler and get 1,305 IR files and link them to one. We traverse the linked IR file and find 45 thread functions. These 45 functions are root nodes in the call graph of Linux kernel functions. We inline all procedure calls into its root nodes. We identify 178 static spinlock variables in the Linux kernel as a synchronization primitive for deadlock analysis.



(a) Kronecker product of T_1 and S_2 (b) Kronecker product of T_1 and S_3

Figure 32: Innermost semaphores in thread T_1

We check locking hierarchy about 178 static spinlock variables and all threads. In the Linux kernel, Only one static spinlock is a violation of locking hierarchy. See Listing 6, Static spinlock variable "panic_lock" in "panic" function is a violation of locking hierarchy. There is no unlock operation in the Linux kernel about "panic_lock" variable.

Listing 6: The panic_lock variable in the Linux kernel

```
 9 void panic(const char *fmt, ...)
10 {
11       static DEFINE_SPINLOCK(panic_lock);
12          ...
13
14       if(!spin_trylock(&panic_lock))
15          panic_smp_self_stop();
16          ...
17 }
```

8.2 Control-flow Graph Reductions

We perform the CFG reduction of **Rule 1** and **Rule 2** during inlining of the kernel thread functions. Total reduction in the number of CFG nodes is 92% and depicted in Figure 33.

8.3 Structure Analysis of the Linux Kernel

We analyzed Linux kernel that consists of 1,305 C files. We compiled C files of Linux kernel and generated 1,305 IR files. We linked LLVM IR files to one IR file. We traversed linked LLVM IR file and analyzed the structure of Linux kernel for deadlock detection We found call graph of all procedure calls in Linux kernel. LLVM IR contains CFG information of functions. We generated CFGs of Linux kernel functions. We inlined all procedure calls into its root nodes

No.	Linux kernel threads	Number of nodes before reduction	Number of nodes after reduction	No.	Linux kernel threads	Number of nodes before reduction	Number of nodes after eduction
1	call_help	41,764	3,253	24	agent_work	32,949	2,558
2	aop_buff_task	33,998	2,689	25	age_core	29,302	2,251
3	reply_thread	31,696	2,482	26	_call_user	41,763	3,253
4	wait_help	47,152	3,775	27	cpufreq_dump	29,322	2,258
5	cpufreq_task	29,367	2,256	28	crypto_probe	39,561	3,101
6	crypto_task	39,183	3,052	29	devtmp	31,585	2,423
7	dlogd	29,645	2,285	30	ca_en_thread	31,728	2,461
8	front_thread	29,709	2,264	31	mark_destroy	29,526	2,262
9	event_thread	29,317	2,249	32	rq_thread	29,414	2,264
10	audit_thread	31,696	2,482	33	dbg_work	29,309	2,251
11	debugd	29,369	2,260	34	start_tcp	30,149	2,322
12	khvc	29,918	2,299	35	kjournal	37,262	2,915
13	memory_profile	29,478	2,261	36	mmpd	32,117	2,499
14	scan_thread	33,708	2,628	37	swapd	29,945	2,327
15	kthread	38,929	3,016	38	kthread_watchdog	29,478	2,272
16	kthread_fn	29,340	2,247	39	oop_thread	31,187	2,395
17	queue_thread	29,440	2,256	40	nb_thread	30,509	2,350
18	gp_thread	29,661	2,271	41	er_thread	29,529	2,265
19	scsi_handler	33,126	2,480	42	sd_thread	51,823	4,149
20	irq_thread	105	6	43	pc_symbol	4,680	3,890
21	t2debug	29,366	2,259	44	audit_send	31,701	2,483
22	msg_wait	29,794	2,278	45	rc_msg_wait	29,794	2,278
23	work_thread	30,704	2,345				

Figure 33: Performed CFG reductions per Linux kernel threads

in the call graph and generated inlined CFGs of kernel threads. We analyzed the structure of Linux kernel for deadlock detection as follows.

- Linux kernel contains 45 functions as root nodes of concurrent threads.
- Linux kernel contains 25,623 functions.
- Linux kernel contains 178 static spinlocks.
- Linux kernel contains 31 SCCs.

Total numbers of called functions in the call graph by Linux kernel threads are depicted in Figure 34.

9 Conclusions

In this report, we presented a Kronecker algebra-based deadlock analysis for the Linux kernel. To find deadlock in the program, we modeled all possible thread interleavings. Kronecker algebra simulates thread execution and calculates all interleavings in the program. We introduced Kronecker algebra to find deadlock in the Linux kernel. We provided the example for deadlock analysis using Kronecker algebra.

CFGs are used as FSMs to model the Linux kernel threads. LLVM IR contains CFG information. To generate LLVM IR of Linux kernel, we extended the existing Linux kernel build system for GCC to LLVM compiler using the script.

We used existing LLVM features such as LLVM passes and properties for program analysis. We presented the stand-alone C++ tool that uses the LLVM pass to model the Linux kernel threads. Threads in the Linux kernel are represented by CFGs. Modeled kernel threads are represented by matrices for Kronecker algebra calculus. The Linux kernel has 45 concurrent threads and communicates via synchronization primitives such as static spinlocks. We detect synchronization primitives in the Linux kernel using LLVM annotation property.

No.	Linux kernel threads	Number of called functions	No.	Linux kernel threads	Number of called functions
1	call_help	2,559	24	agent_work	1,958
2	aop_buff_task	1,930	25	age_core	3
3	reply_thread	1,821	26	_call_user	2,559
4	wait_help	2,854	27	cpufreq_dump	1,690
5	cpufreq_task	9	28	crypto_probe	2,410
6	crypto_task	2,381	29	devtmp	1,823
7	dlogd	1,695	30	ca_en_thread	1,814
8	front_thread	1,715	31	mark_destroy	1,705
9	event_thread	1,693	32	rq_thread	12
10	audit_thread	1,841	33	dbg_work	6
11	debugd	15	34	start_tcp	1,742
12	khvc	1,732	35	kjournal	2,144
13	memory_profile	1,705	36	mmpd	1,851
14	scan_thread	1,980	37	swapd	1,728
15	kthread	2,368	38	kthread_watchdog	1,709
16	kthread_fn	1,692	39	oop_thread	1,784
17	queue_thread	1,697	40	nb_thread	1,786
18	gp_thread	1,725	41	er_thread	1,709
19	scsi_handler	1,906	42	sd_thread	3,157
20	irq_thread	6	43	pc_symbol	2,349
21	t2debug	15	44	audit_send	1,821
22	msg_wait	1,722	45	rc_msg_wait	1,722
23	work_thread	1,780			

Figure 34: Number of called functions in the call graph per Linux kernel threads

The Linux kernel has 45 concurrent threads and 23 thousand functions. Our deadlock analysis is for large code-bases. We presented CFG reduction rules and eliminated CFG nodes which are not relevant for deadlock analysis. CFG reduction makes problem size tractable for Kronecker algebra calculus. We are able to handle SCCs that occur in the call-graph of Linux kernel threads. SCCs represent direct or indirect recursive function calls, which may constitute a deadlock. Repeated inlining up to count N of an N-ary semaphore allows us to detect potential deadlocks with SCCs in callgraphs.

To detect violations of a semaphore locking hierarchy, we devised a test based on program path conditions. The test is formulated as a regular-expression-based predicate on all program paths across the CFG of a thread. On a well-formed program path, semaphore p()- and v()-operations match and adhere to a nesting relation. The nesting order reflects the locking hierarchy. The Kronecker product is applied to test whether the CFG of a thread adheres to a given predicate. Via predicates, we iteratively determine the innermost semaphore operation of threads, thereby retrieving and checking the locking hierarchy among threads.

Our experimental results show static spinlock variable under the violation of locking hierarchy in the Linux kernel. CFG reduction results show the total reduction in the number of CFG nodes. The structure of Linux kernel shows information for deadlock analysis.

As for future work, we are going to incorporate mutexes and barriers as further synchronization primitives into our deadlock analysis method. In the Linux kernel source code, aliasing occurs with function pointers passed as arguments with procedure calls. We will perform alias analysis to create a more precise call graph for Linux kernel threads. Our stand-alone C++ tool

to model the Linux kernel is the sequential program. We are going to change our stand-alone C++ tool to parallel processing program for more large-based program analysis.

References

[1] Alexander Aiken and David Gay. Barrier inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 342–354, New York, NY, USA, 1998. ACM.

[2] Alexander Malkis and Anindya Banerjee. Verification of software barriers. volume 47, pages 313–314, New York, NY, USA, February 2012. ACM.

[3] Amir Kamil and Katherine Yelick. Concurrency analysis for parallel programs with textually aligned barriers. In *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing*, LCPC'05, pages 185–199. Springer-Verlag, 2006.

[4] Barbara Gershon Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. volume 18, pages 277–316. ACM Computing Surveys(CSUR), September 1986.

[5] Barbara Gershon Ryder and Marvin C. Paull. Incremental data-flow analysis algorithms. volume 10, pages 1–50. ACM Transactions on Programming Language and Systems (TOPLAS), January 1988.

[6] Bernd Burgstaller and Bernhard Scholz and Johann Blieberger. A symbolic analysis framework for static analysis of imperative programming languages. volume 85, pages 1418–1439. Journal of Systems and Software, June 2012.

[7] Buchholz Peter and Kemper Peter. Efficient computation and representation of large reachability sets for composed automata. *Discrete Event Dynamic Systems*, 12(3):265–286, 2002.

[8] Bernd Burgstaller and Johann Blieberger. Kronecker algebra for static analysis of ada programs with protected objects. In Laurent George and Tullio Vardanega, editors, *Reliable Software Technologies – Ada-Europe 2014: 19th Ada-Europe International Conference on Reliable Software Technologies, Paris, France, June 23-27, 2014. Proceedings*, pages 27–42. Springer LNCS, 2014.

[9] Department of Computer Science at the University of Western Ontario. Grail+ symbolic computation environment, 2016. [Online; accessed 2016-12-8]. http://137.149.157.5/Mirrors/www.csd.uwo.ca/research/Grail.

[10] Duy-Khanh Le and Wei-Ngan Chin and Yong-Meng Teo. Verification of static and dynamic barrier synchronization using bounded permissions. In *Formal Methods and Software Engineering: 15th International Conference on Formal Engineering Methods, ICFEM 2013, Queenstown, New Zealand, October 29 – November 1, 2013, Proceedings*, pages 231–248. Springer Berlin Heidelberg, 2013.

[11] Ana M Erosa and Laurie J Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Computer Languages, 1994., Proceedings of the 1994 International Conference on*, pages 229–240. IEEE, 1994.

[12] Mahdi Eslamimehr and Jens Palsberg. Sherlock: Scalable deadlock detection for concurrent programs. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 353–365. ACM, 2014.

[13] Johan Janssen and Henk Corporaal. Controlled node splitting. In Tibor Gyimóthy, editor, *Compiler Construction: 6th International Conference, CC'96 Linköping, Sweden, April 24–26, 1996 Proceedings*, pages 44–58. Springer LNCS, 1996.

[14] John B Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. volume 23, pages 158–171. Journal of the ACM, January 1976.

[15] Milton Keynes. Kronecker products and matrix calculus: with applications.

[16] Werner Kuich and Arto Salomaa. *Semirings, Automata, Languages.* Springer, 1986.

[17] Robert Mittermayr and Johann Blieberger. Shared Memory Concurrent System Verification using Kronecker Algebra. Technical Report 183/1-155, Automation Systems Group, TU Vienna, http://arxiv.org/abs/1109.5522, Sept. 2011.

[18] NVIDIA Corporation. Cuda parallel computing platform, 2016. [Online; accessed 2016-08-17]. http://www.nvidia.com/CUDA.

[19] OpenCL. The open standard for parallel programming of heterogeneous systems, 2016. [Online; accessed 2016-08-17]. https://www.khronos.org/opencl.

[20] OpenMP Architecture Review Board. Openmp application program interface version 4.5, November 2015. [Online; accessed 2016-11-28]. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[21] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Universitt Hamburg, 1962.

[22] Wolfgang Reisig. *Petri Nets: An Introduction*, volume 4 of EATCS Monographs on Theoretical Computer Science. Springer Press, 1985.

[23] Vugranam C. Sreedhar and Guang R. Gao and Yong-Fong Lee. A new framework for elimination-based data flow analysis using dj graphs. volume 20, pages 388–435. ACM Transactions on Programming Languages and Systems (TOPLAS), March 1998.

[24] Yuan Zhang and Evelyn Duesterwald. Barrier matching for programs with textually unaligned barriers. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 194–204. ACM, 2007.

[25] Yuan Zhang and Evelyn Duesterwald and Guang R. Gao. Languages and compilers for parallel computing. pages 95–109. Springer-Verlag, 2008.