# Projected Control Graph for Accurate and Efficient Analysis of Safety and Security Vulnerabilities

Ahmed Tamrawi and Suresh Kothari
*Department of Electrical and Computer Engineering*
*Iowa State University, Ames, Iowa, USA*
*Email: {atamrawi, kothari}@iastate.edu*

***Abstract*—The goal of *path-sensitive analysis* (PSA) is to achieve accuracy by accounting precisely for the execution behavior along each path of a control flow graph (CFG). A practical adoption of PSA is hampered by two roadblocks: (a) the exponential growth of the number of CFG paths, and (b) the exponential complexity of a path feasibility check. We introduce *projected control graph* (PCG) as an optimal mathematical abstraction to address these roadblocks.**

**The PCG follows from the simple observation that for any given analysis problem, the number of distinct relevant execution behaviors may be much smaller than the number of CFG paths. The PCG is a projection of the CFG to retain only the relevant execution behaviors and elide duplicate paths with identical execution behavior. A mathematical definition of PCG and an efficient algorithm to transform CFG to PCG are presented.**

**We present an empirical study for three major versions of the Linux kernel to assess the practical benefit of using the optimal mathematical abstraction. As a measure of the efficiency gain, the study reports the reduction from CFG to PCG graphs for all relevant functions for pairing `Lock` and `Unlock` on all feasible execution paths. We built a tool to compute these graphs for 66,609 `Lock` instances. The CFG and PCG graphs with their source correspondence are posted on a website. We used these PCG graphs in a classroom project to audit the results of `Lock` and `Unlock` pairing done by the Linux Driver Verification (LDV) tool, the top-rated formal verification tool for the Linux kernel. Our audit has revealed complex Linux bugs missed by LDV.**

## I. Introduction

A path-sensitive analysis requires that: (a) the execution behavior along each CFG path is accounted individually, and (b) the execution behavior along an infeasible path is excluded. A path-sensitive analysis is critically important for accuracy. Because of its high computational complexity, path-sensitive analysis is avoided in practice by aggregating the execution behaviors [1], [2]. This aggregation is a source of the large number of false positives and negatives in static analyses. Consider a `Lock` followed by a branch node with two branches, with `Unlock` on only one branch. Depending on how the aggregation is interpreted by a particular analysis, the result could be a false negative or inconclusive. It could also be a false positive, if the branch without the `Unlock` is infeasible.

This material is based on research sponsored by DARPA under agreement numbers FA8750-15-2-0080 and FA8750-12-2-0126. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

A path-sensitive analysis can have exponential computational complexity: (1) the number of CFG paths grows exponentially with the number of non-nested branch nodes [3], [4], and (2) path feasibility check can incur an exponential computation [5], [6], [7], [8], [9], [10].

We present a *path equivalence class* (PEC) approach to optimize path-sensitive analysis. Paths are equivalent if their execution behaviors are the same for the purpose of analyzing a given problem. We will give formal definitions in Section IV. For now, let us use an example to illustrate the idea. Consider the problem of pairing `Lock` and `Unlock`. Suppose the control flow graph has a `Lock(X)` for object `X`, followed by a branch node `B1`, followed by another branch node `B2`, and each branch node has two paths. For `B2`, one path has a corresponding `Unlock(X)`. The two `B1` paths do not have any statements relevant for the `Lock`/`Unlock` pairing. If $n$ is the number of non-nested branch nodes then the number of paths is $2^n$. In this example, we have $4$ paths. The $4$ corresponding behaviors relevant to the pairing are: (a) `Lock(X)` followed by `Unlock(X)` on two paths, and (b) `Lock(X)` *not* followed by `Unlock(X)` on two other paths. Thus, the $4$ paths can be partitioned into $2$ equivalence classes corresponding to the $2$ distinct behaviors. Also, note that the `B1` branch node may be irrelevant for performing the path feasibility check. In Sections III and IV , we will formally describe the PEC approach and it's applicability.

Given operations relevant for a problem as a subset of the CFG nodes, the research question is: *what is an optimal abstraction to represent all distinct relevant execution behaviors?* The CFG may have *irrelevant paths*, i.e. paths without relevant operations. The CFG may also have *duplicate equivalent paths*, i.e. paths with the same execution behavior in terms of the sequence of relevant operations along each path. Thus, in general, CFG is not the optimal abstraction for representing relevant execution behaviors. The PEC approach defines an optimal abstraction, called the *projected control graph* (PCG). The PCG incorporates all relevant execution behaviors with one and only one path for each relevant execution behavior. It may seem counterintuitive, but it is shown in Section V how to construct the PCG with all distinct relevant execution behaviors without examining each CFG path.

Another important research question is about the practical impact of the mathematical abstraction: *Is the analysis work*

*significantly reduced in practice by using the PCG instead of the CFG?* Answering this question requires empirical studies with real-world software and complex analysis problems. We report an empirical study with the Linux kernel and $66,609$ instances of a complex analysis problem. The amount of work required for a path-sensitive analysis using the CFG or the PCG is reflected by the size attributes of those graphs. We use them as the metrics for the empirical study.

In Section VI, we present a quantitative study to evaluate the practical impact of the PEC approach. The study covers $66,609$ `Lock` instances from three major releases of the Linux kernel. This study is performed using an automated tool we developed to determine the relevant functions, compute the CFG and PCG for each of those functions, and then measure the size attributes of each CFG and PCG. The CFG and PCG graphs for all the Linux `Lock` instances along with their source correspondence are available through a website [11].

A compact PCG improves efficiency of an automated analysis as well as it facilitates program comprehension. We conducted a study in which a group of students audited the results of the formal verification of `Lock`/`Unlock` pairing by the top-rated Linux Driver Verification (LDV) tool [12]. It was much easier for students to audit using the PCG instead of the CFG. The PCG helped to discover complex Linux bugs missed by the formal verification tool (Sections VI).

The key research contributions are:

- The PEC approach with an optimal mathematical abstraction for accurate and efficient analysis of a wide spectrum of software and safety vulnerabilities (Sections III,IV,V).
- An assessment of the practical impact of using the mathematical abstraction to improve efficiency of a path-sensitive analysis (Sections VI).
- An illustration of how the mathematical abstraction is useful to reveal complex bugs (Sections VI).

## II. A Motivating Example

This example illustrates the grouping of paths in a CFG as a motivation for the proposed PEC approach. The context for the example is the problem of verifying the correct pairing of `Lock` and `Unlock` on all feasible execution paths.

Figure 1 shows the CFG of `hwrng_attr_current_store` function from the Linux kernel. The nodes $\top$ and $\bot$ respectively denote the unique entry and exit nodes added to the CFG. Let `T` and `F` respectively denote the `true` and `false` branches from a branch node.

In this example, we want to group the paths with respect to two *relevant operation nodes* highlighted in gray: `mutex_lock_interruptible (&rng_mutex)` ($e_1$) and `mutex_unlock (&rng_mutex)` ($e_2$). All the paths from the `F` branch of $c_1$ are put in one group because they all have the same sequence $e_1e_2$ of relevant operation nodes. All the paths from the `T` branch of $c_1$ are put in another group because they all have the same sequence $e_1$ of relevant operation nodes. The verification analysis needs to extract

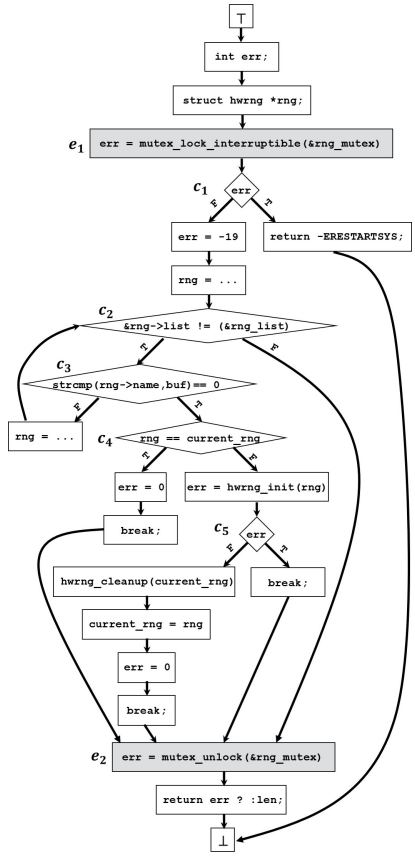


Figure 1. CFG for Function `hwrng_attr_current_store`

from the CFG paths the two distinct possibilities $e_1e_2$ and $e_1$. We will refer to these possibilities as *relevant execution behaviors*. The grouping is a mechanism to extract these distinct possibilities for the purpose of verification. Each relevant execution behavior correspond to a group of paths. We will refer to such a group of paths as a *path equivalence class*.

### A. Representation of Loop Execution Behavior

We use the regular expression $(x_i..x_k)^+$ to represent the execution behavior of a loop that is executed one or more times, with the operation nodes $x_i$ to $x_k$.

The paths due to a loop are accounted as follows. The three possibilities for paths through a loop are: (1) the loop is not entered because the loop entry condition is not satisfied, (2) at the very first iteration of the loop, a `break` is executed, and (3) the loop without a `break` is executed at least once and then a `break` may or may not be executed. Suppose the loop has $k$ `break` points, then the paths are counted as: 1 path for the first possibility, $k$ paths for the second possibility, and $(k+1)$ paths for the third possibility. In the CFG shown in Figure 1, since the loop has 3 `break` points, we count 8 paths due to the loop. The CFG has a total of 9 paths partitioned into 2 path equivalence classes. One equivalence class with 8 paths corresponds to the relevant execution behavior $e_1e_2$. The other path equivalence class has only one path with the relevant execution behavior $e_1$.

Let us illustrate how loops containing relevant operations are handled. Denote by $L_1$ and $U_2$ the `Lock` and the `Unlock` in the above example. For the sake of illustration, suppose we have inside the loop an `Unlock` $U_1$ followed by a `Lock` $L_2$ before the normal loop exit. Now, we have 3 distinct relevant execution behaviors: $L_1$, $L_1(U_1L_2)^+U_2$, and $L_1U_2$. The number of paths in three equivalence classes are respectively 1, 4, and 3. The $c_3$ branch node becomes relevant because it has different execution behaviors on its branches. Unlike the previous analyses such as [8], [13], the PEC approach does not use a heuristic based on loop unrolling.

## III. Software Safety and Security Problems

This section describes the class of software safety and security problems for which the PEC approach is applicable.

*Definition 1:* **2-event matching -** Verify that an event $e_1(O)$ is succeeded by event $e_2(O)$ on every feasible execution path, where the two events are operations on the same object $O$.

Besides the `Lock`/`Unlock` pairing problem which is described in this paper, the 2-event matching covers several problems such as memory `Allocation`/`Deallocation` pairing, or file `Open`/`Close` pairing. A number of vulnerabilities listed by the MITRE Corporation [14] can be viewed as 2-event problems.

*Definition 2:* **2-event anti-matching -** Verify that an event $e_1(O)$ is *not* succeeded by event $e_2(O)$ on any feasible execution path, where the two events are operations on the same object $O$.

Anti-matching covers several software security problems defined according to *confidentiality*, *integrity*, and *availability* (CIA) model [15]. A confidentiality verification problem could be defined as: a *sensitive source* must *not* be followed by a *malicious sink* on any feasible execution path. Similarly, an integrity verification problem could be defined as: an *access of sensitive data* must *not* be followed by a *malicious modification of sensitive data* on any feasible execution path.

The following defines a general class of problems for applying the PEC approach.

*Definition 3:* $n$**-event verification -** Verify on every feasible execution path, that the occurrence of events on the path follow the acceptability test defined by a finite state machine (FSM) $\phi(\mathcal{E})$, where $\mathcal{E}$ is a set of $n$ events that operate on the same object $O$.

## IV. The PEC Framework

In this section, we describe the PEC framework and its mechanism for addressing the challenges for an accurate and efficient analysis.

*Definition 4:* A **Control Flow Graph** (CFG) of a program is defined as $G = (V, E, \top, \bot)$, where $G$ is a directed graph with a set of nodes $V$ representing the program statements and a set of edges $E$ representing the control flow between statements. $\top$ and $\bot$ denote the respective unique entry and exit nodes of the graph.

### A. CFG Paths and Execution Behaviors

A CFG has two types of nodes: **operation nodes** and **branch nodes**. For a branch node $c$, a **branch edge** is an out-coming edge of $c$. A **CFG path** starts at the $\top$ node, goes through a sequence of operation and branch nodes, and ends at the $\bot$ node. We will use the term **governing condition for a path** for the condition associated with each of the branch nodes along the path. Each CFG path has a unique **governing condition expression** consisting of $c_i$ or $\bar{c_i}$ corresponding to that condition being `T` or `F` for that path. A **feasible path** means that there is a possible program execution where the governing condition expression for the path is satisfied. A path with a loop is a corner case for the governing condition expression. Suppose a loop is entered when a condition $c_i$ is `T`, after some iterations of the loop the condition $c_i$ becomes `F` and the loop terminates. We use $\hat{c_i}$ in the path expression to handle this case.

*Definition 5:* The **execution behavior** along a path $P$ (denoted by $B(P)$) is represented by a regular expression consisting of the labels of the operation along the path where each operation and branch node is labeled with a unique identifier.

A subset of the CFG operation nodes are relevant for analyzing a given problem instance. We will refer to these nodes as the **relevant operation nodes**. For example, given a call site `Lock(X)`, the corresponding `Unlock(X)` call sites are relevant operations. In presence of aliasing, the additional relevant operations would be the CFG nodes where `X` is aliased and `Unlock(`$\bar{X}$`)` where $\bar{X}$ is an alias of `X`.

*Definition 6:* The **relevant execution behavior** along a path $P$ (denoted by $RB(P)$) is represented by retaining only the relevant operation labels in the execution behavior regular expression for $P$.

**An example to illustrate the representation of execution behavior:** Let us illustrate the representation of execution behavior using the CFG first shown in Figure 1 and redrawn in Figure 2(a). The highlighted nodes are the relevant operation nodes. Operation nodes are labeled $x_1$ through $x_{15}$, and the branch nodes are labeled $c_1$ through $c_5$ . Figure 2(b) shows a CFG path $P_1$ with the governing condition expression $\bar{c_1}c_2c_3\bar{c_4}\bar{c_5}$ where $\bar{c_i}$ means the path is associated with the condition for $c_i$ being `F`. The $B(P_1)$ is: $\top x_1x_2e_1x_3x_4x_9x_{10}x_{11}x_{12}x_{13}e_2x_{15}\bot$. Figure 2(c) shows a CFG path $P_2$ with a loop; the $B(P_2)$ is: $\top x_1x_2e_1x_3x_4(x_6)^+x_7x_8e_2x_{15}\bot$. The governing condition expression for $P_2$ is $\bar{c_1}\bar{c_2}\hat{c_3}\bar{c_4}$. Note that relevant execution behavior is the same for $P_1$ and $P_2$, i.e., $RB(P_1) = RB(P_2) = e_1e_2$. We may omit $\top$ and $\bot$ while illustrating execution behaviors.

*Definition 7:* **Successors of a node** $u$ in a directed graph $G$, denoted by *suc(u)*, consist of the set of nodes $v \neq u$ such that $\exists$ an edge $(u, v)$.

*Definition 8:* **Successors of a subgraph** $S$ in a directed graph $G$, denoted by *suc(S)*, consist of the set of nodes $v \notin S$ such that $v =$*suc(u)* for $u \in S$.

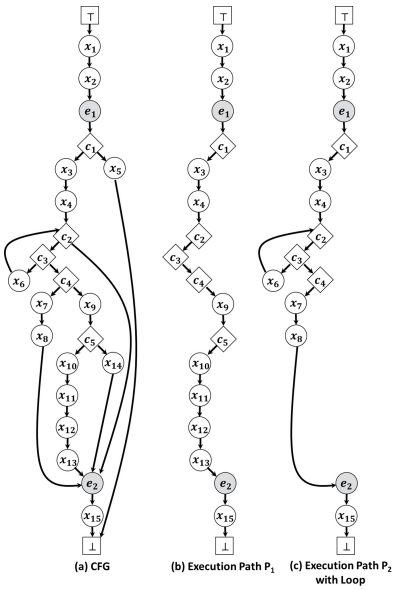


Figure 2. Execution paths in a CFG

*Definition 9:* A node $c$ is an **irrelevant branch node** if there exists a subgraph $S$ containing $c$ and all branch edges of $c$ such that $S$ does not have nodes with relevant operations, and $S$ has a unique successor, i.e., $|suc(S)| = 1$. If such a set $S$ does not exist for a branch node then it is a **relevant branch node**.

*Definition 10:* The **projected control graph** (PCG) of a CFG $G$ is the node-induced subgraph of $G$ consisting of the nodes for relevant operations, the relevant branch nodes, and the entry ($\top$) and exit ($\bot$) nodes.

*Remark 1:* The PEC approach does not collapse distinct relevant operation nodes of CFG. Consider the following example: suppose the node $x_i$ is on one branch and the node $x_j$ on the other branch and both nodes have the same relevant operation `Unlock(X)`. The PEC approach retains both nodes. An extension of the PEC approach would be to define a tighter path equivalence relation by introducing the notion of semantically equivalent relevant operation nodes.

### B. Accuracy and Efficiency with the PEC Approach

The classic challenges for an accurate and efficient analysis are:

1) The *path multiplicity challenge* due to the exponential growth of the number of CFG paths.
2) The *path feasibility challenge* due to the exponential complexity of checking feasibility of unsafe paths.
3) The *object tracking challenge* due to the high computational complexity of the pointer analysis to track the object $O$ for the set of relevant operations.

The PEC approach addresses the *path multiplicity* challenge. There is a one-to-one correspondence between PCG paths and the distinct relevant execution behaviors. Because of this correspondence, analyzing all CFG paths is equivalent to analyzing all PCG paths. Note that the number of distinct behaviors is the lower bound on the number of paths that must be analyzed. The PCG achieves this lower bound (Section V).

The PEC approach addresses the *path feasibility* challenge. There exists a feasible CFG path $P$ with a relevant execution behavior $B$ if and only if there exists a path $P'$ with the behavior $B$ that is feasible with respect to the relevant branch nodes on $P$ (Section V). Thus the PEC approach simplifies the *path feasibility* by retaining only the relevant branch nodes.

Let us now understand how the PEC approach helps to address the *object tracking* challenge. Recall that we start with a set of relevant operations and a FSM. The PEC approach uses an almost linear-time points-to analysis [16] to select the additional relevant operation nodes for constructing the PCG. These additional nodes are for operations that alias the object $O$ on which the FSM events operate.

By keeping track of aliasing, the PCG can be used to track individual paths during an inter-procedural analysis. Suppose an event $e_1(p)$ occurs in a function $f_1$, and $p$ is passed as a parameter $q$ to another function $f_2$ called by $f_1$. Next, an event $e_2(q)$ occurs in function $f_2$. Let $\hat{G_1}$ and $\hat{G_2}$ denote the PCG of $f_1$ and $f_2$ respectively. To perform interprocedural analysis $\hat{G_2}$ is linked to $\hat{G_1}$ at the node where $f_2$ is called in $f_1$. At this call-site node in $\hat{G_1}$, the analysis enters and traverses through $\hat{G_2}$ and at the exit returns to to continue the traversal in $\hat{G_1}$. The linkage between $\hat{G_1}$ and $\hat{G_2}$ can be established because of the unique entry ($\top$) and exit ($\bot$) nodes in each CFG.

```
1   struct  temp    *obj;
2   int done, err1, err2;
3
4   err1 = mutex_lock(&O1);
5   err2 = mutex_lock(&O2);
6   done = f();
7
8   if (done) {
9       temp = &O2;
10      mutex_unlock(&O1);
11  } else {
12      temp = &O1;
13      mutex_unlock(&O2);
14  }
15  /* g() does mutex_unlock(temp) */
16  g(temp);
```

Figure 3. An illustration for the object tracking challenge

We will use the example shown in Figure 3 to illustrate how the PEC approach helps to address the *object tracking* challenge without resorting to expensive pointer analysis. The example involves two `Lock` instances for objects $O1$ and $O2$ respectively. Let us use the line numbers as the labels for the relevant operation nodes. The pointer analysis captures lines 12 and 16 as relevant operation nodes for the instance $O1$. Similarly, the lines 9 and 16 are captured for the instance $O2$. The two relevant execution behaviors $B_{11}$ and $B_{12}$ for the instance $O1$ are: $B_{11}$ = 4,10,16 and $B_{12}$ = 4,12,16. The two relevant execution behaviors $B_{21}$ and $B_{22}$ for the instance $O1$ are: $B_{21}$ = 5,9,16 and $B_{22}$ = 5,13,16.

**Accuracy:** Observe that the two execution behaviors for each of the instances suffice for performing a correct verification analysis. We will illustrate the correctness for the

$O1$ instance, the other instance follows the same pattern. Since line 12 is *not* included in behavior $B_{11}$, line 16 can be suppressed by the verification analyzer. Since line 12 is included in behavior $B_{12}$, line 16 will *not* be suppressed by the verification analyzer. The analyzer could perform inter-procedural analysis by linking the PCG for the function `g`. Thus, by accounting for both behaviors, the verification analyzer can perform an accurate analysis to conclude that the instance $O1$ is safe.

**Efficiency:** Let us now discuss efficiency. A light-weight points-to analysis such as [16] suffices to capture the lines 12 and 16 as the relevant operations for the instance $O1$. The PEC approach resolves the ambiguity due to the points-to analysis by separating the execution behaviors along paths. With the PEC approach, we have not seen a need for performing an expensive points-to analysis for accuracy. An interesting topic for research is to conduct studies to assess whether the combination of an inexpensive points-to analysis and the PEC approach can always achieve high accuracy without resorting to an expensive points-to analysis.

Note that line 6 is not captured as a relevant operation although it implicitly affects the relevant control flow. In this example, the implicit effect does not matter. If line 6 were to be included, it would lead to unnecessary analysis. On purpose, the PEC approach omits implicit control effects in the first pass. The verification analyzer can subsequently perform the implicit effect analysis only where it is needed.

## V. CFG to PCG Transformation

We present an efficient algorithm to compute PCG. It uses Tarjan's algorithm to compute strongly-connected components of a directed graph [17].

**Step 1: T-Irreducible Graph** Reduce $G_{\text{CFG}}$ to the *T-irreducible* graph $G_{\text{T-irr}}$ by applying the following basic transformations $T = \{T_1, T_2, T_3\}$ until the resultant graph cannot be further reduced.

*$T_1$: Elide Non-Relevant Operation Nodes*

Let $n$ be a *non-relevant operation* 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 4(a))

*$T_2$: Elide Self-Loop Edges*

Let $n$ be a *non-relevant operation* node that has a self-loop edge $(n, n)$. The $T_2$ transformation removes that edge (Figure 4(b)). When a loop block does not contain relevant operation nodes, execution of the loop is immaterial. $T_2$ transformation elides such loops.

*$T_3$: Elide Irrelevant Branch Nodes*

Let $n$ be a *branch* node without a relevant operation such that all branch edges lead to the same successor $m$ of $n$. The $T_3$ transformation elides $n$ and its branches so that the predecessors of $n$ become predecessors $m$ (Figure 4(c)). $T_3$ elides branch nodes that become vacuous after $T_1$ elides all non-relevant operation nodes.

We examined the irreducible graphs obtained by applying the three transformations and found examples of complex CFGs where some of the *irrelevant branch nodes* (defined in Section IV) were not completely elided from the irreducible graph. As illustrated in Figure 5, the rest of the algorithm (steps 2 to 5) is designed to elide these remaining irrelevant branch nodes.

*Definition 11:* $G_{\text{CG}}$ is the **condensation graph** of a directed graph $G$ if each strongly-connected component (SCC) of $G$ contracts to a single node in $G_{\text{CG}}$ and the edges of $G_{\text{CG}}$ are induced by edges in $G$.

**Step 2: Non-Relevant Operation Condensation Graph** Compute the subgraph $G_{\text{I}}$ of $G_{\text{T-irr}}$ induced by its non-relevant operation nodes. Then, construct the non-relevant operation condensation graph $G_{\text{NRCG}}$ of $G_{\text{I}}$.

**Step 3: Relevant Operation Condensation Graph** Construct a new graph $G_{\text{RCG}}$ by adding the event nodes in $G_{\text{T-irr}}$ to $G_{\text{NRCG}}$. If an edge exists between an SCC and an event node $n$ in $G_{\text{T-irr}}$ then introduce an edge in $G_{\text{RCG}}$ between the contracted node for that SCC and the event node $n$.

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

**Step 5: Final PCG** Transform $G_{\text{cPCG}}$ into $G_{\text{PCG}}$ by expanding each *remaining* contracted SCC in $G_{\text{cPCG}}$ back to the original SCC as in $G_{\text{T-irr}}$. The resultant graph $G_{\text{PCG}}$ after this step is the PCG.

Figures 5($a$-$f$) illustrate CFG to PCG transformation. The relevant operation nodes are highlighted.

Note that a strongly-connected component (SCC) of branch nodes with two or more successors must be retained. Figure 6 shows the PCG of function `cancel_bulk_urbs` from the Linux kernel. The SCC consisting of the branch nodes $c_1$ and $c_2$ is retained as it has two successors, which are the terminal node $\perp$ and the operation node $e_1$.

The algorithmic complexity of the CFG to PCG transformation 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 et.al. [17] to compute strongly-connected components of a directed graph. This algorithm also has complexity $O(|V| + |E|)$, yielding the complexity of $O(|V|+|E|)$ for the CFG to PCG transformation. The run-time of the transformation does not depend on the number of paths in the CFG.

### A. PCG Theory with Proofs

We use the term **event** for relevant operations. We also use the notion of a colored graph $G$ where a subset of

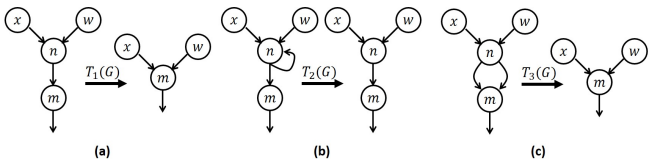


Figure 4. T-irreducible graph transformations: (a) $T_1$, (b) $T_2$, (c) $T_3$

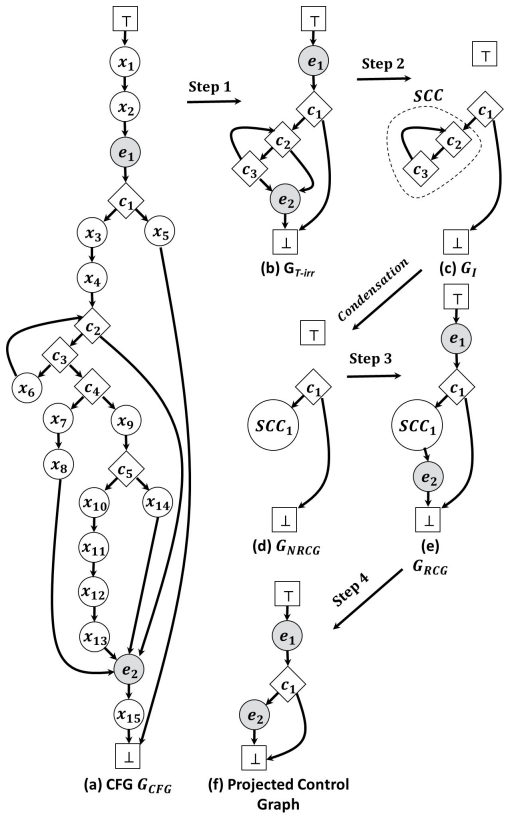


Figure 5. CFG to PCG Transformation Illustration

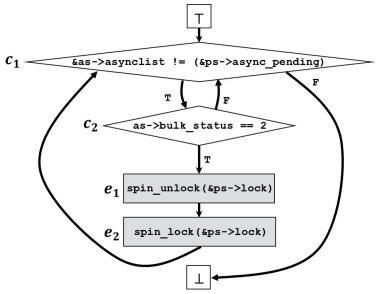


Figure 6. PCG for the function `cancel_bulk_urbs`

nodes are colored and each of those nodes has a unique color. The colored nodes represent relevant operations in our application. A relevant execution behavior can be thought of as a regular expression of colored nodes. A $T$**-irreducible** transform of $G$ is the graph $G^{'}$ that cannot be further reduced by applying the three transformations $T_1, T_2, T_3$ with respect to the colored nodes.

*Definition 12:* Given a CFG $G$ and a set $\mathcal{E}$ of relevant operation nodes in $G$, a **path equivalence relation** $\mathcal{R}_\mathcal{E}$ is defined on the set of paths in $G$: paths are related iff they have the same relevant execution behavior w.r.t. $\mathcal{E}$.

*Theorem 1:* Given a CFG $G$, a set $\mathcal{E}$ of relevant operations (*events*), and the equivalence relation $\mathcal{R}_\mathcal{E}$, there is a one-to-one and onto mapping between the path equivalence classes of $\mathcal{R}_\mathcal{E}$ and the paths of the PCG $\hat{G}$, and each PCG path produces the relevant execution behavior for a path equivalence class.

**Proof.** The equivalence relation $\mathcal{R}_\mathcal{E}$ partitions the CFG paths into equivalence classes such that all paths in an equivalence class have the same relevant execution behavior, and the CFG paths that are in different equivalence classes have different execution behaviors.

Since the PCG $\hat{G}$ is the node-induced subgraph of the given CFG $G$ consisting of the events and the relevant branch nodes, it follows that given a PCG path $P$, it corresponds to a unique relevant execution behavior $B$ and conversely given a relevant execution behavior $B$ there is a unique PCG path $P$ for which the relevant execution behavior is $B$. Thus, there is a one-to-one and onto mapping between the equivalence classes of $\mathcal{R}_\mathcal{E}$ and the paths of the PCG $\hat{G}$. □

*Theorem 2:* There exists a feasible CFG path $P$ with a relevant execution behavior $B$ if and only if there exists a path $\hat{P}$ with the behavior $B$ that is feasible with respect to the relevant branch nodes on $P$.

**Proof.** If every path with behavior $B$ is infeasible with respect to the relevant branch nodes, then all paths equivalent to $P$ are infeasible, because the addition of irrelevant branch nodes cannot make an infeasible path feasible. On the other hand, suppose there exists a CFG path $P'$ with behavior $B$ that is feasible with respect to the relevant branch nodes. By the definition of irrelevant branch nodes, an equivalence class has paths going through all possible branches at an irrelevant branch node. So, if the path $P'$ is not feasible due to some irrelevant branch nodes then we can choose feasible branches at those nodes to construct a new CFG path that is feasible and equivalent to $P'$. Thus, if there exists a path $P'$ with behavior $B$ that is feasible with respect to the relevant branch nodes on $P$, then there always exists a feasible CFG path with behavior $B$. □

*Definition 13:* The **boundary of subgraph** $S$ in a directed graph, denoted by *boundary(S)*, is the set of nodes $u \in S$ such that *suc(u)* $\in$ *suc(S)*.

Note: See Definition 7 and Definition 8 for the definitions of the successor of a node and the successor of a subgraph.

*Theorem 3:* Let $G$ be a colored acyclic graph. If $G$ is $T$-*irreducible* then for any subgraph $S$ of $G$ containing only non-colored nodes, $|suc(S)| \geq 2$.

**Proof.** If a non-colored node $u \in G$ has only one successor then it is eliminated by transformation $T_1$. Thus, since $G$ is $T$-*irreducible*, $|suc(u)| \geq 2$ for all non-colored nodes $u \in G$. Also, by assumption, $G$ is an acyclic graph. Using these two facts, we will show that the subgraph $S$ must have a node with at least two successors outside $S$ and thus $|suc(S)| \geq 2$.

Let $P_{v_0 \to v_n} : (v_0, v_1), (v_1, v_2), \cdots, (v_{n-1}, v_n))$ be a maximal path in subgraph $S$. Since $v_n$ is the terminal node of this maximal path $P$, its successor cannot be another node in $S$ *not* on the path $P$. Also, the successor of $v_n$ cannot be another node on the path $P$ because $G$ is an acyclic graph, so $v_n$ must belong to *boundary*($S$) and all its successors must be outside the subgraph $S$. Since $v_n$ is a non-colored node, $|suc(v_n)| \geq 2$. Since $v_n$, a node in $S$, has at least two successors outside of $S$, we have $|suc(S)| \geq 2$. This

completes the proof. □

*Corollary 1:* Let $G$ be a CFG and $G_{\text{cPCG}}$ be the condensed PCG. Then, for any subgraph $S$ containing non-colored nodes of $G_{\text{cPCG}}$, $|suc(S)| \geq 2$.

**Proof.** Note that the condensed PCG $G_{\text{cPCG}}$ is the graph resulting from step (4) of the transformation from the CFG to PCG. By construction, the condensed graph $G_{\text{cPCG}}$ is a colored $T$*-irreducible* graph. Also, by construction $G_{\text{cPCG}}$ is an acyclic graph. By applying the above theorem to $G_{\text{cPCG}}$ we get the proof of the corollary. □

*Corollary 2:* The PCG does not contain any irrelevant branch nodes.

**Proof.** Let $G$ be the CFG and let $G_{\text{T-irr}}$ be the irreducible graph obtained by applying transformations $T_1$, $T_2$, $T_3$ to $G$. We will prove that that all the irrelevant branch nodes will be eliminated when $G_{\text{cPCG}}$ is constructed. According to the definition, a node $c$ is irrelevant if there is a subgraph $S$ that contains $c$, all its branch edges, $S$ has no event nodes, and $|suc(S)| = 1$. It follows from this definition and from the corollary 1 that $G_{\text{cPCG}}$ does not contain any irrelevant branch nodes. Thus, the final graph PCG also does not contain any irrelevant branch nodes, because it consists of the nodes in $G_{\text{cPCG}}$ and all the event nodes. □

## VI. Assessment of Practical Impact

We present an empirical study to explore: *Is the analysis work significantly reduced in practice by using the optimal mathematical abstraction PCG instead of the CFG?* Since the analysis work is reflected by the size of the CFG and the PCG graphs, we use the size reduction from CFG to PCG as the metric to measure the work reduction. A compact PCG facilitates program comprehension. We did an experiment with a group of 40 students to audit results of a formal verification tool. Using the PCG graphs, the students found Linux bugs missed by the formal verification tool. We illustrate an example bug revealed using the PCG.

We built a tool that constructs the CFG and the PCG graphs and computes their size attributes. The tool is built using EnSoft's Atlas [18], [19], a platform to build software analysis, transformation, verification, and visualization tools. We created the CFG and the PCG graphs for each `Lock` instance from three versions (3.17-rc1, 3.18-rc1 and 3.19-rc1) of the Linux kernel. We enabled all possible `x86` build configurations via `allmodconfig` flag. The CFG and the PCG graphs for 66,609 `Lock` instances are posted on a website [11].

### A. Quantitative Assessment

Table I shows the distribution of nodes, edges, and branch nodes for both the CFGs and PCGs for all the relevant functions. Compared to 30914 CFGs, only 115 PCGs have more than 30 nodes, which is a reduction of 99%. Compared to 35145 CFGs, only 879 PCGs have more than 30 edges, which is a reduction of 97%. Compared to 17120 CFGs, only 1810 PCGs have more than 10 branch nodes, which is a reduction of 89%. Recall that the PCGs simplify path feasibility checks by reducing the number of branch nodes. Compared to 8644 CFGs, 30999 PCGs have no branch nodes, which is a 259% increase of cases where PCG eliminates the need for path feasibility check.

Table I
Linux kernel CFG to PCG Reduction for `Lock`/`Unlock` Pairing

| Graph | Artifact | Distribution | | | | |
|---|---|---|---|---|---|---|
| CFG | Nodes | $\leq 5$ | $6 \rightarrow 10$ | $11 \rightarrow 30$ | $31 \rightarrow 50$ | $> 50$ |
| | | 5,022 | 11,724 | 35,665 | 14,805 | 16,109 |
| | Edges | $\leq 5$ | $6 \rightarrow 10$ | $11 \rightarrow 30$ | $31 \rightarrow 50$ | $> 50$ |
| | | 6,670 | 10,025 | 31,485 | 15,002 | 20,143 |
| | Br-Nodes | $= 0$ | $1 \rightarrow 5$ | $6 \rightarrow 10$ | $11 \rightarrow 30$ | $> 30$ |
| | | 8,644 | 40,836 | 16,725 | 13,586 | 3,534 |
| PCG | Nodes | $\leq 5$ | $6 \rightarrow 10$ | $11 \rightarrow 30$ | $31 \rightarrow 50$ | $> 50$ |
| | | 66,820 | 12,515 | 3,875 | 109 | 6 |
| | Edges | $\leq 5$ | $6 \rightarrow 10$ | $11 \rightarrow 30$ | $31 \rightarrow 50$ | $> 50$ |
| | | 58,662 | 14,314 | 9,470 | 690 | 189 |
| | Br-Nodes | $= 0$ | $1 \rightarrow 5$ | $6 \rightarrow 10$ | $11 \rightarrow 30$ | $> 30$ |
| | | 30,999 | 45,282 | 5,234 | 1,756 | 54 |

The reduction from CFG to PCG is particularly important for a CFG with a large number of branch nodes. For example, for function `dst_ca_ioctl` the reductions from CFG to PCG are: from 349 nodes to 2 nodes, from 518 edges to only one edge, and from 163 branch nodes to zero branch nodes.

### B. Linux Bug Discovery with PCG

We conducted an empirical study where 36 undergraduate students from a software engineering course and 4 graduate students from our research group used the PCG and CFG graphs from the website [11] to audit results of `Lock` and `Unlock` pairing by the Linux Driver Verification (LDV) tool [12]. In all 400 instances of varying difficulty were chosen for audit. Each instance was independently audited by two undergraduate students and also by two graduate students. A Linux bug discovered through this study is described here.

Figure 7 shows the PCG for function `toshsd_thread_irq` that has calls to `Lock` and `Unlock`. The CFG for this function is more complex with 8 branch nodes and multiple loops. The multiple CFG paths between the `Lock` and the `Unlock` are all equivalent and they get mapped to one PCG path.

The PCG for `toshsd_thread_irq` shows a path on which the lock is not followed by an unlock. From the PCG, this path is feasible if its governing condition expression ($C_1\overline{C_2}$) is true. The feasibility check is easy to do manually and it shows that the path is feasible and thus it is a bug. This bug was reported to the Linux organization and it was fixed.

## VII. Related Work

The papers by Choi *et al.* [20] and Ramalingam [21] introduce the notion of *sparse evaluation graph* (SEG). Their research uses the same general principle that for a given analysis problem, a compact program graph can be constructed by removing irrelevant program statements. Apart from the common general principle, the PEC approach and

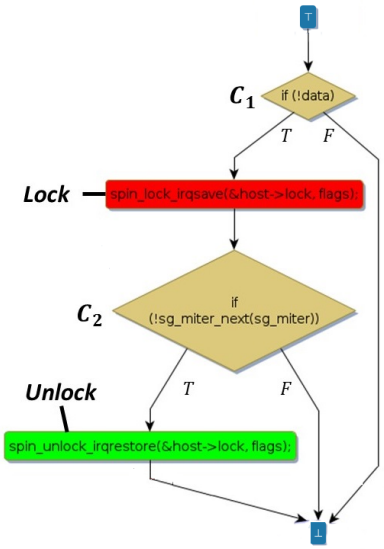


Figure 7. A Linux bug discovery using PCG

the PCG are completely different from the SEG approach. The fixed-point algorithm to compute the SEG aggregates data flow behaviors on different paths. The SEG is not meant for performing accurate path-sensitive analysis.

The Binary Decision Diagram (BDD) [22] has been used in different contexts of program analysis as a way to reduce the explosion of state space [8], [13]. The Binary Decision Tree (BDT) to BDD reduction has been also used for path-sensitive analysis [9]. Unlike BDT to BDD reduction, the PCG transformation does not require the input CFG to be acyclic.

CFG pruning techniques have been proposed in [23], [24] to overcome the computational complexity of exploring all paths. The PEC approach and PCG have evolved from our earlier research on *event view* [25] and *event flow graphs* [26].

## VIII. Conclusion

This paper presents an efficient and accurate approach to analyze software for a broad spectrum of safety and security vulnerabilities. It provides a rigorous formulation and a practical method to apply the approach. The approach is illustrated with a challenging analysis problem of pairing `Lock` and `Unlock` on all feasible execution paths. The tool support for the approach is developed using Atlas [18], [19], a platform for developing software analysis and visualization tools. The approach involves transforming CFG to PCG. The effectiveness of the approach depends on the reduction from CFG to PCG. The approach is evaluated using three versions of the Linux kernel. The CFGs and PCGs, for each of the $66,609$ `Lock` instances from the three versions of the Linux kernel, are posted on a website [11]. The paper presents an example of a Linux bug discovered using the approach.

## Acknowledgment

We thank our colleagues from Iowa State University and EnSoft for their help with this paper. Dr. Kothari is the founder President and a financial stakeholder in EnSoft.

## References

[1] "Coverity," http://www.coverity.com.

[2] "Klocwork K7," http://www.klocwork.com/.

[3] M. Das, S. Lerner, and M. Seigle, "ESP: Path-sensitive program verification in polynomial time," *ACM SIGPLAN Notices*, vol. 37, no. 5, pp. 57–68, 2002.

[4] L. Carter, J. Ferrante, and C. Thomborson, "Folklore confirmed: reducible flow graphs are exponentially larger," in *Proceedings of the 30 th ACM SIGPLAN-SIGACT*, 2003.

[5] M. Ngo and K. Tan, "Detecting large number of infeasible paths through recognizing their patterns." ACM, 2007.

[6] V. Vojdani and V. Vene, "Goblint: Path-sensitive data race analysis," in *Annales Univ. Sci. Budapest., Sect. Comp*, 2009.

[7] A. Navabi, N. Kidd, and S. Jagannathan, "Path-sensitive analysis using edge strings," *Purdue University Technical Report 10-006*, 2010.

[8] T. Ball and S. K. Rajamani, "Bebop: A path-sensitive interprocedural dataflow engine." ACM, 2001, pp. 97–103.

[9] Z. Xu and J. Zhang, "Path and context sensitive interprocedural memory leak detection," in *The Eighth International Conference on Quality Software*, 2008.

[10] I. Dillig, T. Dillig, and A. Aiken, "Sound, complete and scalable path-sensitive analysis," in *ACM SIGPLAN Notices*, vol. 43, no. 6. ACM, 2008, pp. 270–280.

[11] "Linux Results," http://kcsl.ece.iastate.edu/linux-results/.

[12] "Linux Driver Verification (LDV) Tool," http://linuxtesting.org/project/ldv.

[13] Y. Sui, S. Ye, J. Xue, and P.-C. Yew, "SPAS: scalable path-sensitive pointer analysis on full-sparse SSA," in *Programming Languages and Systems*. Springer, 2011, pp. 155–171.

[14] "Common Weakness Enumeration," http://cwe.mitre.org.

[15] C. Perrin, "The CIA triad," *http://www.techrepublic.com/blog/security/the-cia-triad/488*, 2008.

[16] B. Steensgaard, "Points-to Analysis in Almost Linear Time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '96. New York, NY, USA: ACM, 1996, pp. 32–41.

[17] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[18] T. Deering, S. Kothari, J. Sauceda, and J. Mathews, "Atlas: a new way to explore software, build analysis tools," in *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 588–591.

[19] "Atlas Platform, EnSoft Corp." http://www.ensoftcorp.com.

[20] J.-D. Choi, R. Cytron, and J. Ferrante, "Automatic construction of sparse data flow evaluation graphs," in *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1991, pp. 55–66.

[21] G. Ramalingam, *On sparse evaluation representations*. Springer, 1997.

[22] S. B. Akers, "Binary decision diagrams," *IEEE Trans. Computers*, vol. 27, no. 6, pp. 509–516, 1978.

[23] M. Ramanathan, A. Grama, and S. Jagannathan, "Path-sensitive inference of function precedence protocols," in *ICSE*, 2007.

[24] H. K. Cho, T. Kelly, Y. Wang, S. Lafortune, H. Liao, and S. Mahlke, "Practical lock/unlock pairing for concurrent programs," in *Code Generation and Optimization (CGO)*, 2013.

[25] S. Neginhal and S. Kothari, "Event views and graph reductions for understanding system level c code," in *ICSM*, 2006.

[26] A. Tamrawi and S. Kothari, "Event-Flow Graphs for Efficient Path-Sensitive Analyses," *arXiv preprint arXiv:1404.1279*, 2014.