$\mathcal{L}$-SAP: Scalable and Accurate Lock/Unlock Pairing for the Linux Kernel

Ahmed Tamrawi, Iowa State University
Suraj Kothari, Iowa State University

This paper describes $\mathcal{L}$-SAP, a tool that uses a novel scalable and accurate static lock/unlock pairing analysis. It incorporates algorithmic innovations to address the major challenges to advance the state-of-the-art for accurate and scalable pairing analysis. We evaluate $\mathcal{L}$-SAP on three recent versions of the Linux kernel totaling 37 MLOC. $\mathcal{L}$-SAP is able to accurately pair $66,151$ (99.3% of the total) locks in 3 hours with no false negatives. This analysis has led to the discovery of seven synchronization bugs, which were accepted by the Linux community. Our evaluation results show that $\mathcal{L}$-SAP performs strictly and significantly better compared to the currently top-rated Linux kernel device driver verification tool (LDV) [LDV 2015]. $\mathcal{L}$-SAP shows major performance improvements over LDV in: accuracy by reducing the number of unpaired locks $49\times$ fold, and scalability by reducing the analysis time $59\times$ fold.

Categories and Subject Descriptors: F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages - Program Analysis

Additional Key Words and Phrases: Lock/unlock pairing, Static analysis, Linux kernel

ACM Reference Format:
Ahmed Tamrawi, Suraj Kothari, 2015. $\mathcal{L}$-SAP: Scalable and Accurate Lock/Unlock Pairing for the Linux Kernel. *ACM Trans. Softw. Eng. Methodol.* 9, 4, Article 39 (March 2010), 21 pages.
DOI: 0000001.0000001

1. INTRODUCTION

Synchronization problems can be catastrophic - a business-transaction server can crash resulting in a big financial loss, or a safety-critical control system can halt causing loss of lives. With multi-threading and event-driven processing, it is challenging to ensure resilience of software systems to synchronization problems. These problems can elude dynamic analyses and regression testing because their occurrence often depends on intricate sequences of low-probability events [Engler and Ashcraft 2003]. Running a program to examine all possible behaviors is prohibitively expensive and time-consuming. Thus, automated static analyses are crucial to complement testing and dynamic analyses.

An accurate, scalable, and completely automated static analysis has been the holy grail of research. Over the years, many static analysis approaches have been proposed to discover synchronization problems in C programs [Engler and Ashcraft 2003; Sterling 1993; Dillig et al. 2008; Voung et al. 2007; Pratikakis et al. 2006; Cho et al. 2013; Nori et al. 2009]. These state-of-the-art approaches are based on older versions of the Linux kernel (< 4 MLOC) or on medium-sized programs that are orders of magnitude smaller. These approaches have led to new advances in data and control flow

This work is supported by Defense Advanced Research Projects Agency (DARPA), under grant FA8750-12-2-0126.
Author's addresses: A. Tamrawi and S. Kothari, Electrical and Computer Engineering Department, Iowa State University, Ames, Iowa 50010.

DOI: 0000001.0000001

analyses, and new heuristics to apply techniques such as Binary Decision Diagrams (BDDs) to analyze large software [Lhoták 2006]. These advances have pushed further the boundaries of scalability and accuracy. However, there is a fundamental limitation: a general-purpose accurate lock/unlock pairing analysis is not intrinsically scalable as it involves NP hard problems [Church 1936; Turing 1936; Rice 1953]. Our approach is to specifically address commonly occurring roadblocks for scalability and accuracy to arrive at a solution that works well in practice. The Linux kernel code base has unique combinations of specific characteristics that attract researchers and practitioners to challenge their tools [Beyer and Petrenko 2012]. This motivated us to use the Linux kernel code base as a good target system to test our approach. Algorithmic innovations presented in this paper are inspired by the following guiding research question: what scalability and accuracy roadblocks are typical in practical applications, which can be addressed by customized program analysis that achieves accuracy and scalability without being too restrictive?

Consider the lock/unlock pairing analysis: a lock L is paired with an unlock U iff U can release the lock acquired by L. We use the term lock/unlock to refer to a lock/unlock operation (i.e., function call). The existence of unpaired locks on feasible execution paths results in synchronization problems. This pairing of a lock with its corresponding unlocks on all possible feasible execution paths requires the following: (a) a *data flow analysis* to *map* each lock to corresponding unlocks that reference the same *lock object*, (b) a *control flow analysis* to *pair* each lock with mapped unlocks. This is achieved by *identifying* all possible intra- and inter-procedural execution paths that have a lock ensued by a mapped unlock, and (c) a *feasibility analysis* to check the feasibility of an execution path on which a lock is not ensued by an unlock. A general-purpose, completely automated, and accurate analysis is intractable for each of the above three requirements.

We present a novel *scalable* and *accurate* static lock/unlock pairing analysis that is explicitly designed to handle the analysis roadblocks we observed in the Linux kernel. We design a *type-based* analysis and leverage our previous work [Gui and Kothari 2010] to satisfy the analysis requirement (a) to map each lock with a set of corresponding unlocks to perform object-sensitive analysis. To efficiently meet the analysis requirement (b), we design a *novel* control flow graph (CFG) compaction/pruning algorithm to minimize the set of paths that must be examined for path-sensitive accurate analysis. We also design *compact function summaries* for context-sensitive scalable inter-procedural analysis. For requirement (c), among this minimized set of paths, we separate the paths on which a lock is not ensued by an unlock. By construction, this is a necessary and sufficient set of paths that should be checked for feasibility. Thus, the need for feasibility analysis is also minimized by applying it only in the cases where it is needed.

We developed $\mathcal{L}$-SAP, a tool that uses our novel lock/unlock pairing analysis. It analyzes each lock and it reports whether the lock is paired with an unlock or unpaired (i.e., a potential bug). In either case, $\mathcal{L}$-SAP produces evidence so that an analyst can easily cross-check the analysis result. We have applied $\mathcal{L}$-SAP to recent three versions (3.17-rc1, 3.18-rc1, and 3.19-rc1) of the Linux kernel totalling > 37 MLOC. $\mathcal{L}$-SAP is fast and scalable, and it is able to accurately pair $66,151$ (99.3% of the total) locks in 3 hours with no false negatives. Our analysis discovered 7 synchronization bugs that were reported to the Linux community and accepted by them. Compared with the Linux Driver Verification (LDV) tool [LDV 2015], a top-rated verification framework in the competition on software verification (SV-COMP) [Beyer 2012; Beyer 2013; Beyer 2014] in the category of Linux device drivers verification, $\mathcal{L}$-SAP reduces by $49\times$ the number of statically unpaired locks and by $59\times$ the analysis time.

To the best of our knowledge, we are the first to perform lock/unlock pairing analysis of the recent versions of the Linux kernel. Our evaluation results show that $\mathcal{L}$-SAP provides a scalable, practical, and accurate lock/unlock pairing analysis for the Linux kernel. $\mathcal{L}$-SAP is publicly available [Tamrawi 2015] so that other researchers can reproduce our results. $\mathcal{L}$-SAP is developed using Atlas [Deering et al. 2014], which is a platform to develop program comprehension, analysis, and validation tools. Atlas is available for free academic use from EnSoft [Ensoft 2002] corporation.

To summarize, this paper reports the following research contributions:

(1) A novel scalable and accurate lock/unlock pairing analysis algorithm and its implementation in the tool $\mathcal{L}$-SAP (Section 3.4).
(2) Efficient object- and context-sensitive inter-procedural analysis by incorporating: type-based analysis to map a lock to appropriate unlocks (Section 3.1), and compact function summaries (Section 3.4.1).
(3) Efficient path-sensitive analysis with a novel linear-time control flow graph pruning algorithm to compute a compact derivative of CFG called the event flow graph (Section 3.3).
(4) A comprehensive evaluation of $\mathcal{L}$-SAP on three recent versions of the Linux kernel done automatically in 3 hours. The experimental results show that $\mathcal{L}$-SAP correctly pairs 99.3% of the locks and identifies 7 synchronization bugs (Section 4).

The remainder of the paper is organized as follows. We first present the motivation and major challenges in Section 2. Next, Section 3 describes our static lock/unlock pairing analysis algorithm. Section 4 presents the experimental results and Section 5 outlines related work. Section 6 describes the extensibility of our approach and future work. Finally, we conclude in Section 7.

2. MAJOR CHALLENGES AND MOTIVATION

In this section, we discuss the major challenges for lock/unlock pairing in the Linux kernel that motivated the design of our analysis algorithm.

2.1. Path-Explosion

To perform path-sensitive analysis, the challenge lies in the exponential number of control flow paths - 2^n paths with n non-nested two-way branch nodes. Our experimental evaluation on the Linux kernel shows that kernel exhibits many functions with very large number of execution paths. For example, function `register_cdrom`[1] has 20 non-nested branch nodes. This path explosion gets worse with inter-procedural analysis as a path gets split in multiple paths in a called function. To address the path-explosion challenge, $\mathcal{L}$-SAP uses a novel CFG pruning technique that produces a compact derivative of CFG called *event flow graph* (EFG). This is achieved by introducing an *equivalence relation* on the CFG paths to partition/group them into equivalence classes. It is then sufficient to perform the lock/unlock pairing analysis on these equivalence classes (i.e., paths in EFG) rather than on the individual paths in a CFG. We have adopted the graph algorithm by Tarjan *et al.* [Tarjan 1972] to come up with a graph compaction algorithm to form the equivalence classes efficiently. Although the number of paths in a CFG is very large, the number of path equivalence classes is quite small, as seen from our results of the Linux kernel, and that enables a scalable path-sensitive analysis.

[1] http://lxr.free-electrons.com/source/drivers/cdrom/cdrom.c?v=3.19#L316

2.2. Inter-procedural Analysis

In several instances of lock in the Linux kernel, the corresponding unlocks occur several levels down the call chains, thus requiring inter-procedural analysis. The pairing algorithm must handle the following cases: (1) function A contains a lock and calls -through a call sequence- function D which contains an unlock, (2) function A is called by function D through a call sequence (the reference to the locked object is returned upward the reverse call sequence from A to D), (3) A and D are called by a common parent, or (4) A and D are called asynchronously sharing the locked object as a heap object. A mixed combination of these cases can happen because one lock in A can be paired with multiple unlocks in several different functions D on different paths. To address this inter-procedural analysis challenge, $\mathcal{L}$-SAP generates *compact* function summaries to enable an efficient inter-procedural and context-sensitive analysis.

2.3. Pointers

While pointer analysis algorithms have advanced significantly, they cannot be completely accurate because of the fundamental limitation as discussed in the introduction. Typical accuracy hurdles for pointer analysis are pointer arithmetic, heap objects, function pointers, aggregate structures, offset-references, complex pointer references. For example, a highly accurate bit representation to track a pointer loses accuracy when a pointer is passed to a linked list. Further complications are library and system calls. In their review [Godefroid and Lahiri 2012], Godefroid and Lahiri from Microsoft Research note "Indeed, in practice, symbolic execution of large complex programs is rarely fully precise due to external library or system calls, un-handled program instructions, pointer arithmetic, floating-point computations, etc".

To provide scalable and sound analyses and conservatively mitigate inaccuracies of pointer analyses, $\mathcal{L}$-SAP uses *type-based* analysis. Our evaluation on multiple versions of the Linux kernel shows that only a tiny percentage (0.2%) of the locks cannot be paired using our type-based analysis. To achieve accuracy and scalability of the type-based analysis, $\mathcal{L}$-SAP leverages an innovative algorithm [Gui and Kothari 2010] to compute the minimum set of functions for the above four variants of inter-procedural analysis.

2.4. Feasibility Analysis

A path on which a lock is not paired with an unlock may or may not be an error depending on whether the path is feasible or not. Thus, analyzing feasibility of paths is important to avoid false positives. Analyzing path feasibility can incur exponential computation [Ngo and Tan 2007; Navabi et al. 2010; Vojdani and Vene 2009; Dillig et al. 2008; Bodik et al. 1997] because it involves checking satisfiability of branch conditions governing a path. Typical complications for path feasibility analysis are correlations between branch conditions, loops, and inter-procedural paths.

First, our pairing algorithm minimizes the need for performing feasibility analysis by sequencing the analysis steps to produce at the end, exactly those path equivalence classes on which a lock is not paired with an unlock. The feasibility analysis is required only for these cases to avoid false positives. Second, the analysis calculates Boolean expressions that express the conditions under which each lock is not paired with unlock. Finally, $\mathcal{L}$-SAP uses BDDs [Whaley 2010] to check the satisfiability of these expressions to determine whether the paths that contain the unpaired locks are infeasible. To compute correlation between conditions on a given Boolean expression, $\mathcal{L}$-SAP applies an intra-procedural textual equality based analysis. Our evaluation on multiple versions of the Linux kernel shows that only a tiny percentage ($< 0.2\%$) of the unpaired lock cases cannot be handled by our approach to the path feasibility analysis.

3. $\mathcal{L}$-SAP APPROACH

This section describes the approach for static lock/unlock pairing analysis used in $\mathcal{L}$-SAP. This analysis is explicitly designed to pair locks/unlocks for `mutex` and `spin` synchronization mechanisms, both widely used in the Linux kernel. Table I shows the specific locks/unlocks for `mutex` and `spin` synchronization mechanisms in the Linux kernel.

Table I. Locks/Unlocks for `mutex`/`spin` synchronization mechanisms in Linux kernel

Calls	`mutex` synchronization	`spin` synchronization

Lock Calls	`mutex_lock`
	`mutex_trylock`
	`mutex_lock_interruptible`
	`mutex_lock_killable`
	`atomic_dec_and_mutex_lock`
		`spin_lock`
		`spin_trylock`
		`spin_lock_irqsave`
		`spin_lock_irq`
		`spin_lock_bh`

Unlock Calls	`mutex_unlock`	`spin_unlock`
		`spin_unlock_irqsave`
		`spin_unlock_irq`
		`spin_unlock_bh`

Figure 1 shows an overview of our lock/unlock pairing analysis. The pairing analysis has five steps. In the first step, $\mathcal{L}$-SAP *maps* each lock to the set of corresponding unlocks. The mapping is performed via *type-based* analysis, which introduces the notion of *signature*. A lock $L(o)$ is mapped to unlock $U(m)$ iff the objects o and m have the same signature. In the second step, for each signature o, $\mathcal{L}$-SAP creates the *matching pair graph* (MPG_o) as defined in [Gui and Kothari 2010]. The nodes in this graph provide the minimum set of functions for inter-procedural pairing analysis of locks and unlocks with signature o. The directed edges in MPG_o represent call relationships. In the third step, for each function in MPG_o, $\mathcal{L}$-SAP prunes the CFG to produce a compact CFG named the *event flow graph* (EFG). The EFG enables efficient path-sensitive lock/unlock pairing by defining an equivalence relation on the CFG paths such that it is sufficient to examine only one path from each equivalence class. These first three steps set the stage for an efficient pairing algorithm. In the fourth step, $\mathcal{L}$-SAP iterates over the set of signatures to apply the pairing algorithm to each MPG_o and pair the locks and unlocks with signature o. For efficiency, the pairing algorithm computes context-sensitive function summaries using the EFGs computed in the previous step. In the fifth step, $\mathcal{L}$-SAP calculates Boolean expressions for the conditions governing each potential-error path on which a lock with signature o is either: (i) not paired with an unlock with signature o, or (ii) paired with a lock of signature o (a potential deadlock). Then, using BDDs [Whaley 2010], $\mathcal{L}$-SAP examines whether the potential-error paths are feasible.

These five steps are described below in detail.

3.1. Step 1: Lock/Unlock Mapping

The lock/unlock mapping is performed through type-based analysis via the notion of a *signature* such that: a lock $L(o)$ is mapped to unlock $U(m)$ iff the object o and m have the same signature. The signature-based analysis works as follows:

Consider the pointer $\mathcal{P}$ given by the expression: $(a_n \cdots a_3(.||\texttt{->})a_2(.||\texttt{->})a_1)$. In this expression, $\mathcal{P}$ is being accessed through a chain of member-selection C operators (`.` and/or `->`). We define the *hierarchal type* for $\mathcal{P}$ as the tuple $(T_{a_n}, \cdots, T_{a_3}, T_{a_2}, T_{a_1})$, where T_{a_i} denotes the type associated with the member a_i. For example, the hierarchal type for pointer (`x->y->z`) is given by the tuple (X, Y, Z) where $T_x = X, T_y = Y$, and

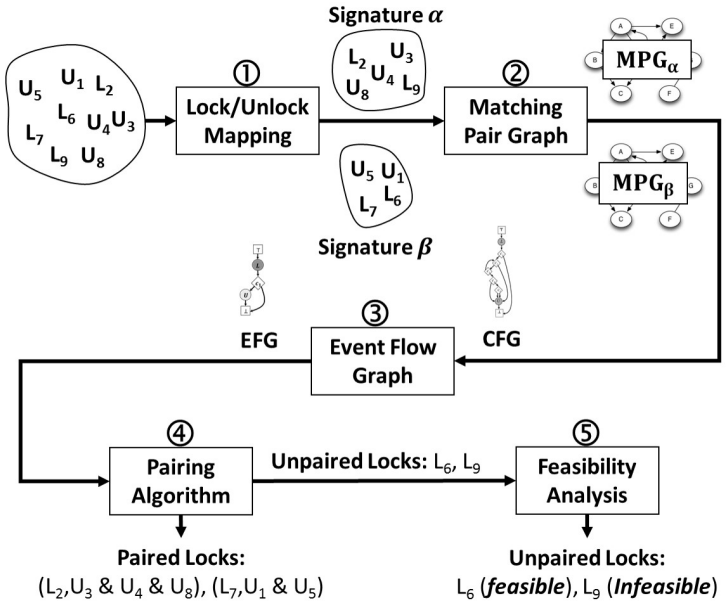

Fig. 1. An Overview of $\mathcal{L}$-SAP Pairing Analysis

$T_z = Z$. For a directly referenced pointer, the hierarchal type is the same as its type. For example, the hierarchal type for pointer (k) is T_k.

We use the term *object signature* (S_o) to denote: (1) the object (variable) *name* in case o is a global variable that is directly referenced, and (2) the *hierarchal type* for the object (pointer) o otherwise. Based on these definitions, $\mathcal{L}$-SAP maps each lock to a set of corresponding unlocks as follows:

(1) Mine all call-sites to lock and unlock based on Table I.
(2) A lock $L(o)$ is mapped to unlock $U(m)$ iff $S_o = S_m$.

Let us illustrate this through an example: let us say that function A has the lock $L(\texttt{x->y->z})$ and function B calls unlock $U(\texttt{l->m->n})$. Then, $\mathcal{L}$-SAP will map the lock $L(\texttt{x->y->z})$ to the unlock $U(\texttt{l->m->n})$ iff both signatures ($S_{\texttt{x->y->z}}$ and $S_{\texttt{l->m->n}}$) are the same. In other words, their hierarchal types are the same where $(T_x, T_y, T_z) = (T_l, T_m, T_n)$. In case of a global lock object, the signature is the global variable's name.

3.2. Step 2: Matching Pair Graph

For each signature o, $\mathcal{L}$-SAP creates a matching pair graph (MPG_o) as defined in [Gui and Kothari 2010]. The nodes in the graph provide the minimum set of functions for inter-procedural pairing analysis of locks and unlocks with signature o. The directed edges in MPG_o represent call relationships. MPG_o captures the four inter-procedural cases identified in Section 2.2. For the fourth case resulting from asynchronous processing, the MPG_o would have two disconnected nodes, i.e., the corresponding functions are not connected by a call sequence because they are invoked asynchronously.

We take a signature o and the associated locks/unlocks as inputs, then compute the matching pair graph MPG_o by running a set of Atlas queries against the system's call graph. This query-based approach is faster compared to recursive traversing of the system's call graph.

Currently, we do not resolve function pointers. This is a source of inaccuracy. Consider the example of lock L_A in function A and unlock U_B in function B where L_A is mapped to U_B as both have the same signature o. Now, assume function C calls A and B via function pointers. Because, we do not resolve function pointers, we will miss function C in MPG_o; it will contain only the functions A and B. The good thing is, the matching pair graph provides the human analyst with hints about the pairing possibility between L_A and U_B. Results in Section 4 show only a small percentage of unpaired locks due to presence of function pointers.

3.3. Step 3: Event Flow Graph

In this step, we prune the CFG to form equivalence classes of CFG paths. This is done by introducing an innovative compact derivative of CFG, called the event flow graph (EFG). Mathematically, the EFG defines an *equivalence relation* on the CFG paths, where each path in the EFG corresponds to a group/class of equivalent paths in the CFG. Two CFG paths are considered *equivalent* if they have the same event trace. Each event trace is a sequence of relevant events on a CFG path. The set of relevant events are defined with respect the lock/unlock pairing analysis. The EFG is the minimal graph for computing all the event traces, i.e., each path in the EFG produces a unique event trace.

We have a novel linear-time algorithm - in the size of a given CFG - to compute EFG without examining each CFG path. The algorithm is modeled as a set of graph transformations starting with a CFG. The algorithm is an adaptation of Tarjan's algorithm to compute strongly-connected components of a directed graph [Tarjan 1972]. Below, we present the details of our CFG pruning algorithm to compute EFG:

(0) Marking Event Nodes. The initial step in our CFG pruning algorithm is marking the event nodes in the given CFG that are relevant to lock/unlock pairing analysis. In our CFG representation, the CFG has unique entry and exit nodes and a CFG node corresponds to a single program statement. Given the matching pair graph MPG_o for signature o and the CFG G_{CFG} for function $f \in MPG_o$, the events of interest for lock-/unlock pairing are as follows: (1) the CFG nodes that correspond to lock/unlock function calls that are associated with signature o, and (2) the CFG nodes that correspond to call-sites for functions in MPG_o.

(1) T-Irreducible Graph Construction: Start with a CFG G_{CFG} with the marked event nodes and transform it into a ***T-irreducible*** graph $G_{\text{T-irr}}$ by applying the following set of basic transformations $T = \{T_1, T_2, T_3\}$ until the resultant graph cannot be further reduced by applying transformations in T.

T_1: Elimination of Non-branching and Non-event Nodes

Let n be a *non-event* node with a single successor m. The T_1 transformation is the consumption of node n by m. Induced edges are introduced so that the predecessors of node n become predecessors of node m. (Figure 2(a))

The T_1 transformation eliminates every node from CFG that is neither a branch node nor an event node. These nodes are removed because they are irrelevant to the analysis.

T_2: Elimination of Self-Loop Edges

Let n be a *non-event* node that has a self-loop edge (n, n). The T_2 transformation removes that edge. (Figure 2(b))

The intuition behind T_2 transformation is: in a loop block that contains no event nodes, execution of the loop is immaterial. Therefore, T_2 removes the self-loop edges.

T_3: Elimination of Irrelevant Branch Nodes

Let n be a *non-event* node that has two or more outgoing edges, all pointing to the same successor m of n. Then the T_3 transformation is the consumption of node n by m and the predecessors of node n become predecessors of node m. (Figure 2(c))

The intuition behind the T_3 transformation is as follows: Imagine the case where a branch node n has only non-event nodes on its branches, and all those branches ultimately merge at node m. If the non-event nodes on those branches are eliminated by the T_1 transformation, all branches will point to node m. At this point, the branching at n is *irrelevant* so the branch node n can be eliminated.

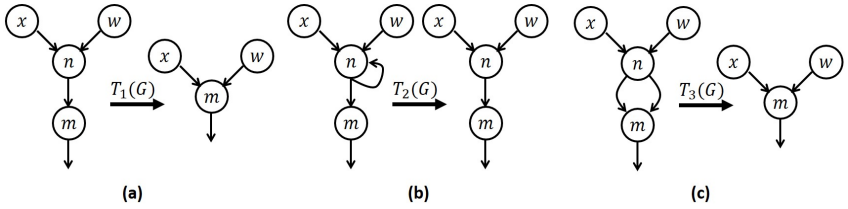

Fig. 2. T-irreducible graph transformations: (a)T_1, (b)T_2, (c)T_3

DEFINITION 1. *G_{CG} is the* condensation graph *of a directed graph G if each* strongly-connected component *(SCC) of G* contracts *to a single node in G_{CG} and the edges of G_{CG} are induced by edges in G.*

(2) Non-Event Condensation Graph Construction: Compute the subgraph G_{I} of $G_{\text{T-irr}}$ induced by its non-event nodes. Then, construct the non-event condensation graph G_{NECG} of G_{I}.

(3) Event Condensation Graph: Construct a new graph G_{ECG} by adding the event nodes in $G_{\text{T-irr}}$ to G_{NECG}. If an edge exists between an SCC and an event node n in $G_{\text{T-irr}}$ then introduce an edge in G_{ECG} between the contracted node for that SCC and the event node n.

(4) Condensed EFG Construction: Transform G_{ECG} into a *T-irreducible* graph G_{cEFG} by applying the set of basic transformation $T = \{T_1, T_2, T_3\}$ as in Step (1). The resultant graph G_{cEFG} after this step is the *condensed EFG*.

(5) EFG Construction: Transform G_{cEFG} into G_{EFG} by expanding each *remaining* contracted SCC in G_{cEFG} back to the original SCC as in $G_{\text{T-irr}}$. The resultant graph G_{EFG} after this step is the EFG.

Figures 3(a-f) illustrate our CFG pruning approach by showing the successive graphs constructed by our CFG to EFG graph transformations, starting with the CFG (graph a) and ending with the EFG (graph f) where the highlighted nodes correspond to the event nodes of interest for pairing $L(o)$ with $U(o)$. It is sufficient to perform the lock/unlock pairing analysis using the pruned CFG (EFG) instead of the CFG. The path analysis becomes much simpler as there are many fewer equivalence classes compared to the corresponding number of paths in a CFG; EFG has 2 paths while CFG has at least 9 paths. Moreover, EFG can minimize computation for checking path feasibility as path's Boolean expressions may get much simpler; the EFG has one branch node compared to the five branch nodes in its corresponding CFG.

CFG to EFG Complexity. The algorithmic complexity of constructing the T-irreducible graph (Steps 1 and 4) is $O(|V| + |E|)$ where $|V|$ and $|E|$ are the respective numbers of nodes and edges in the CFG. For detecting the SCCs in step (2), we use an algorithm by Tarjan [Tarjan 1972] to compute strongly-connected components of a directed graph. The run-time of this algorithm is also $O(|V| + |E|)$, yielding a linear

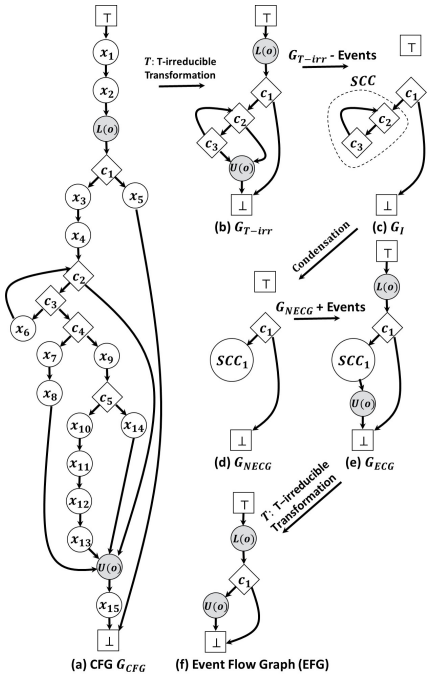

Fig. 3. A transformation from CFG to EFG (CFG Pruning)

run-time complexity of $O(|V|+|E|)$ for our CFG pruning (CFG to EFG transformation). It is important to note that the algorithm is *independent* on the number of paths.
Notes about the EFG: The EFG is a compact derivative of the CFG. It retains only the event nodes and relevant branch nodes that are of interest to our lock/unlock pairing analysis and it has many fewer equivalence classes compared to the corresponding number of paths in a CFG. Thus, the lock/unlock pairing analysis (path analysis) becomes much simpler. Also, the computation for checking path feasibility is minimized as the EFG retains only the subset of the CFG branch nodes that are relevant to the pairing analysis. We recommend reading through our technical report [Tamrawi and Kothari 2014] for more uses, case studies, details, and mathematical proofs on the correctness and compactness of EFGs.

3.4. Step 4: Pairing Algorithm

$\mathcal{L}$-SAP iterates over the set of signatures to apply the pairing algorithm to each MPG_o to pair the locks and unlocks with signature o. For efficiency, the pairing algorithm computes context-sensitive function summaries using the EFGs computed in the previous step. Note that, if a function appears in two matching pair graphs with corresponding signatures o_1 and o_2, then the function would have two contexts as well as two summaries. $\mathcal{L}$-SAP visits the matching pair graph in a bottom-up manner while: (1) computing compact function summaries for each visited function, plugging in the summaries of the callees at call-sites while analyzing the callers, and (2) keeping track of the lock/unlock pairs, unpaired locks, and deadlocks.

3.4.1. Compact Function Summaries. For each function, the function summary is computed by traversing its EFG in a depth-first manner while keeping track of all entry/exit locks/unlocks on all EFG paths. Let us illustrate our approach to computing

compact function summaries. Figures 4(a) and (b) show the EFG for functions f and g. In this example, f calls g at statement: `Call g;`. The function summary for g should retain the information relevant for analyzing f. The pairing analysis is concerned with what follows a given lock: (i) a lock followed by unlock implies pairing, (ii) a lock followed by another lock implies a deadlock, and (iii) a lock not followed by lock or unlock implies an unpaired lock.

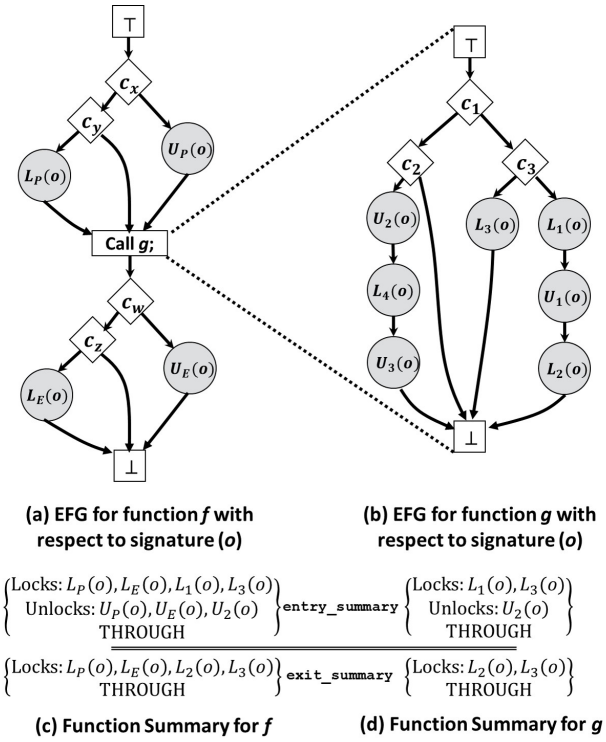

Fig. 4. Compact function summaries for caller (f) and callee (g)

The function summary consists of two sub-summaries: *entry* and *exit* summaries.

The entry summary for g summarizes: (i) the possible unlock(s) in g that can be paired with $L_P(o)$, (ii) the lock(s) in g that cause deadlocks with $L_P(o)$, and (iii) the possibility of not pairing $L_P(o)$ with lock/unlock in g. Case (iii) is possible if there exists a path in g that does not have locks/unlocks. The entry summary for g - denoted by `entry_summary` in Figure 4(d)- includes: the entry locks ($L_1(o)$ and $L_3(o)$), the entry unlock ($U_2(o)$), and the `THROUGH` state denoting the existence of paths in g that do not have any locks/unlocks.

The exit summary for g summarizes: (a) the possible lock(s) in g that can be paired with the lock ($L_E(o)$)/unlock ($U_E(o)$) in f, and (b) the possibility that a lock before calling g can be paired with a lock/unlock after calling g. Case (b) is possible if there is a `THROUGH` state in g. The exit summary for g - denoted by `exit_summary` in Figure 4(d) - includes: the exit locks ($L_2(o)$ and $L_3(o)$) and the `THROUGH` state.

Since the pairing algorithm is traversing the MPG_o in a bottom-up manner, the summary for g is available to compute the summary for f. Figure 4(c) shows the entry and exit summaries for f. Note that: (1) the entry summary for g is part of the entry summary of f because the entry locks/unlocks in g can be the entry locks/unlocks for f

too, and (2) the exit summary of g is part of the exit summary of f as the exit locks in g can be the exit locks in f.

3.4.2. Pairing Algorithm. Listing 1 describes the pairing algorithm. It iterates over the set of signatures and takes as input: the matching pair graph MPG_o, the EFG for each function within MPG_o. Then, it outputs: the set of lock/unlock pairs, unpaired locks, and deadlocks (if any).

```
1  main(MPG_o)
2    functions ← reverse_topological_sort(MPG_o);
3    for(each function in functions)
4      efg ← get_EFG(function);
5      entry_node ← get_entry_node(efg);
6      exit_node ← get_exit_node(efg);
7      node_summary ← {pre_sum: {}, post_sum: {}};
8      traverse_efg(entry_node, node_summary);
9      summary.entry_summary ← pre_sum for entry_node;
10     summary.exit_summary ← post_sum for exit_node;
11     summaries.put(function, summary);
12     if (function ∈ roots(MPG_o)) AND (summary.exit_summary contains a lock)
13       report the lock(s) in summary.exit_summary as unpaired lock(s);
14 end
15
16 traverse_efg(node, ns)
17   if node contains a lock function call
18     if ns.post_sum contains a lock
19       report a potential deadlock between the current lock the lock(s)in ns.post_sum.
20     update the pre_sum and post_sum of ns with the current lock.
21   else if the node is a call-site for function within MPG_o
22     sum ← summaries.get(called function by node);
23     if (ns.post_sum contains a lock) AND (sum.entry_summary contains a lock)
24       report a potential deadlock between the lock(s) in ns.post_sum and the lock(s) in sum.
             entry_summary;
25     ns.pre_sum ← sum.entry_summary;
26     ns.post_sum ← sum.exit_summary;
27   else if node is unlock
28     update the pre_sum and post_sum of ns with this unlock;
29
30   if node is visited before AND ns holds the same summary when the node previously visited
31       return ns.pre_sum;
32
33   pre_sum ← {};
34   for(each s in successors(node))
35     pre_sum += traverse_efg(s, ns.post_sum);
36   if ns.post_summary contains a lock AND pre_summary contains an unlock
37     report lock/unlock pairing between the lock(s) in ns.post_summary and the unlock(s) in
           pre_sum;
38   update ns.pre_sum with pre_sum;
39   return ns.pre_sum;
40 end
```

Listing 1. Pairing Algorithm

The pairing algorithm (Listing 1) starts by visiting the matching pair graph MPG_o in a bottom-up manner (lines 2-13). For each function in MPG_o, the algorithm retrieves the EFG (line 4), gets the entry/exit nodes (lines 5-6), and passes the entry node and its empty node summary to the function `traverse_efg` to start the EFG traversal in a depth-first manner (line 8). Upon the return of `traverse_efg` (lines 9-12), the function summary for `function` is computed as follows: the `entry_summary` is the same as the entry summary (`pre_sum`) for the EFG entry node (line 9), and the `exit_summary` is equivalent to the (`post_sum`) of the EFG exit node (line 10). This function summary is then stored in a global structure (`summaries`) for later use (line 11). Lines (12-13) check whether the current function is one of the roots in MPG_o and the `exit_summary` of its summary contains a lock. If so, it reports the lock(s) in `exit_summary` as unpaired locks.

Lines (17-20) of function `traverse_efg` check whether the currently visited node (`node`) contains a lock function call, if that is the case: the algorithm checks if there is a potential deadlock by checking if the `post_sum` of the previously visited node contains a lock. If so, it reports a potential deadlock between the lock in `ns.post_sum` and the current lock at `node`. Finally, it updates the current node summary with the current lock. In lines (21-26), the algorithm checks if the current node (`node`) is a call-site for a function within MPG_o, then if the `entry_summary` of the called function contains a lock and the `post_sum` of previously visited node contains a lock (line 23): the algorithm reports a potential deadlock between the lock(s) in `ns.post_sum` and the lock(s) in `sum.entry_summary` (line 24). At lines 25-26, the current node summary is updated with the summary of the called function. In case of the current visited node is an unlock function call (lines 27-28), then the algorithm updates the current node summary with the current unlock.

In our pairing algorithm, `traverse_efg` can visit an EFG node multiple times if new information that affects the locking/unlocking analysis is present. At lines (30-31), the algorithm checks whether the current node is visited before and it stops traversing through this node, if the current node summary is the same as the one when previously visited. In other words, if no new information is presented at this node, then no need to re-visit the node. Otherwise, the traversal continues to line 33. Lines (33-35) iterate through the successors of the current node and passes each successor to `traverse_efg` to recursively visit subsequent nodes. Upon the return of `traverse_efg`, the `pre_sum` is updated with the entry summary for each of its successors. Once iterating through the successors is completed (lines 36-37), the algorithm checks whether the `post_sum` of the current node contains a lock and the `pre_sum` of the successors contains an unlock, if that is the case: the algorithm reports locks/unlocks pairing between the lock(s) in `ns.post_sum` and the unlock(s) in `pre_sum` of successors. Then at line 38, the algorithm updates the `pre_sum` of the current node's with the `pre_sum` of successors. Finally, the updated `pre_sum` for the current node is returned (line 39).

3.5. Step 5: Feasibility Check for Potential-Error Paths

A path on which a lock is not paired with an unlock may or may not be an error depending on whether the path is feasible or not. This also applies to a path that has a lock paired with another lock. Thus, checking feasibility of such paths is required to avoid false positives. Unlike existing approaches to static lock/unlock pairing, $\mathcal{L}$-SAP does not encode any information about path feasibility throughout the lock/unlock pairing algorithm. Instead, it only checks the feasibility of *potential-error paths*. These are the paths that have a lock that is : (1) not paired with any unlocks, or (2) paired with a lock with the same signature (potential deadlock). $\mathcal{L}$-SAP applies feasibility analysis to EFG paths instead of CFG paths. This is because the EFG contains fewer paths/conditions than its corresponding CFG. Currently, $\mathcal{L}$-SAP can perform intra-procedural feasibility analysis as follows:

Let the lock L_p in function A be unpaired lock, and p be the path that has L_p. Then, if p is an intra-procedural path, $\mathcal{L}$-SAP will check the feasibility of the EFG path p. If p is an inter-procedural path (i.e., across functions), then $\mathcal{L}$-SAP will only check the feasibility of the sub-path of p contained within A. We call the intra-procedural EFG path that will be checked for feasibility, a *potential-error path*.

To check the feasibility of a potential-error path, $\mathcal{L}$-SAP traverses this path and calculates a *path condition* which is the conjunction (AND) ($\wedge$) of the branch conditions along that path. These are the conditions that must be true for the path to be executed. Then, it translates the path condition to Boolean expressions by assigning a Boolean variable to each condition. To reveal the branch condition correlations in the Boolean expressions, $\mathcal{L}$-SAP uses textual equality. For example, let (`error` $\wedge$ `!error`) be a path

condition for a problematic path. Then, by assigning each condition a boolean variable, the corresponding boolean expression will be: $c_1 \wedge c_2$ where (c_1 = `error`) and (c_2 = `!error`). Based on the textual equality between `error` in c_1 and c_2, $\mathcal{L}$-SAP can infer the correlation: $c_2 = !c_1$. Thus, the resulting boolean expression should be: $c_1 \wedge !c_1$. This final expression's satisfiability will be tested to determine the problematic path's feasibility.

Finally, $\mathcal{L}$-SAP uses BDDs [Whaley 2010] to check the satisfiability of the resultant boolean expressions for each potential-error path to determine its feasibility. In future, we hope to advance our feasibility analysis to cover inter-procedural potential-error paths and to infer more accurate branch correlations via techniques such as: global value numbering [Click 1995] or constant propagation [Wegman and Zadeck 1991].

4. IMPLEMENTATION AND EXPERIMENTAL RESULTS

In this section, we discuss lock/unlock pairing analysis results obtained by evaluating $\mathcal{L}$-SAP on three recent versions of the Linux kernel. These three versions together have 37 million lines of complex multi-threaded C code. To evaluate $\mathcal{L}$-SAP, we have considered three evaluation criteria: (1) competitiveness against the currently top-rated Linux kernel device driver verification tool (LDV) [LDV 2015], (2) analysis speed, and (3) pairing accuracy. Our experiments were done on a 3.0 GHZ Intel Core2 Duo processor machine with 128 GB memory, running Ubuntu Linux 14.

4.1. Implementation and Experimental Setup

We implemented $\mathcal{L}$-SAP using Atlas from EnSoft [Ensoft 2002]. Atlas is a platform to develop program comprehension, analysis, and validation tools. Both Atlas platform and $\mathcal{L}$-SAP are Eclipse plugins. Atlas first compiles the given source code and creates a graph database of relationships between program artifacts. Then, one can interactively comprehend and/or analyze the source code using Atlas's interpreter or by writing Java programs to analyze the source code using Atlas APIs. For more information about Atlas, refer to [Deering et al. 2014] and EnSoft's website [Ensoft 2002]. Atlas is freely available for academic use.

$\mathcal{L}$-SAP leverages the Atlas's query capabilities for lock/unlock mapping (Section 3.1) and generating the matching pair graph (Section 3.2). It also uses Atlas APIs to implement: the algorithm to transform a CFG into an EFG (Section 3.3), the lock/unlock pairing algorithm (Listing 1), and the feasibility analysis (Section 3.5). $\mathcal{L}$-SAP is publicly available for download [Tamrawi 2015].

We used $\mathcal{L}$-SAP to analyze three recent versions (3.17-rc1, 3.18-rc1 and 3.19-rc1) of the Linux kernel along with the device drivers. We enabled all possible `x86` build configurations via `allmodconfig` flag to incorporate all source files. In Table II, columns `LOC`, `Functions`, `Build`, `Nodes`, `Edges`, and `Time` show for each kernel version the numbers for lines of code, functions, build time, nodes and edges in the graph database pre-computed by Atlas and the time for this pre-computation.

Table II. Linux Kernel Artifacts

Kernel	LOC	Functions	Build	Index (Graph Database)		
				Nodes	Edges	Time
3.17-rc1	12.3 M	571,012	15m 12s	43.1 M	133.4 M	2h 14m
3.18-rc1	12.3 M	571,498	15m 48s	43.2 M	133.6 M	2h 5m
3.19-rc1	12.4 M	577,650	16m 29s	43.4 M	134.2 M	2h 15m

We compare $\mathcal{L}$-SAP against the Linux Driver Verification tool (LDV) [LDV 2015] which is the current top-rated tool in the software verification competition (SV-COMP) [Beyer 2012; Beyer 2013; Beyer 2014] in the Linux device drivers verification category. The LDV's developers were generous to provide us with the LDV's results on the same versions of the Linux kernel and the same build configurations we have used.

4.2. Experimental Results

4.2.1. Lock/Unlock Pairing Analysis. $\mathcal{L}$-SAP is applied to pair the locks/unlocks for the `mutex` and `spin` synchronization mechanisms in the Linux kernel (Table I).

Table III shows the comparison of $\mathcal{L}$-SAP and LDV. Column `Type` identifies the synchronization mechanism. Columns `Sigs`, `Locks` and `Unlocks` show the numbers for signatures and lock/unlock call-sites. For instance, in kernel 3.18-rc1, for the `spin` synchronization, the numbers for signatures, lock call-sites, and unlock call-sites are respectively 2180, 14265, and 16917. Note that a lock may be paired with multiple unlocks on different execution paths.

The `Paired` column shows the number of paired locks, i.e., $\mathcal{L}$-SAP can successfully pair a lock with unlock(s) with the same signature on all execution paths. For `spin` signatures, $\mathcal{L}$-SAP can accurately pair 99.4% of the locks in kernel 3.17-rc1 and 99.5% of the locks for the other two versions. For `mutex` signatures, $\mathcal{L}$-SAP can accurately pair: 99.1% of the locks in 3.17-rc1, 98.8% in 3.18-rc1, and 99.3% in 3.19-rc1. For `spin` lock, the percentages of locks paired by LDV are: 63.2%, 64.2%, and 63.9%. For `mutex` lock, the percentages of locks paired by LDV are: 69.7%, 68.8%, and 69.2%. After manually examining `Paired` cases for imprecision assessment, we found that $\mathcal{L}$-SAP does not produce any false negatives and can correctly pair all the cases in `Paired` column.

Column `P-Unpaired` shows the number unpaired locks reported by $\mathcal{L}$-SAP. These are the cases in which potential-error paths are found to be feasible by $\mathcal{L}$-SAP. After manually examining `P-Unpaired` cases, the cases confirmed as actual bugs are reported in Column `A-Unpaired`. We have reported these bugs and they have been accepted by the Linux community.

LDV fails to handle an average of 34% of the locks: 3% due to false positives and 31% due to scalability (LDV crashes). By contrast, $\mathcal{L}$-SAP does not have scalability issue, it correctly handles all but (0.5%) of `spin` locks and (0.9%) of `mutex` locks. We also checked that $\mathcal{L}$-SAP subsumes all correct parings done by LDV, and every lock unpaired by $\mathcal{L}$-SAP is either unpaired or not handled by LDV. Thus, $\mathcal{L}$-SAP is strictly and significantly more accurate than LDV.

Column `Analysis Time` denotes the total time needed for each analysis. $\mathcal{L}$-SAP takes (~ 53) minutes for `spin` locks, and (12 to 15) minutes for `mutex` locks. Overall, $\mathcal{L}$-SAP takes three hours for completing the analysis of three versions of the Linux kernel while LDV takes 156 hours.

4.2.2. Bug Case Study. Listing 2 shows an actual bug found by $\mathcal{L}$-SAP. For brevity, we show only the relevant code. In function `megasas_reset_fusion`, there is a lock at line 4 and its corresponding unlock is at line 10. The bug occurs when the function exits at line 7. In this case, the locked object `instance->reset_mutex` is never unlocked.

```
1  //Source file: drivers/scsi/megaraid/megaraid_sas_fusion.c
2  int megasas_reset_fusion(...)
3  {
4      /*...*/
5      mutex_lock(&instance->reset_mutex);
6      if (instance->adprecovery == MEGASAS_HW_CRITICAL_ERROR){
7          /*...*/
8          return FAILED;
9      }
10     /*...*/
11     mutex_unlock(&instance->reset_mutex);
12     return retval;
13 }
```

Listing 2. Bug in (v3.17-rc1): Unpaired Lock (Line 4)

Table III. `mutex` and `spin` lock/unlock pairing results on Linux kernel versions (3.17-rc1, 3.18-rc1 and 3.19-rc1)

Kernel	Type	Sigs	Locks	Unlocks	$\mathcal{L}$-SAP				LDV			
					Paired	P-Unpaired	A-Unpaired	Analysis Time	Paired	P-Unpaired	A-Unpaired	Analysis Time
3.17-rc1	`spin`	2,165	14,180	16,817	14,097 (99.4%)	83	1	53m 20s	8,962 (63.2%)	30	0	26h 16m
	`mutex`	1,687	7,887	9,497	7,813 (99.1%)	74	1	14m 55s	5,494 (69.7%)	40	0	26h 31m
3.18-rc1	`spin`	2,180	14,265	16,917	14,188 (99.5%)	77	3	53m 57s	9,152 (64.2%)	32	0	30h 22m
	`mutex`	1,664	7,893	9,550	7,801 (98.8%)	92	0	12m 59s	5,427 (68.8%)	48	0	29h 40m
3.19-rc1	`spin`	2,206	14,393	17,026	14,314 (99.5%)	79	2	53m 25s	9,204 (63.9%)	31	0	31h 55m
	`mutex`	1,700	7,991	9,653	7,938 (99.3%)	53	0	15m 29s	5,527 (69.2%)	44	0	29h 12m

The seven bug cases reported in Column A-Unpaired are similar to the case listed here. In earlier versions of the Linux kernel, we have found and reported more complex bugs with inter-procedural paths. Some of these complex bugs persisted over several years through successive versions of the kernel. A complete listing of the seven bug cases reported by $\mathcal{L}$-SAP in Column A-Unpaired and other cases not reported in this paper is in [Tamrawi 2015].

4.2.3. Limitations of Our Pairing Analysis. According to Table III, the percentage of falsely reported unpaired locks and deadlocks is small: (0.5%) for spin locks and (0.9%) for mutex locks. This inadequacy of our analysis can be attributed to the following limitations:

— Inability to process function pointers. $\mathcal{L}$-SAP cannot track the inter-procedural cases in which a function is called via a function pointer.
— Not being able to recognize infeasibility of paths in some cases due to: (a) the lack of inter-procedural feasibility analysis, and (b) the use of textual equality is not advanced enough to find complex correlations between branch conditions.
— The use of signature-based analysis in lock/unlock mapping (Section 3.1) can lead to inaccuracies. We found this to be rare ($< 0.2\%$ of the lock instances).

Listing 3 presents an example that illustrates an analysis roadblock not currently handled by $\mathcal{L}$-SAP. The difficulty is that hash_locks is a pointer to an array of spin objects all with the same type-based signature. In this case, $\mathcal{L}$-SAP falsely reports a deadlock between the lock at line 4 and the lock at line 6 because the spin objects at lines 4 and 6 are not distinguished by the signature-based analysis. We have not found a good way to address this roadblock. Even with generic pointer analyses, it is still challenging to distinguish between different elements of an array.

```
1 //Source file: drivers/md/raid5.c
2 static inline void lock_all_device_hash_locks_irq(...)
3 {
4    /*...*/
5    spin_lock(conf->hash_locks);
6    for (i = 1; i < NR_STRIPE_HASH_LOCKS; i++)
7       spin_lock(conf->hash_locks + i);
8    /*...*/
9 }
```

Listing 3. Signature Problem (v3.19-rc1)

4.2.4. CFG versus EFG. Table IV presents the average and maximum number of nodes in a function before and after CFG pruning (EFG). Our CFG pruning significantly reduces both average and maximum number of nodes, which is essential for the scalability of $\mathcal{L}$-SAP.

Table IV. Nodes count before/after CFG pruning

Kernel	Before Pruning (CFG)		After Pruning (EFG)	
	Average	Maximum	Average	Maximum
3.17-rc1	34	747	7	105
3.18-rc1	34	935	7	119
3.19-rc1	34	935	7	105

5. RELATED WORK

In this section, we survey previous work related to static lock/unlock pairing and CFG pruning techniques.

5.1. Data Races and Deadlocks Detection

Detecting data races and deadlocks is a well-known challenging problem [Engler and Ashcraft 2003]. Although not directly related, there are many papers on dynamic analyses to detect race conditions and deadlocks [Savage et al. 1997; Cheng et al. 1998; Choi et al. 2002; Xie et al. 2013; Dinning and Schonberg 1990; Mellor-Crummey 1991; Perkovic and Keleher 1996; Marino et al. 2009; Prvulovic and Torrellas 2003; Huang et al. 2014; Hsiao et al. 2014]. However, running a program to examine all possible behaviors is prohibitively expensive and time-consuming. Thus, automated static analyses are crucial to complement testing and dynamic analyses.

Relevant static approaches can be divided into: *model checking* methods that emphasize precision, and *static program analysis* methods that emphasize scalability.

5.1.1. Model Checking. Software model checking and its applicability are surveyed in [Jhala and Majumdar 2009]. Classical techniques model systems as labeled transition systems (LTS) and verify properties in temporal logic (TL). Software model checking requires significant effort to model the system (that involves writing an abstract specification for the system in a specific language that the model checker understands). Model checking an entire system such as the Linux kernel (> 12 MLOC) is a monumental task. This is clear from the lock/unlock pairing timing results (Table III) for LDV [LDV 2015] which uses Blast [Beyer et al. 2007] model checker. Other model checkers such as [Henzinger et al. 2004; Qadeer and Wu 2004; Chandra et al. 1999] have been applied to find and prove absence of data races. Abstract model checkers are designed to scale by mapping program states to an abstract domain [Cousot and Cousot 1977]. However, these checkers have limited applicability to special-purpose programs [Henzinger et al. 2014; Ivančić et al. 2008].

5.1.2. Static Analyzers. Warlock [Sterling 1993], Extended Static Checking [Detlefs et al. 1998] (ESC) and ESC/Java [Leino et al. 2000] are static race detection approaches for C, Modula-3 and Java respectively. These approaches utilize a theorem prover to find race conditions. However, these approaches require annotations to inject knowledge into the analysis and to reduce the number of false positives. Thus, applying these approaches to large code bases such as the Linux kernel would require time-consuming and tedious annotation effort and at the end annotations can be erroneous and checking their correctness would be a daunting task for millions of lines of complex code.

Saturn [Dillig et al. 2008] is considered a scalable static analysis engine that is both sound and complete with respect to the user-provided analysis script (abstraction), written in its Calypso language. ESP [Das et al. 2002] is a path-sensitive analysis tool that scales to large programs by merging superfluous branches leading to the same analysis state. However, the lock analysis script bundled with Saturn and ESP is neither sound nor complete, most notably because of its lack of global alias analysis and incomplete function summaries for inter-procedural analyses.

RacerX [Engler and Ashcraft 2003] and Relay [Voung et al. 2007] are static lockset based analyses for C code that scale to large real world programs including the Linux kernel. RacerX [Engler and Ashcraft 2003] uses a top-down approach to compute *absolute locksets* (the set of locks held by the program) at each program point. RacerX is able to run on an older version of the Linux kernel (v 2.5.62 - 1.8 MLOC) in tens of minutes. In contrast, Relay [Voung et al. 2007] uses a bottom-up approach to compute *relative locksets*, which describes the changes in the locks being held rela-

tive to the function entry point, at each program point. Relay was able to analyze an older version of the Linux kernel (v 2.6.15 - 4.5 MLOC) in 72 hours. RacerX reports that in order to scale, the analysis discards valuable information, such as truncating the function summaries. Consequently, discarding possible races. Moreover, both approaches (RacerX and Relay) require significant post-processing of large volume of warnings/false positives to produce evidence for manual validation. Unlike these approaches, $\mathcal{L}$-SAP produces evidence which includes the matching pair graph (MPG) of the minimum set of functions for inter-procedural analysis and the call chains between them, the event flow graphs and a compact summary for each of the functions in MPG. This evidence makes it easy for the human analyst to cross-check the results produced by $\mathcal{L}$-SAP, or to complete the analysis manually for the cases where $\mathcal{L}$-SAP reports potential-error paths but cannot provide conclusive results. A complete listing of all the pairing information (graph evidence) produced by $\mathcal{L}$-SAP is in [Tamrawi 2015].

Locksmith [Pratikakis et al. 2006; Pratikakis et al. 2011] is a sound static race detector for C. It uses a constraint based technique to infer the correlation of memory locations to the locks that protect them. If a shared location is not consistently protected by the same lock, a race is reported. Locksmith analyzes source code fast. However, it finds superficial errors and it produces a number of false alarms, which is about 90% on the device drivers of an older version of the Linux kernel and about 98% on some POSIX applications.

Recently, Cho *et al.* [Cho et al. 2013] proposed a lock/unlock pairing mechanism that combines an inter-procedural analysis and dynamic checking for better detection of races and deadlocks. Their analysis uses dynamic checking to compensate for imperfections in their static analysis. However, the proposed dynamic analysis introduces an overhead for the overall analysis. Moreover, the lock/unlock pairing mechanism has been only applied to code orders of magnitude smaller than the Linux kernel.

To the best of our knowledge, we are not aware of any scalable and accurate program analysis tool that can verify the lock/unlock pairing property for the recent versions of the Linux kernel.

5.2. CFG Pruning Techniques

Event flow graphs are inspired by the work done by Neginhal *et al.* [Neginhal and Kothari 2006]. They developed the C-Vision tool that introduced the notion of *event view*. C-Vision reductions are based on user-input to determine irrelevant nodes/edges to be removed. There is no algorithmic notion to compute the compact CFG. However, in this paper, we provide a linear-time algorithm to compute the event flow graph with regard to the given events of interest to lock/unlock pairing analysis. Thus, leading to significantly-improved scalability.

CFG pruning techniques have been proposed in [Ramanathan et al. 2007; Cho et al. 2013] to overcome the computational complexity of exploring all paths. EFGs can complement their techniques as the EFG transformation achieves further reduction in the graph size. Other pruning techniques have been introduced by Choi *et al.* [Choi et al. 1991] and Ramalingam [Ramalingam 1997] to optimize data flow graphs. While there is some commonality, those techniques are not well-suited for our lock/unlock pairing analysis; the equivalence relation -defined by [Choi et al. 1991; Ramalingam 1997]- is defined with regard to data flow analysis problems. This equivalence relation is different from the one defined by EFG. Path-sensitive analysis requires preserving the unique event traces and that will not be achieved by the cited techniques.

6. EXTENSIBILITY AND FUTURE WORK

We plan to improve our analysis to overcome the limitations outlined in Section 4.2.3. We also plan to extend and reformulate our lock/unlock pairing algorithm to address

many other problems such as: pairing of memory allocation and deallocation to detect memory leaks, or pairing of sensitive sources and malicious sinks to detect malware [Holland et al. 2015]. The underlying accuracy and scalability challenges for these other pairing problems are essentially the same as the lock/unlock pairing. However, the actual analysis difficulties can vary. Our ongoing study of actual difficulties for pairing memory allocation and deallocation calls in the Linux kernel indicates that the problem can be addressed by further enhancements of the algorithmic innovations reported in this paper. Thus, our future work will generalize the pairing algorithm to address a broad class of pairing problems.

7. CONCLUSIONS

$\mathcal{L}$-SAP is a static tool that uses a novel scalable and accurate lock/unlock pairing analysis. It uses algorithmic innovations based on a study of observed difficulties for lock-/unlock pairing in the Linux kernel. We evaluated $\mathcal{L}$-SAP on three recent versions of the Linux kernel along with the device drivers. The evaluation results show major accuracy and scalability improvements over the currently top-rated Linux kernel device driver verification tool (LDV) [LDV 2015]. The analysis using $\mathcal{L}$-SAP has led to the discovery of seven synchronization bugs. For each pairing of a lock with corresponding unlocks with the same signature, $\mathcal{L}$-SAP produces evidence which includes the matching pair graph (MPG) of the minimum set of functions for inter-procedural analysis and the call chains between them, the event flow graphs and a compact summary for each of the functions in MPG. This evidence makes it easy for the human analyst to cross-check the results produced by $\mathcal{L}$-SAP, or to complete the analysis manually for the cases where $\mathcal{L}$-SAP reports potential-error paths but cannot provide conclusive results.

ACKNOWLEDGMENTS

The authors would like to thank thank EnSoft [Ensoft 2002] for providing/assisting us with Atlas.

REFERENCES

Dirk Beyer. 2012. Competition on software verification. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 504–524.

Dirk Beyer. 2013. Second competition on software verification. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 594–609.

Dirk Beyer. 2014. Status report on software verification. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 373–388.

Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. 2007. The software model checker Blast. *International Journal on Software Tools for Technology Transfer* 9, 5-6 (2007), 505–525.

Dirk Beyer and Alexander K Petrenko. 2012. Linux driver verification. In *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*. Springer, 1–6.

Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. 1997. Refining data flow information using infeasible paths. In *Software EngineeringESEC/FSE'97*. Springer, 361–377.

Satish Chandra, Brad Richards, and James R Larus. 1999. Teapot: A domain-specific language for writing cache coherence protocols. *Software Engineering, IEEE Transactions on* 25, 3 (1999), 317–333.

Guang-Ien Cheng, Mingdong Feng, Charles E Leiserson, Keith H Randall, and Andrew F Stark. 1998. Detecting data races in Cilk programs that use locks. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 298–309.

Hyoun Kyu Cho, Terence Kelly, Yin Wang, Stéphane Lafortune, Hongwei Liao, and Scott Mahlke. 2013. Practical lock/unlock pairing for concurrent programs. In *Code Generation and Optimization (CGO)*.

Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. 1991. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 55–66.

Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and precise datarace detection for multithreaded object-oriented programs. In *ACM SIGPLAN Notices*, Vol. 37. ACM, 258–269.

A. Church. 1936. A note on the Entscheidungsproblem. *J. Symb. Log.* 1, 1 (1936), 40–41.

Cliff Click. 1995. Global code motion/global value numbering. In *ACM SIGPLAN Notices*, Vol. 30. ACM, 246–257.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 238–252.

Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-sensitive program verification in polynomial time. In *ACM SIGPLAN Notices*, Vol. 37. ACM, 57–68.

Tom Deering, Suresh Kothari, Jeremias Sauceda, and Jon Mathews. 2014. Atlas: a new way to explore software, build analysis tools. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 588–591.

David L Detlefs, K Rustan M Leino, Greg Nelson, and James B Saxe. 1998. Extended static checking. (1998).

Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, complete and scalable path-sensitive analysis. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 270–280.

Anne Dinning and Edith Schonberg. 1990. *An empirical comparison of monitoring algorithms for access anomaly detection*. Vol. 25. ACM.

Dawson Engler and Ken Ashcraft. 2003. RacerX: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, Vol. 37. ACM, 237–252.

Ensoft. 2002. EnSoft Corp. (2002). Retrieved September 30, 2015 from http://www.ensoftcorp.com

Patrice Godefroid and Shuvendu K Lahiri. 2012. From Program to Logic: An Introduction. In *Tools for Practical Software Verification*. Springer, 31–44.

Kang Gui and Suraj Kothari. 2010. A 2-phase method for validation of matching pair property with case studies of operating systems. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*. IEEE, 151–160.

Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. 2004. Race checking by context inference. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 1–13.

Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L McMillan. 2014. Abstractions from proofs. *ACM SIGPLAN Notices* 49, 4 (2014), 79–91.

Benjamin Holland, Tom Deering, Suresh Kothari, Jon Mathews, and Nikhil Ranade. 2015. Security Toolbox for Detecting Novel and Sophisticated Android Malware. *ICSE* (2015).

Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L Pereira, Gilles A Pokam, Peter M Chen, and Jason Flinn. 2014. Race detection for event-driven mobile applications. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 326–336.

Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal sound predictive race detection with control flow abstraction. *ACM SIGPLAN Notices* 49, 6 (2014), 337–348.

Franjo Ivančić, Zijiang Yang, Malay K Ganai, Aarti Gupta, and Pranav Ashar. 2008. Efficient SAT-based bounded model checking for software verification. *Theoretical Computer Science* 404, 3 (2008), 256–274.

Ranjit Jhala and Rupak Majumdar. 2009. Software model checking. *ACM Computing Surveys (CSUR)* 41, 4 (2009), 21.

LDV. 2015. Linux Driver Verification (LDV) tool. (2015). Retrieved September 30, 2015 from http://linuxtesting.org/project/ldv

K Rustan M Leino, Greg Nelson, and James B Saxe. 2000. ESC/Java user's manual. *ESC* 2000 (2000), 002.

Ondrej Lhoták. 2006. *Program analysis using binary decision diagrams*. Ph.D. Dissertation. McGill University.

Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: effective sampling for lightweight data-race detection. In *ACM Sigplan Notices*, Vol. 44. ACM, 134–143.

John Mellor-Crummey. 1991. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. ACM, 24–33.

Armand Navabi, Nicholas Kidd, and Suresh Jagannathan. 2010. Path-Sensitive Analysis Using Edge Strings. (2010).

Srinivas Neginhal and Suraj Kothari. 2006. Event Views and Graph Reductions for Understanding System Level C Code. In *ICSM*.

Minh Ngoc Ngo and Hee Beng Kuan Tan. 2007. Detecting large number of infeasible paths through recognizing their patterns. 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. ACM, 215–224.

Aditya V Nori, Sriram K Rajamani, SaiDeep Tetali, and Aditya V Thakur. 2009. The Yogi Project: Software property checking via static analysis and testing. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 178–181.

Dejan Perkovic and Peter J Keleher. 1996. Online data-race detection via coherency guarantees. In *OSDI*, Vol. 96. 47–57.

Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. 2006. LOCKSMITH: context-sensitive correlation analysis for race detection. *ACM SIGPLAN Notices* 41, 6 (2006), 320–331.

Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. 2011. LOCKSMITH: Practical static race detection for C. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 1 (2011), 3.

Milos Prvulovic and Josep Torrellas. 2003. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. IEEE, 110–121.

Shaz Qadeer and Dinghao Wu. 2004. KISS: keep it simple and sequential. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 14–24.

G Ramalingam. 1997. *On sparse evaluation representations*. Springer.

Murali Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Path-sensitive inference of function precedence protocols. In *ICSE*.

Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411.

Nicholas Sterling. 1993. WARLOCK-A Static Data Race Analysis Tool.. In *USENIx Winter*. 97–106.

Ahmed Tamrawi. 2015. $\mathcal{L}$-SAP. (2015). Retrieved September 30, 2015 from http://home.engineering.iastate.edu/~atamrawi/l-sap

Ahmed Tamrawi and Suresh Kothari. 2014. Event-Flow Graphs for Efficient Path-Sensitive Analyses. *arXiv preprint arXiv:1404.1279* (2014).

Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

Alan Mathison Turing. 1936. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math* 58, 345-363 (1936), 5.

Vesal Vojdani and Varmo Vene. 2009. Goblint: Path-sensitive data race analysis. In *Annales Univ. Sci. Budapest., Sect. Comp*, Vol. 30. 141–155.

Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. 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*. ACM, 205–214.

Mark N Wegman and F Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 2 (1991), 181–210.

John Whaley. 2010. JavaBDD-Java Binary Decision Diagram Library. (2010).

Xinwei Xie, Jingling Xue, and Jie Zhang. 2013. Acculock: Accurate and efficient detection of data races. *Software: Practice and Experience* 43, 5 (2013), 543–576.

Received February 2007; revised March 2009; accepted June 2009