Projected Control Graph for Computing Relevant Program Behaviors

Ahmed Tamrawi[a], Suresh Kothari[b]

[a]*EnSoft Corp., Ames, Iowa, USA 50010*
[b]*Department of Electrical and Computer Engineering, Iowa State University, Ames, Iowa, USA 50011*

Abstract

Many software engineering tasks require analysis and verification of all behaviors relevant to the task. For example, all relevant behaviors must be analyzed to verify a safety or security property. An efficient algorithm must compute the relevant behaviors directly without computing all the behaviors. This is crucial in practice because it is computationally intractable if one were to compute all behaviors to find the subset of relevant behaviors.

We present a mathematical foundation to define relevant behaviors and introduce the Projected Control Graph (PCG) as an abstraction to directly compute the relevant behaviors. We developed a PCG toolbox to facilitate the use of the PCG for program comprehension, analysis, and verification. The toolbox provides: (1) an interactive visual analysis mechanism, and (2) APIs to construct and use PCGs in automated analyses. The toolbox is designed to support multiple programming languages.

Using the toolbox APIs, we conducted a verification case study of the Linux kernel to assess the practical benefits of using the PCG. The study shows that the PCG-based verification is faster and can verify 99% of 66,609 instances compared to the 66% instances verified by the formal verification tool used by the Linux Driver Verification (LDV) organization. This study has revealed bugs missed by the formal verification tool. The second case study is an interactive use of the PCG Smart View to detect side-channel vulnerabilities in Java bytecode.

Keywords: Program Behaviors, Software Analysis, Software Verification, Software Security, Software Safety

---

1. Introduction

Accounting precisely for the execution behavior along each path of a Control Flow Graph (CFG) blows up the computational complexity: (1) the number of

Email addresses: `ahmedtamrawi@ensoftcorp.com` (Ahmed Tamrawi), `kothari@iastate.edu` (Suresh Kothari)

CFG paths grows exponentially with the number of non-nested branch nodes [1, 2], and (2) path feasibility checks can incur an exponential computation [3, 4, 5, 6, 7, 8]. Moreover, the number of paths blows up because of loop iterations.

In practice, the number of behaviors relevant to a task is often significantly smaller than the totality of behaviors. We present a mathematical foundation for computing *relevant program behaviors* as *relevant base behaviors* and *relevant iterative behaviors*. The goal is to compute the relevant behaviors directly without computing all possible behaviors. Based on the mathematical foundation, we introduce the Projected Control Graph (PCG) as an abstraction to directly compute the relevant behaviors.

Along with the mathematical foundation, we present insightful examples to illustrate possibilities of the drastic reduction in the number of behaviors from all behaviors to the relevant behaviors. Next, we summarize results of applying our PCG-based verification to the Linux kernel to verify the pairing of `Lock` instances with corresponding `Unlock` instances on all feasible control flow paths [9]. A control flow path is feasible if the path can be taken during an actual execution, i.e., variable values can be attained to satisfy the conditions governing the path. The study includes three versions of the Linux kernel which altogether have 37 million lines of code and $66,609$ `Lock` instances. We present a comparison with the formal verification tool that uses BLAST [10]. This BLAST tool has been a top performer in the annual software verification competition (SV-COMP) [11] and it is used by the Linux Driver Verification (LDV) organization [12]. The BLAST tool verifies $43,766$ (65.7%) of `Lock` instances as correctly paired (safe), and it is inconclusive (crashes or times out) on $22,843$ instances. The BLAST tool does not find any unsafe instances and requires 172 hours and 56 minutes for its verification. Our PCG-based automated verification tool verifies $66,151$(99.3%) of `Lock` instances as safe, and it is inconclusive on 451 instances. Seven unsafe instances are found through our study, including an instance that was incorrectly verified as safe by the BLAST tool. The PCG-based tool required 3 hours and 24 minutes.

Our second study is to use the PCG interactively to analyze Java bytecode to detect side-channel vulnerabilities. A compact PCG not only improves efficiency of an automated analysis, it also facilitates an interactive visual analysis and program comprehension. Using the Atlas Platform [13, 14], we have designed a visual analysis mechanism, called the *PCG Smart View*, to use the PCG interactively.

The key research contributions are:

- A mathematical foundation to define and compute relevant behaviors as *relevant base behaviors* and *relevant iterative behaviors*.

- The PCG as a graph abstraction to directly compute the relevant behaviors and an efficient algorithm to compute the PCG.

- An assessment of the practical impact of using the PCG interactively and programmatically for analyzing or verifying large software.

The remainder of the paper is organized as follows. We first describe the class of software safety and security problems to which the mathematical foundation for computing relevant behaviors applies in Section 2. Next, Section 3 describes the mathematical foundation for computing relevant behaviors. Section 4 presents the linear-time algorithm for constructing the PCG from its corresponding CFG. The developed PCG toolbox is presented in Section 5. Section 6 presents our Linux verification study that assesses the practical benefits of using PCGs in automated analysis and for interactive analysis. Section 7 discusses the use of PCGs in detecting side-channel vulnerabilities. Section 8 presents the related work. Finally, we conclude in Section 9.

2. Software Safety and Security Properties

This section describes a fairly broad class of software safety and security properties which can be verified efficiently using the PCG. In general, the PCG can be of significant value for program comprehension, analysis, and verification.

Definition 1 (**2-event matching**). *Verify that an event $e_1(O)$ is succeeded by an 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 verification 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 [15] can be viewed as 2-event problems.

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

Anti-matching covers software security verification defined according to *Confidentiality*, *Integrity*, and *Availability* (CIA) triad [16]. 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 to sensitive data* must *not* be followed by a *malicious modification to sensitive data* on any feasible execution path.

The following defines the general class of verification tasks for applying the PCG.

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.*

3. Mathematical Foundation for Computing Relevant Program Behaviors

We present a mathematical foundation to define relevant program behaviors. Using this foundation, we introduce the PCG as an efficient technique to compute relevant behaviors directly without computing all program behaviors.

Definition 4. *A **Control Flow Graph** (CFG) of a function 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 G.*

A CFG node is called a **branch node** if it has at least two outgoing *edges*, called the *branch edges*. We use the term *condition node* when we discuss a branch node and its associated condition. A *path* in CFG starts from $\top$ and ends with $\bot$. A path can iterate through a loop any number of times. We will use $[c_i]$ or $[\bar{c_i}]$ respectively to denote the `true` or `false` values taken for the condition expression C_i.

Definition 5 (**Relevant Statements**). *A subset of program statements that are determined to be relevant for a particular program analysis or verification task.*

3.1. An Overview of Computing Relevant Program Behaviors

Our approach is motivated by empirical studies of loops in open-source C and Java software and the need for a practical approach to account for loop behaviors relevant to verify safety and security properties. Each program behavior is a sequence of program statements. A sequence can have repetitions because of loop iterations. Each relevant program behavior is a sub-sequence that consists of only the relevant program statements and the relevant conditions.

Computing the relevant program behaviors involves: (a) computing the relevant program statements, (b) computing the relevant conditions to determine the feasibility of relevant behaviors, and (c) computing the relevant program behaviors. While this paper is primarily about (b) and (c), we have designed our PCG tool support to accommodate analyzers for part (a).

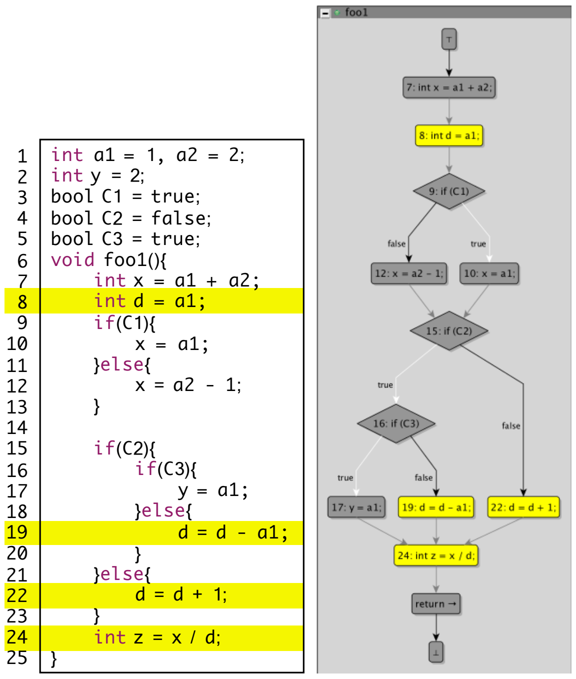
In Section 5, we will discuss our PCG tool support designed to work with a variety of analyzers for computing the relevant statements for a particular task. For example, the relevant statements for verifying the lock/unlock pairing requires a data flow analyzer that tracks the pointer to the lock object passed to the `Lock` instance and identifies the corresponding `Unlock` instances. The relevant statements include the data flow statements relevant for tracking the pointer.

The mathematical foundation for computing relevant behaviors introduces the notion of *base behaviors* and uses it as a basis for computing *iterative behaviors*. The base behaviors are defined using acyclic graphs. These acyclic graphs are obtained by removing the back edges from loops. The mathematical foundation progresses from acyclic graphs, loops without nesting, and finally to nested loops. The mathematical foundation requires a unique entry for each loop but it allows multiple loop termination nodes corresponding to `break` or `return`.

Given the relevant statements for a particular analysis or verification task, the PCG is an optimal graph abstraction that yields both: the relevant conditions and the relevant program behaviors for performing the task efficiently and accurately.

3.2. Relevant Base Behaviors for Acyclic CFGs

Let us start with an illustration. Consider the problem of verifying the function `foo1` (Figure 1(a)) for division-by-zero (DBZ) vulnerability on line 24 which involves division by d. The CFG for `foo1` is shown in Figure 1(b). The CFG is an acyclic graph with six paths. Each path yields a unique base behavior. The base behaviors are as listed in Table 1. The behaviors (B_1 and B_2) exhibit the DBZ vulnerability. The yellow highlighted statements shown in Figure 1 are the *relevant program statements*. In this example, these statements are relevant to the DBZ vulnerability because they affect the value of the denominator d in line 24.



```
 1  int a1 = 1, a2 = 2;
 2  int y = 2;
 3  bool C1 = true;
 4  bool C2 = false;
 5  bool C3 = true;
 6  void foo1(){
 7       int x = a1 + a2;
 8       int d = a1;
 9       if(C1){
10            x = a1;
11       }else{
12            x = a2 - 1;
13       }
14
15       if(C2){
16            if(C3){
17                 y = a1;
18            }else{
19                 d = d - a1;
20            }
21       }else{
22            d = d + 1;
23       }
24       int z = x / d;
25  }
```

(a) foo1 source code (b) foo1 CFG

Figure 1: A division-by-zero (DBZ) vulnerability

Verifying any vulnerability poses challenges: (1) computing the vulnerable behaviors out of all behaviors, and (2) path feasibility, i.e., checking the feasi-

Table 1: Base behaviors and relevant base behaviors for `foo1` (Figure 1(a))

Base Behaviors	**Relevant Base Behaviors**
$B_1 : 7, 8, 9[c_1], 10, 15[c_2], 16[\bar{c_3}], 19, 24$	$RB_1 : 8, 15[c_2], 16[\bar{c_3}], 19, 24$
$B_2 : 7, 8, 9[\bar{c_1}], 12, 15[c_2], 16[\bar{c_3}], 19, 24$	
$B_3 : 7, 8, 9[c_1], 10, 15[\bar{c_2}], 22, 24$	$RB_2 : 8, 15[\bar{c_2}], 22, 24$
$B_4 : 7, 8, 9[\bar{c_1}], 12, 15[\bar{c_2}], 22, 24$	
$B_5 : 7, 8, 9[c_1], 10, 15[c_2], 16[c_3], 17, 24$	$RB_3 : 8, 15[c_2], 16[c_3], 24$
$B_6 : 7, 8, 9[\bar{c_1}], 12, 15[c_2], 16[c_3], 17, 24$	

bility of CF paths with vulnerable behaviors. The large number of behaviors (i.e., 2^n behaviors for n non-nested branch nodes) is one reason the problem of computing all behaviors becomes computationally intractable. Computing the path feasibility is intractable because it is equivalent to the satisfiability problem [17]. Later, we shall see a third challenge due to iterations of loops.

A closer inspection reveals that multiple base behaviors can be grouped so that each group corresponds to a unique *relevant base behavior*. The relevant statements for the DBZ vulnerability on line 24 (Figure 1(a)) are: $8, 19, 22$ and 24. The relevant base behaviors and the corresponding groups of base behaviors are listed in Table 1. Of the three relevant base behaviors, RB_1 exhibits the DBZ vulnerability. Conditions $C2$ and $C3$ are included in relevant base behaviors as *relevant conditions*. These conditions determine the feasibility of relevant base behaviors, in particular the vulnerable relevant behavior is RB_1.

Let us finish this subsection with formal definitions for *base behaviors* and *relevant base behaviors*. The formal definition for *relevant conditions* is given later (Definition 17). For now, a condition is relevant if there is a relevant statement that is control dependent on that condition.

Definition 6. *For a path P in an acyclic CFG, its* ***base behavior*** *B_P is the sequence of program statements and conditions along P.*

Definition 7. *The* ***relevant base behaviors*** *for an acyclic CFG are the distinct sub-sequences derived from base behaviors by retaining only the relevant statements and the relevant conditions.*

In the above example, three distinct sub-sequences RB_1, RB_2, RB_3 are derived from 6 behaviors B_1 to B_6 by retaining only the relevant statements and relevant conditions.

3.3. Relevant Iterative Behaviors for CFGs with Loops

Definition 8. ***Successors of a node*** *u in a directed graph G, denoted by* $\mathrm{suc}(u)$*, consist of the set of nodes $v \neq u$ such that $\exists$ an edge (u, v).*

Definition 9. ***Node*** *d* ***dominates node*** *n if every path from $\top$ to n goes through d.*

Definition 10. *A* ***back edge*** *is an edge whose head dominates its tail.*

Definition 11. *A* ***loop*** *$\mathcal{L}$ is a subgraph with the following properties:*

1. *$\mathcal{L}$ has a unique node h, called the loop header, to enter the subgraph $\mathcal{L}$. The header dominates all nodes of $\mathcal{L}$.*
2. *$\mathcal{L}$ contains at least one **back edge** (n, h) where $n \in \mathcal{L}$.*
3. *$\mathcal{L}$ has one or more **termination nodes**, defined as the nodes that have a successor that is outside $\mathcal{L}$. Termination nodes which account for `break` or `return` are referred to as exceptional termination nodes. Other termination nodes are referred to as normal termination nodes. A loop header in `while` and `for` loops is a normal termination node. In case of `do-while` loops, the normal termination node is the condition node associated with the boolean expression for `while`.*

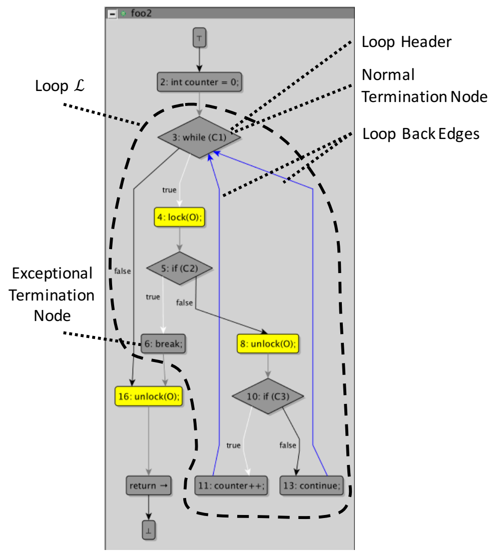
3.3.1. Relevant Iterative Behaviors for Non-nested Loops

Consider the problem of verifying lock/unlock pairing: a `Lock` must be followed by a corresponding `Unlock` and it must not be followed by another `Lock`. Let us show how to compute the relevant iterative behaviors to verify the property.

Figure 2 shows the code for the function `foo2` and its CFG. `foo2` has a loop $\mathcal{L}$, which is the subgraph of nodes corresponding to lines 3 to 15 with the loop header at line 3, normal termination node at line 3, exceptional termination node at line 6, and the blue colored edges (11, 3) and (13, 3) are the back edges. Figure 3 shows the acyclic graph for the loop $\mathcal{L}$ obtained by removing the two back edges.

```
1   void foo2(bool C1, bool C2, bool C3){
2       int counter = 0;
3       while(C1){
4           lock(O);
5           if(C2){
6               break;
7           }else{
8               unlock(O);
9           }
10          if(C3){
11              counter++;
12          }else{
13              continue;
14          }
15      }
16      unlock(O);
17  }
```

(a) foo2 source code



(a) foo2 CFG

Figure 2: Loops with `break` and `continue`

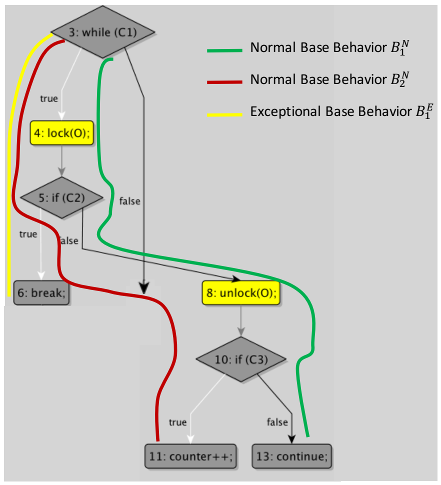


Figure 3: The acyclic graph for loop $\mathcal{L}$ in `foo2` (Figure 2)

Definition 12. *The **acyclic graph of a loop** $\mathcal{L}$ is the graph $A(\mathcal{L})$ obtained by removing the back edges from $\mathcal{L}$. The **acyclic graph of a CFG** G is the graph $A(G)$ obtained by removing all the back edges from loops in G.*

Definition 13. ***Base behaviors** of a loop or a cyclic CFG are the base behaviors defined by the acyclic graphs obtained from the loop or the CFG by removing the back edges.*

The base behaviors for loops are partitioned into: (a) *normal base behaviors* (B^N) - the behaviors along the paths that terminate at the normal termination point, and (b) *exceptional base behaviors* (B^E) - the behaviors along the paths that terminate at the exceptional termination points.

Definition 14. *An **iterative behavior** is a sequence of base behaviors of length k, where k is a positive integer representing the number of iterations of a loop.*

Definition 15. *An **iterative relevant behavior** is a sequence of relevant base behaviors of length k, where k is a positive integer representing the number of iterations of a loop.*

For $k > 0$ iterations, p normal base behaviors, e exceptional base behaviors, the number of iterative behaviors is: $(p^k + e \times p^{k-1})$ with (p^k) iterative behaviors that do not end with an exceptional base behavior, and $(e \times p^{k-1})$ iterative behaviors with $k-1$ iterations of normal base behaviors followed by a final iteration of an exceptional base behavior. Note that the maximum number of normal base behaviors for an acyclic graph of a loop is $p = 2^n$, where n is number of branch nodes along these behaviors. In the acyclic graph for loop $\mathcal{L}$ in function `foo2` (Figure 3), there are two normal base behaviors B_1^N and B_2^N and one exceptional base behavior B_1^E.

In the function `foo2`, the relevant statements are: $4, 8$ and 16. The relevant conditions are: C_1 and C_2 with corresponding statements 3 and 5 respectively. The base behaviors B_1^N and B_2^N are grouped because they represent the same relevant normal base behavior which is: $\boxed{3[c_1], 4, 5[\bar{c_2}], 8}$. The relevant exceptional base behavior corresponding to B_1^E is: $\boxed{3[c_1], 4, 5[c_2]}$. We use the notation $(v_1, v_2, \cdots)^k$ to denote that the sequence of statements $(v_1, v_2, \cdots)$ is repeated k times. With that notation: the relevant behaviors of the loop $\mathcal{L}$ for $k > 0$ iterations are: $\boxed{(3[c_1], 4, 5[\bar{c_2}], 8)^k}$ and $\boxed{(3[c_1], 4, 5[\bar{c_2}], 8)^{k-1}, 3[c_1], 4, 5[c_2]}$.

Substituting the relevant behaviors of the loop, we get for the entire function `foo2` the following relevant behaviors to verify the lock/unlock pairing: $\boxed{(3[c_1], 4, 5[\bar{c_2}], 8)^k, 3[\bar{c_1}], 16}$ and $\boxed{(3[c_1], 4, 5[\bar{c_2}], 8)^{k-1}\ ,\ 3[c_1], 4, 5[c_2], 16}$, where $k > 0$ is the number of loop iterations. We have one more behavior $\boxed{3[\bar{c_1}], 16}$ for the case that the loop is not entered ($k = 0$). One can verify that all three relevant base behaviors are safe.

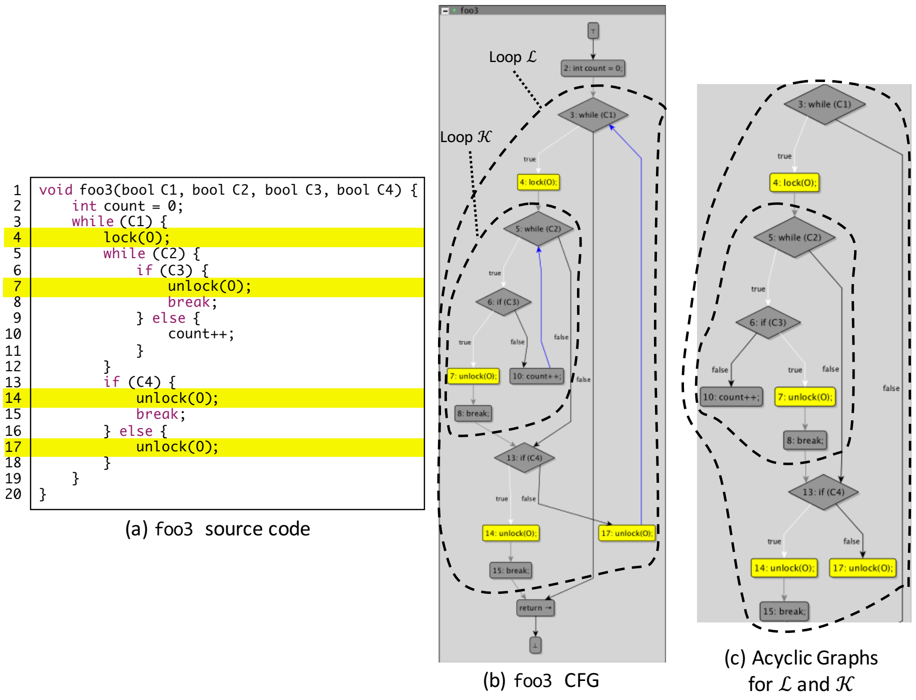
3.3.2. Relevant Iterative Behaviors for Nested Loops

As before, the iterative behaviors are defined using the base behaviors. However, the base behaviors for nested loops are defined recursively as follows.

Remark 1. *The* ***base behaviors in the presence of nested loops*** *are defined recursively by substituting the iterative behaviors of each nested loop on a path.*

Let us illustrate this process by computing the relevant behaviors to verify lock/unlock pairing for the function `foo3` shown in Figure 4. The code and the CFG are shown in the Figure 4. Line 3 (Figure 4(a)) represents the loop header for the outer loop $\mathcal{L}$ and Line 5 represents the loop header for the inner loop $\mathcal{K}$. $\mathcal{L}$ has one normal termination node at line 3 and one exceptional termination node at line 15. $\mathcal{K}$ has one normal termination node at line 5 and one exceptional termination node at line 8. The loop back edge $(17, 3)$ is associated with $\mathcal{L}$ and and the loop back edge $(10, 5)$ is associated with $\mathcal{K}$. The acyclic graph for the two loops is shown in Figure 4(c).

The relevant statements for verifying the lock/unlock pairing are: $4, 7, 14$ and 17. In this example, all conditions C_1, C_2, C_3, and C_4 are relevant conditions. The relevant behaviors for loop $\mathcal{K}$ with $j > 0$ iterations are: $\boxed{(5[c_2], 6[\bar{c_3}])^j}$ and $\boxed{(5[c_2], 6[\bar{c_3}])^{j-1}, 5[c_2], 6[c_3], 7}$. There is one more behavior if $\mathcal{K}$ is not entered ($j = 0$) which is $\boxed{5[\bar{c_2}]}$. The relevant behaviors for loop $\mathcal{L}$ with $k > 0$ iterations are: $\boxed{(3[c_1], 4, \Delta, 13[\bar{c_4}], 17)^k}$ and $\boxed{(3[c_1], 4, \Delta, 13[\bar{c_4}], 17)^{k-1},\ 3[c_1], 4, \Delta, 13[c_4], 14}$ where Δ denotes an iterative relevant behavior of the inner loop $\mathcal{K}$. There is one more behavior for the case that the loop $\mathcal{L}$ is not entered ($k = 0$) which is $\boxed{3[\bar{c_1}]}$.



```
1  void foo3(bool C1, bool C2, bool C3, bool C4) {
2      int count = 0;
3      while (C1) {
4          lock(O);
5          while (C2) {
6              if (C3) {
7                  unlock(O);
8                  break;
9              } else {
10                 count++;
11             }
12         }
13         if (C4) {
14             unlock(O);
15             break;
16         } else {
17             unlock(O);
18         }
19     }
20 }
```

(a) foo3 source code

(b) foo3 CFG

(c) Acyclic Graphs for $\mathcal{L}$ and $\mathcal{K}$

Figure 4: An illustrative example using two nested loops

3.4. Behavior Compaction

Since the number of iterations of a nested loop and its behaviors could vary across iterations of the outer loop, computing the base behaviors for the outer loop may become extremely computation intensive. We introduce the notion of *behavior compaction* to mitigate this computational complexity. The compaction rules vary based on the verification requirement. We illustrate the compaction for the loop $\mathcal{K}$ nested in the loop $\mathcal{L}$ shown in Figure 4.

The compaction rules for the lock/unlock pairing, where `L` and `U` denote `Lock` and `Unlock` operations respectively and ϕ denotes a behavior that does not contain any relevant statement and thus it is a *null sequence*, are: (i) $(\texttt{U})^j = \texttt{U}$, (ii) $(\phi, \texttt{L}) = \texttt{L}$, (iii) $(\phi, \texttt{U}) = \texttt{U}$, and (iv) $(\texttt{L}, \texttt{U})^j = (\texttt{L}, \texttt{U})$. We will use the abbreviations RNBB and REBB to denote the relevant normal base behavior and relevant exceptional base behavior, respectively. Applying the compaction rules, the relevant base behaviors for loop $\mathcal{K}$ with j iterations are:

- The RNBB $\boxed{(5[c_2], 6[\bar{c_3}])^j}$ can be compacted as: ϕ.
- The REBB $\boxed{(5[c_2], 6[\bar{c_3}])^{j-1}, 5[c_2], 6[c_3], 7}$ can be compacted as: `U`.

- The relevant behavior when the loop is not iterated $\boxed{5[\bar{c_2}]}$ can be compacted as: ϕ.

After the behavior compaction, there are two unique relevant behaviors for the loop $\mathcal{K}$: ϕ and $\texttt{U}$. Similarly, the relevant base behaviors for the loop $\mathcal{L}$ are:

- The RNBB $\boxed{(3[c_1], 4, \Delta, 13[\bar{c_4}], 17)^k}$ can be compacted as: $(\texttt{L}, \Delta, \texttt{U})^k$.
- The REBB $\boxed{(3[c_1], 4, \Delta, 13[\bar{c_4}], 17)^{k-1}, 3[c_1], 4, \Delta, 13[c_4], 14}$ can be compacted as: $(\texttt{L}, \Delta, \texttt{U})^{k-1}, \texttt{L}, \Delta, \texttt{U}$.
- The relevant behavior when the loop is not entered $\boxed{3[\bar{c_1}]}$ can be compacted as: ϕ.

Incorporating the compacted relevant behaviors of $\mathcal{K}$ into the relevant base behaviors of $\mathcal{L}$, the compacted relevant base behaviors for $\mathcal{L}$ are:

- Relevant normal base behaviors:
  1. $(\texttt{L}, \phi, \texttt{U})^k$. Compacted as: $\texttt{L}, \texttt{U}$.
  2. $(\texttt{L}, \texttt{U}, \texttt{U})^k$. Compacted as: $\texttt{L}, \texttt{U}$.
- Relevant exceptional base behaviors:
  1. $(\texttt{L}, \phi, \texttt{U})^{k-1}, \texttt{L}, \phi, \texttt{U}$. Compacted as: $\texttt{L}, \texttt{U}$.
  2. $(\texttt{L}, \phi, \texttt{U})^{k-1}, \texttt{L}, \texttt{U}, \texttt{U}$. Compacted as: $\texttt{L}, \texttt{U}$.
  3. $(\texttt{L}, \texttt{U}, \texttt{U})^{k-1}, \texttt{L}, \phi, \texttt{U}$. Compacted as: $\texttt{L}, \texttt{U}$.
  4. $(\texttt{L}, \texttt{U}, \texttt{U})^{k-1}, \texttt{L}, \texttt{U}, \texttt{U}$. Compacted as: $\texttt{L}, \texttt{U}$.
- The relevant behavior when the loop is not entered: ϕ.

All relevant behaviors are compacted as $(\texttt{L}, \texttt{U})$ and ϕ, which implies correct pairing of `Lock` and `Unlock` on all paths of `foo3`.

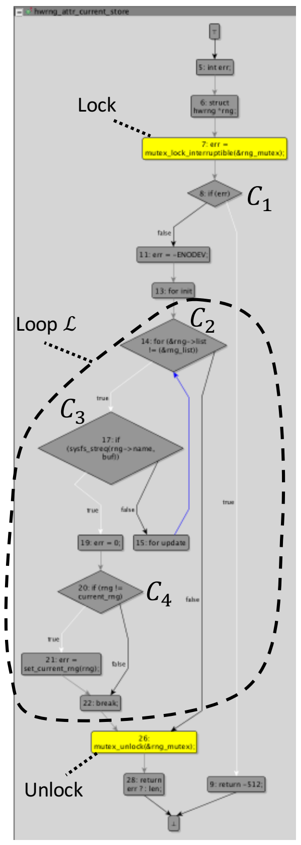
3.5. Linux Example

This lock/unlock pairing example is from the Linux kernel. Figure 5 shows the CFG of `hwrng_attr_current_store` function. The relevant statements in this example are: 7 corresponding to `L` and 26 corresponding to `U`. There is one relevant condition which is C_1. Following the previous discussion for computing relevant behaviors, there is one normal base behavior for loop $\mathcal{L}$ and two exceptional base behaviors. These behaviors do not contain relevant statements and thus yield ϕ. The overall relevant behaviors in this example are: (1) $\boxed{\texttt{L}, 8[c_1]}$ and (2) $\boxed{\texttt{L}, 8[\bar{c_1}], \texttt{U}}$. The first behavior needs the path feasibility check because it is a potential vulnerability.

Later in Section 4.2, we will show the PCGs for the examples presented in this section and how the PCG directly computes the relevant behaviors without the need to compute all behaviors.

```
static DEFINE_MUTEX(rng_mutex);

static ssize_t hwrng_attr_current_store(struct
device *dev, struct device_attribute *attr,
const char *buf, size_t len){

    int err;
    struct hwrng *rng;
    err = mutex_lock_interruptible(&rng_mutex);
    if (err)
        return -ERESTARTSYS;

    err = -ENODEV;

    for (rng = list_first_entry(&rng_list,
                                   typeof(*rng), list);
            &rng->list != (&rng_list);
            rng = list_next_entry(rng, list))
    {
        if (sysfs_streq(rng->name, buf))
        {
            err = 0;
            if (rng != current_rng)
                err = set_current_rng(rng);
            break;
        }
    }

    mutex_unlock(&rng_mutex);

    return err ? : len;
}
```

(a) hwrng_attr_current_store source code



(b) hwrng_attr_current_store CFG

Figure 5: Code and CFG for the function **hwrng_attr_current_store**

4. The Projected Control Graph (PCG)

The mathematical foundation uses relevant base behaviors to compute iterative behaviors. The PCG is a graph abstraction to compute the relevant base behaviors efficiently. We shall present an efficient algorithm to obtain the PCG by transforming a CFG.

Definition 16. ***Successors of a subgraph*** *S in a directed graph G, denoted by $\mathrm{suc}(S)$, consist of the set of nodes $v \notin S$ such that $v \in \mathrm{suc}(u)$ for $u \in S$.*

Definition 17. *A condition node C is an* ***irrelevant condition node*** *if there exists a subgraph S containing C and all branch edges of C such that S does not have nodes corresponding to relevant statements, and S has a unique successor, i.e., $|\mathrm{suc}(S)| = 1$. If such a set S does not exist for a condition node then it is a* ***relevant condition node****.*

Definition 18. ***Predecessors of a node*** *u in a directed graph G, denoted by* $\mathrm{pred}(u)$*, consist of the set of nodes $v \neq u$ such that $\exists$ an edge (v, u).*

Definition 19 (**Projected Control Graph (PCG)**)**.** *The PCG of a CFG $G = (V, E, \top, \bot)$ is $PCG = (V', E', \top, \bot)$, where V' consists of the nodes in G for relevant statements and relevant conditions. E' are the resultant set of edges connecting only V' after the removal of all irrelevant statement nodes in V and retaining all relevant condition nodes in V. For each removed irrelevant node $r \in V - V'$ or a subgraph of irrelevant nodes $R \subseteq V - V'$, a new set of edges are introduced so that pred(r)/pred(R) become predecessors of suc(r)/suc(R), then all incoming and outcoming edges of r or each node in R are removed. $\top$ and $\bot$ denote the respective unique entry and exit nodes of the PCG.*

4.1. CFG to PCG Transformation

We present an efficient algorithm to compute the PCG. It uses Tarjan's algorithm to compute strongly-connected components of a directed graph [18]. We will use the terms relevant and irrelevant node to denote a node in a graph corresponding to a relevant and irrelevant statement for a particular analysis.

Step 1: *T-Irreducible Graph $G_{T\text{-}irr}$ Construction*

Reduce the CFG G_{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 Irrelevant Nodes*

Let n be an *irrelevant* node with a single successor m. The T_1 transformation is the consumption of node n by m. New edges are introduced so that predecessors of n become predecessors of m. (Figure 6(a))

T_2: *Elide Self-Loop Edges*

Let n be an *irrelevant* node that has a self-loop edge (n, n). The T_2 transformation removes that edge (Figure 6(b)). When a loop block does not contain relevant nodes, execution of the loop is immaterial as it yields a ϕ behavior. T_2 transformation elides such loops.

T_3: *Elide Simple Irrelevant Condition Nodes*

Let n be a *condition* node that does not correspond to a relevant node and without relevant nodes on its branch edges 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 of m (Figure 6(c)). T_3 elides only a subset of the *irrelevant condition nodes* (Definition 17). The condition nodes elided by T_3 are the ones that become vacuous after T_1 elides all irrelevant nodes.

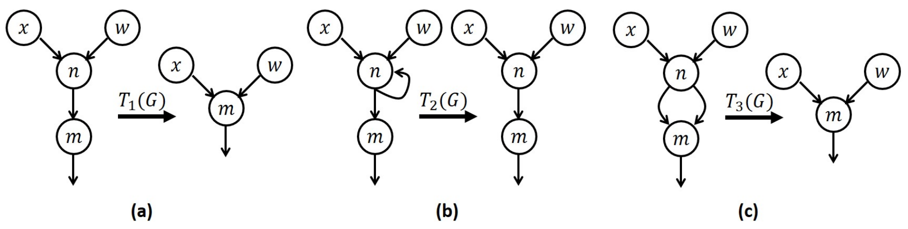


Figure 6: T-irreducible graph transformations: (a)T_1, (b)T_2, (c)T_3

Listing 1 shows the worklist algorithm for creating a $G_{\text{T-irr}}$ from an input graph G.

```
Procedure constructTIrreducibleGraph($G(V, E, \top, \bot)$)
    do
        // Transformation $T_1$ to elide irrelevant nodes.
        for each irrelevant node $n \in G : \mathtt{suc}(n) = \{m\}$
            for each edge $(p, n)$ where $p \in \mathtt{pred}(n)$
                $G \leftarrow G(V, E + (p, m) - (p, n), \top, \bot)$
            end for
            $G \leftarrow G(V - n, E, \top, \bot)$
        end for

        // Transformation $T_2$ to elide self-loop edges.
        for each irrelevant node $n \in G : n$ has a self-loop edge
            $G \leftarrow G(V, E - (n, n), \top, \bot)$
        end for

        // Transformation $T_3$ to elide irrelevant condition nodes.
        for each condition node $n \in G : \mathtt{suc}(n) = \{m\}$ and $n$ is
            irrelevant statement
            for each edge $(p, n)$ where $p \in \mathtt{pred}(n)$
                $G \leftarrow G(V, E + (p, m) - (p, n), \top, \bot)$
            end for
            $G \leftarrow G(V - n, E, \top, \bot)$
        end for
    while (there is a change in $G$)
    return $G$
end Procedure
```

Listing 1: The worklist algorithm for creating a $G_{\text{T-irr}}$ from an input graph G

We examined the irreducible graphs obtained by applying the three transformations in Step 1 and found examples of complex CFGs where some of the *irrelevant condition nodes* (Definition 17) were not completely elided from the irreducible graph. The rest of the algorithm (Steps 2 to 5) is designed to elide these remaining irrelevant condition nodes.

Definition 20. *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.*

Step 2: *Irrelevant Nodes Condensation Graph G_{IRCG} Construction*

Compute the subgraph G_{I} of $G_{\text{T-irr}}$ induced by its irrelevant nodes. Then, construct the irrelevant node condensation graph G_{IRCG} of G_{I}.

Step 3: *Relevant Nodes Condensation Graph G_{RCG} Construction*

Construct a new graph G_{RCG} by adding the relevant nodes in $G_{\text{T-irr}}$ to G_{IRCG}. If an edge exists between a node in an SCC and a relevant node n in $G_{\text{T-irr}}$, then introduce an edge in G_{RCG} between the contracted node for that SCC and the relevant node n.

Step 4: *Condensed PCG G_{cPCG} Construction*

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

Step 5: *Final PCG G_{PCG} Construction*

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

Figures 7(*a*-*g*) illustrate CFG to PCG transformation. In this example, the relevant nodes (highlighted in yellow) are: r_1 and r_2. Note that the G_{cPCG} in Figure 7(*f*) and the *PCG* in Figure 7(*g*) are the same as there are no remaining SCCs in G_{cPCG} that need to be expanded in *PCG*.

4.2. PCG Examples

Figure 8(a) shows the PCG for function `foo1` (Figure 1) that we used to verify the DBZ vulnerability. The relevant statements are highlighted in yellow. We can see that there are three paths in the PCG and each path provides a unique relevant behavior. Note that the condition node C_1 is not present in the PCG as it is an irrelevant condition (Definition 17) causing multiple CFGs paths to exhibit the same relevant behavior, i.e., B_1 and B_2 behaviors result in the same relevant behavior RB_1 in Table 1. Therefore, checking whether function `foo1` is vulnerable requires checking the feasibility of the conditions C_2 and C_3 as both are relevant conditions.

Figure 8(b) shows the PCG for function `foo2` (Figure 2) that we use to verify correct lock/unlock pairing. By removing the back edges from the PCG, it results in three paths. Each path contributes a unique relevant base behavior. The CFG has two normal base behaviors due to the C_3 condition, but both normal base behaviors exhibit the same relevant base behavior. Thus, the PCG transformation eliminates the C_3 condition as it is an irrelevant condition.

The same reasoning applies to the PCG in Figure 8(c) for function `foo3` with nested loops (Figure 4). We can see that each path in the acyclic PCG exhibits a unique relevant base behavior. With behavior compaction, we can see that all relevant base behaviors are safe. The PCG retains all condition nodes in this example as they are relevant conditions. Note that the SCC of the condition

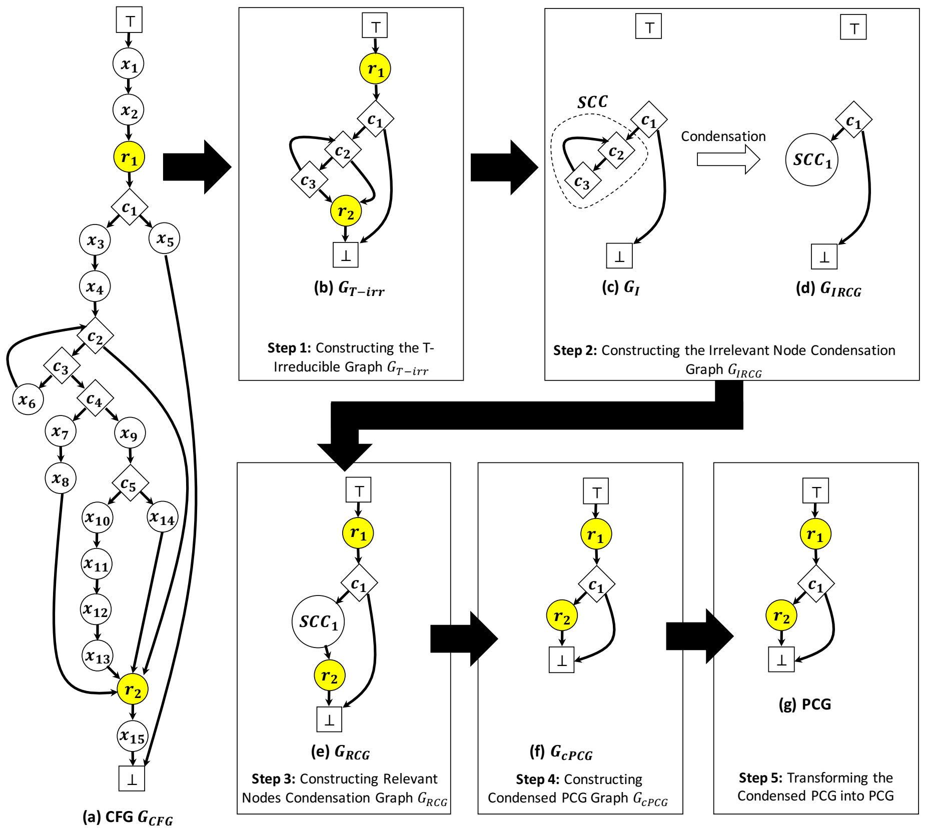

(a) CFG G_{CFG}
(b) G_{T-irr}
Step 1: Constructing the T-Irreducible Graph G_{T-irr}
SCC
Condensation
(c) G_I
(d) G_{IRCG}
Step 2: Constructing the Irrelevant Node Condensation Graph G_{IRCG}
(e) G_{RCG}
Step 3: Constructing Relevant Nodes Condensation Graph G_{RCG}
(f) G_{cPCG}
Step 4: Constructing Condensed PCG Graph G_{cPCG}
(g) PCG
Step 5: Transforming the Condensed PCG into PCG

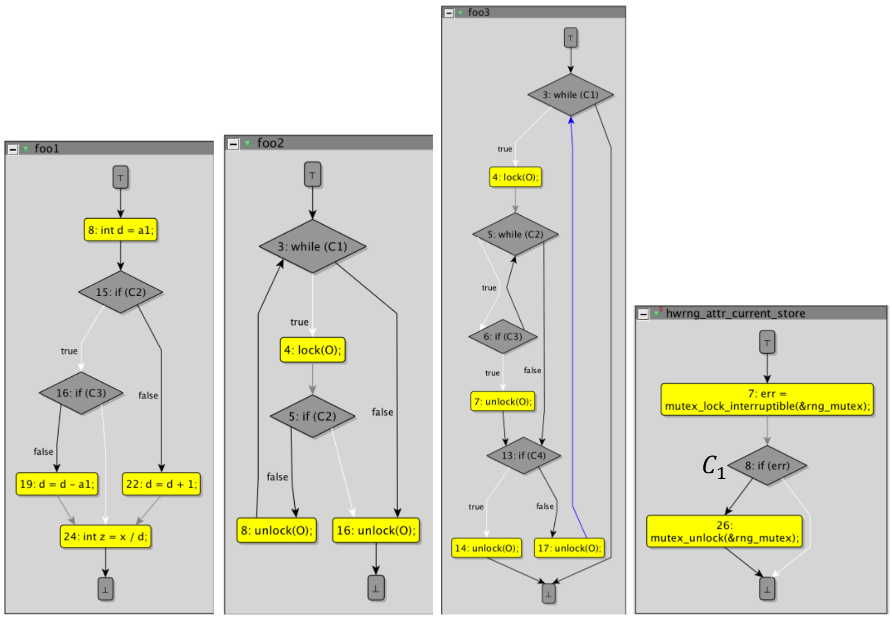
Figure 7: CFG to PCG Transformation Illustration

nodes C_2 and C_3 is retained in the PCG as it has two relevant successors, which are the relevant condition node C_4 and the relevant node `unlock(0)` (Line 7).

Finally, Figure 8(d) shows the PCG for function `hwrng_attr_current_store` (Figure 5). The PCG has two paths and each path corresponds to a unique relevant behavior. Note that C_1 is retained in the final PCG as it is needed to check the feasibility of the vulnerable path from the `true` branch of condition C_1. The condition nodes C_2, C_3 and C_4 in the CFG (Figure 5(b)) are removed in the PCG as they are irrelevant condition nodes.

4.3. Algorithmic Complexity

Let $|V|$ and $|E|$ be the respective numbers of nodes and edges in the CFG. For creating the T-irreducible graphs in Step 1 and Step 4, we use the worklist algorithm in Listing 1. This algorithm never grows in the size of the input graph and the maximum size of the graph it will traverse is $|V| + |E|$, yielding the complexity of $O(|V| + |E|)$. For detecting the SCCs in Step (2), we use an algorithm by Tarjan *et al.* [18] to compute strongly-connected components



(a) `foo1` PCG (b) `foo2` PCG (c) `foo3` PCG (d) `hwrng_attr_current_store` PCG

Figure 8: PCGs for the examples in Section 3

of a directed graph. This algorithm also has a complexity of $O(|V| + |E|)$, 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.

4.4. PCG Theory with Proofs

In this section, we show the *optimality of the PCG*, which is to minimize the repetitive computation encountered in computing the relevant base behaviors from the CFG. Given a CFG G, computing the relevant base behaviors for every path of $A(G)$ (Definition 12) involves unnecessary repetitions because multiple paths in $A(G)$ can produce the same relevant base behavior.

Definition 21. *The **optimality property** of PCG $\hat{G}$ is defined as: (1) each path in $A(\hat{G})$ (Definition 12) yields a distinct relevant base behavior, (2) $A(\hat{G})$ includes the totality of relevant base behaviors of $A(G)$ where G is the CFG, and (3) $\hat{G}$ does not contain irrelevant statements.*

For establishing the optimality of PCG, we use the notion of a colored graph G where a subset of nodes are colored and each of those nodes has a unique color. The colored nodes represent relevant statements for a particular analysis.

Definition 22. *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) *and* suc(S) *have a non-empty intersection.*

Note: See Definition 8 and Definition 16 for the definitions of the successor of a node and the successor of a subgraph.

Theorem 1. *Let G be a colored graph such that any subgraph S of G containing only non-colored nodes is an acyclic graph. If G is T-irreducible then* $|\text{suc}(S)| \geq 2$.

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*, $|\textit{suc(u)}| \geq 2$ for all non-colored nodes $u \in G$. Also, by assumption, any subgraph S of G containing only non-colored nodes 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 $|\textit{suc(S)}| \geq 2$.

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 S 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, we have $|\textit{suc}(v_n)| \geq 2$. Since $v_n \in S$ and has at least two successors outside of S, we have $|\textit{suc(S)}| \geq 2$. This completes the proof. □

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

Proof. Note that the condensed PCG G_{cPCG} is the graph resulting from Step (4) of the transformation from the CFG to PCG. By construction, the condensed graph G_{cPCG} is a colored *T-irreducible* graph. Also, by construction any subgraph S of G_{cPCG} containing only non-colored nodes is an acyclic graph. By applying Theorem 1 to G_{cPCG}, we prove the corollary. □

Corollary 2. *The PCG does not contain any irrelevant condition 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 all the irrelevant condition nodes will be eliminated when G_{cPCG} is constructed. According to Definition 17, a node c is an irrelevant condition node if there is a subgraph S that contains c, all its branch edges, S has no relevant nodes (colored nodes), and $|\textit{suc(S)}| = 1$. It follows from this definition and Corollary 1 that G_{cPCG} does not contain any irrelevant condition nodes. Thus, the final graph PCG also does not contain any irrelevant condition nodes, because it consists of the colored nodes in G_{cPCG} and the non-colored nodes resulting from expanding each remaining contracted SCC in G_{cPCG}. □

Note: According to Definition 12, $A(G)$ and $A(\hat{G})$ denote the acyclic graphs derived from the CFG G and its PCG $\hat{G}$.

Definition 23. *Let $\mathcal{E}$ denotes the set of relevant nodes in a CFG G, then a* ***path equivalence relation*** *$\mathcal{R}_{\mathcal{E}}$ is an equivalence relation on the set of paths in $A(G)$ such that two paths are equivalent iff they have the same relevant base behavior.*

Corollary 3 (**Optimality of PCG**). *Let $\mathcal{E}$ be the set of relevant nodes in a CFG G and let $\hat{G}$ be the corresponding PCG. Then, PCG has the optimality property (Definition 21).*

Proof. Let $\mathcal{R}_{\mathcal{E}}$ be the path equivalence relation (Definition 23) defined on G. Let $A(G)$ and $A(\hat{G})$ be the respective acyclic graphs for G and $\hat{G}$.

To prove the optimality property (Definition 21), let us show a one-to-one and onto mapping between the equivalence classes of $\mathcal{R}_{\mathcal{E}}$ and the paths in $A(\hat{G})$ such that each equivalence class and the corresponding path are associated with the same relevant base behavior.

Since the PCG retains all the relevant statements and the relevant condition nodes (Corollary 2), it follows that each path in $A(\hat{G})$ yields a distinct relevant base behavior. Each equivalence class also corresponds to a distinct relevant base behavior. Thus, we get one-to-one and onto mapping between the equivalence classes of $\mathcal{R}_{\mathcal{E}}$ and the paths in $A(\hat{G})$ such that each equivalence class and the corresponding path are associated with the same relevant base behavior. □

Note: A tighter path equivalence relation can be defined by introducing the notion of semantically equivalent relevant nodes. The established optimality is with respect to a weaker path equivalence which does not take into account the semantic equivalence of relevant nodes.

Theorem 2. *If there exists a feasible CFG path P with a relevant base behavior B then there exists a feasible PCG path $\hat{P}$ with the relevant base behavior B.*

Proof. Note that the conditions on each CFG path P contains the set of conditions on the corresponding PCG path $\hat{P}$. Thus, if P is feasible then $\hat{P}$ is also feasible. Also, the relevant statements and relevant conditions on P are retained on $\hat{P}$; so, it has the same relevant base behavior B as the path P. Thus, if there is a feasible CFG path P with a relevant base behavior B then there is a feasible PCG path $\hat{P}$ with the relevant base behavior B. □

Remark 2. *The converse of Theorem 2 would hold except for the **problematic scenario** described below. By the definition of irrelevant condition nodes (Definition 17), an equivalence class has paths going through all possible branches at an irrelevant condition node. So, given a PCG path $\hat{P}$ with relevant base behavior B, we can always choose feasible branches at irrelevant condition nodes to construct a CFG path P with a relevant base behavior B that is feasible with respect to its governing conditions.*

*The **problematic scenario** is one in which all the CFG paths corresponding to a PCG path have a statement S (e.g., an infinite loop) that is not otherwise relevant but it makes a part of the CFG path unreachable. Since the statement S is not included in the PCG, the complete PCG path is reachable. Thus, we can have a PCG path $\hat{P}$ with a relevant base behavior B that is not observable in the CFG because the corresponding CFG paths are broken due to the statement S.*

Note: Theorem 2 shows that path feasibility can be more efficiently computed due to the PCG because only the relevant conditions identified by the PCG need to be used for checking the feasibility.

4.5. CFG to PCG Mapping as a Graph Homomorphism

Let us conclude this section by pointing out an important connection to graph homomorphisms. Graph homomorphisms are mappings that preserve adjacency of vertices and they are widely used in applications involving graph coloring and other problems [19].

The CFG to PCG mapping is a graph homomorphism. Intuitively, a homomorphism from G to H is a partition of G, each node of H represents one partition, the edges in H correspond to the edges between partitions. Given partitions P_1 and P_2, there is an edge from P_1 to P_2 if and only if there exists x in P_1, y in P_2, and (x, y) is an edge in G. A precise definition of *graph homomorphism* for directed graphs is as follows. Let G and H be directed graphs. Let $V(G)$ and $V(H)$ be their node sets and $E(G)$ and $E(H)$ be their edge sets. A homomorphism of G to H is a mapping $f : V(G) \rightarrow V(H)$ such that

1. if $\exists\ x, y \in V(G)$ such that $f(x) = f(u)$, $f(y) = f(v)$, $(x, y) \in E(G)$, and $f(u) \neq f(v) \implies (f(u), f(v)) \in E(H)$;
2. if $\nexists\ x, y \in V(G)$ such that $f(x) = f(u)$, $f(y) = f(v)$, $(x, y) \in E(G) \implies (f(u), f(v)) \notin E(H)$.

In CFG to PCG, the homomorphism is a one-to-one mapping between the nodes for relevant statements and relevant conditions and the other nodes in the CFG are mapped to their relevant successors in the PCG.

5. PCG Tool Support

We developed a PCG toolbox to facilitate the use of the PCG for program comprehension, analysis, and verification. The toolbox provides: (1) “PCG Smart View” - an interactive visual analysis mechanism, and (2) APIs to construct and use PCGs in automated analyses. The toolbox is designed to support multiple programming languages. We have developed the PCG toolbox as an Atlas [14, 13] plug-in to leverage features including: multi-language support, a graph database, a query language, the eXtensible Common Software Graph Schema (XCSG), a variety of program analyzers, and interactive program graph visualization. The toolbox is deployed as an Eclipse [20] plug-in, so it can also leverage the Eclipse infrastructure.

5.1. PCG Toolbox Workflow

Using the PCG toolbox involves the following two phases:

1. *Relevant statement selection*: The user may do so interactively by clicking on code statements or by clicking on the corresponding CFG nodes. Alternatively, the user can invoke an automated analyzer (e.g., for the DBZ problem (Figure 1), we use an analyzer that computes the Use-Def (UD) chains).

2. *PCG construction*: The PCG construction is based on the CFG to PCG transformation algorithm in Section 4. The step can be done interactively through a visual interface called the PCG Smart View or programmatically by using the toolbox APIs.

5.2. PCG Toolbox Infrastructure

The PCG toolbox leverages the Eclipse and Atlas infrastructure for:

- *Visual interactions using program graphs or source code*: This capability is used in the toolbox to select relevant statements by creating a CFG and clicking on CFG nodes, synchronizing these selections with creation and visualization of the corresponding PCG as it evolves in response to selections. A similar capability is provided to select relevant statements by clicking on the source code.

- *Queries through the Atlas Shell*: This capability is used in the toolbox in two ways: (a) to build a set of relevant statements using queries, and (b) to use the PCG as input for a subsequent analysis.

- *Write Java programs with the toolbox APIs and Atlas Queries*: This capability is used to build a verifier or analyzer that obtains the PCG and operates on it for further processing. Section 6.1 describes our PCG-based verifier for lock/unlock pairing in the Linux kernel.

- *Analyzers in Atlas*: As discussed earlier, a number of analyzers in Atlas can be used with the toolbox to select relevant statements. For example, the toolbox uses an Atlas-based analyzer for detecting loops based on the DLI algorithm [21]. The analyzer identifies all of the loop back edges.

- *XCSG Schema*: XCSG provides support for multiple languages, which provides a base for the toolbox to do the same.

- *Graph Database*: Atlas uses attributed graphs [13] for representing program semantics. An analyzer can use the Atlas tagging mechanism to add attributes to nodes and edges of a program graph. The tagging has multiple uses including its use for analyses to communicate with each other. As an example, we use the loop-detection analyzer to compute and tag the loop back edges. These loop back edges are then used by another analyzer to create an acyclic graph to compute the paths corresponding to relevant base behaviors.

Section 6 and Section 7 present case studies on how we use the PCG toolbox interactively via the PCG Smart View and programmatically via the toolbox APIs to verify the lock/unlock pairing in the Linux kernel and detect side-channel vulnerability in Java bytecode.

6. Case Study 1: Linux Verification

The PCG is applicable for verifying the *matching* and *anti-matching* safety and security properties discussed in Section 2. We present a Linux verification study to show the practical benefits of using PCGs in automated and interactive analyses.

6.1. PCG-Based Automated Verification

We developed a PCG-based automated analyzer to verify the lock/unlock pairing in the Linux kernel [9]. The PCG-based verifier is developed using the PCG toolbox APIs (Section 5). The study compares the verification results of the PCG-based verifier with the Berkeley Lazy Abstraction Software Verification Tool (BLAST) [10]. This tool, top rated in the software verification competition (SV-COMP) [11], is used by the Linux Driver Verification (LDV) organization [12]. The problem we have chosen is to verify the lock/unlock pairing on all feasible paths. The study is based on three versions of the Linux operating system with altogether 37 million lines of code and 66, 609 `Lock` instances.

6.1.1. PCG-Based Automated Verification: Performance Improvements

BLAST verifies 43, 766(65.7%) of `Lock` instances as safe; it is inconclusive (crashes or times out) on 22, 843 instances. BLAST does not find any unsafe instances. BLAST required 172 hours and 56 minutes for its verification. The PCG-based automated verification tool verifies 66, 151(99.3%) of `Lock` instances as safe, and it is inconclusive on 451 instances. Seven unsafe instances found through our study were reported as bugs to the Linux organization. These were accepted and fixed. The PCG-based verifier required 3 hours and 24 minutes.

Since the analysis work is related to the size of the CFG or PCG, we use the size reduction from CFG to PCG as the metric to measure the work reduction. Figure 9 shows the distribution of nodes, edges, and condition nodes for both the CFGs and PCGs for all the relevant functions for lock/unlock pairing analysis in Linux kernel (v3.19-rc1). Compared to 30, 914 CFGs, only 115 PCGs have more than 30 nodes, which is a reduction of 99%. Compared to 35, 145 CFGs, only 879 PCGs have more than 30 edges, which is a reduction of 97%. Compared to 17, 120 CFGs, only 1, 810 PCGs have more than 10 condition nodes, which is a reduction of 89%. Recall that PCGs simplify path feasibility checks by reducing the number of condition nodes. Compared to 8, 644 CFGs, 30, 999 PCGs have no condition nodes, which is a 259% increase of cases where PCGs eliminate the need for path feasibility checks.

The reduction from CFG to PCG is particularly important for a CFG with a large number of condition nodes. Table 2 lists the reductions for the ten

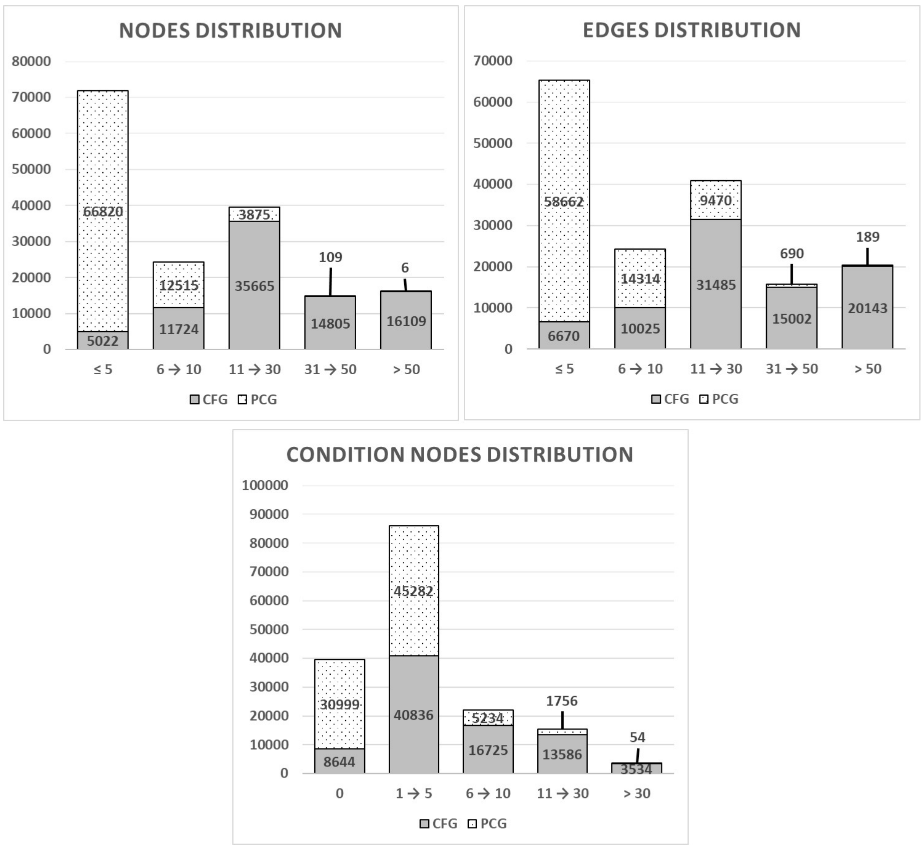


Figure 9: Linux kernel CFG to PCG reduction for lock/unlock pairing

functions with the largest number of condition nodes from the Linux kernel (v3.19-rc1). For example, for function `ptlrpc_connect_interpret` the reductions from CFG to PCG are: from 791 nodes to 8 nodes, from $1,000$ edges to only 9 edges, and from 214 condition nodes to 2 condition nodes. For function `arcnet_interrupt` the reduction from CFG to PCG are: from 4-million paths in $A(G)$ to only 2 paths in $A(\hat{G})$. The number of paths are for the acyclic graphs $A(G)$ and $A(\hat{G})$ (Definition 12) corresponding to the CFG G and the PCG $\hat{G}$ respectively.

6.2. Interactive Verification using PCG

In a landmark paper [22] on program verification and proofs in mathematics, De Millo, Lipton and Perlis (the first recipients of the Turing Award) argue that tools for program verification must provide evidence to support verification. With the growing need for software assurance for mission-critical systems, there is renewed interest in automated verification with evidence [23]. The bugs we have found in the results of automated verification substantiate the argument

Table 2: A comparison of CFG vs. PCG

Function Name	Nodes		Edges		Conditions		Paths	
	CFG	PCG	CFG	PCG	CFG	PCG	CFG	PCG
`ptlrpc_connect_interpret`	791	8	1,000	9	214	2	380,414	3
`kiblnd_passive_connect`	668	24	840	40	174	17	34,216	18
`client_common_fill_super`	644	17	801	29	162	13	1,724,067	14
`qib_make_ud_req`	630	9	833	13	160	5	20,586	6
`xfrm6_input_addr`	574	8	769	11	151	4	1,719	7
`kiblnd_create_conn`	568	16	714	27	149	12	3,748	12
`jbd2_journal_commit_transaction`	522	4	648	3	127	0	2,697	1
`ceph_writepages_start`	416	13	540	21	126	9	1,004	7
`arcnet_interrupt`	408	6	588	6	183	1	4,004,200	2
`macsec_post_decrypt`	390	8	521	9	104	2	1,381	3

for evidence. The PCG-based automated verification tool produces supporting evidence. The evidence includes the CFG and the PCG for each analyzed function. Overall, PCGs are smaller than corresponding CFGs. The CFG and the PCG graphs for $66,609$ `Lock` instances are posted on a website [24].

6.2.1. PCG-Based Interactive Analysis User Study

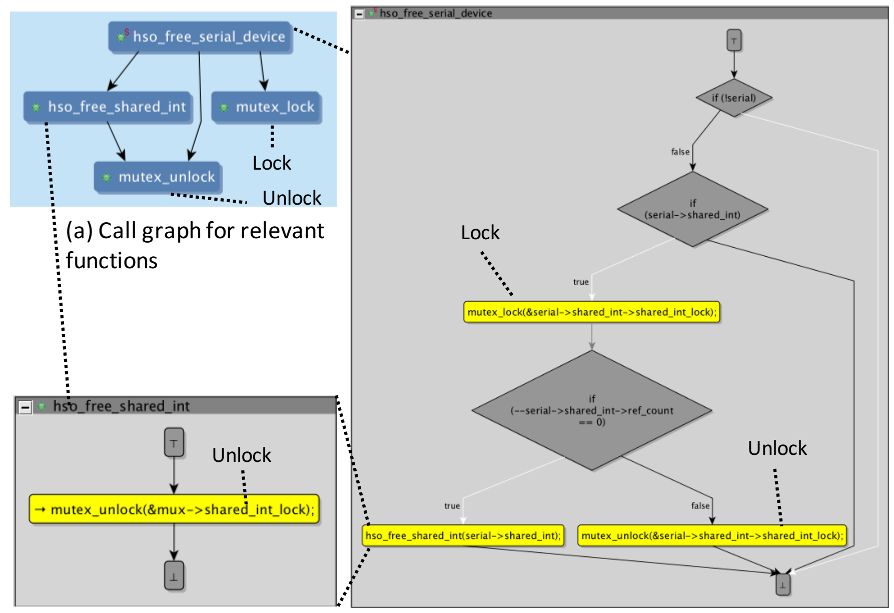
The study was performed by 36 undergraduate students from a software engineering course and 4 graduate students from our research group. We chose 400 `Lock` instances of varying difficulty for the audit. We asked the students to use the PCG and CFG graphs from the website [24] and PCG Smart View (Section 5) to audit the verification results. Each `Lock` instance was independently audited by two undergraduate students and also by two graduate students.

We report here four representative examples of interesting findings from interactive analysis using PCGs. We present a difficult verification instance (Example 4), which requires an analysis of function pointers. The instance brings out the need for interactive analysis for cases which can be inordinately difficult for a completely automated analysis. This instance is incorrectly verified as safe by the automated analysis of the BLAST tool. The PCG-based verification tool is inconclusive on this instance but it provides evidence that is quite valuable for a human analyst to complete the verification.

Example 1: Correct Lock/Unlock Pairing

The first example shows the PCG serving as valuable evidence to facilitate manual verification of correct lock/unlock pairing. Figure 10(a) shows the functions that must be examined for the `Lock` in the function `hso_free_serial_device`. Figures 10(b) and 10(c) show the PCGs for the functions `hso_free_shared_int` and `hso_free_serial_device`, respectively.

In this example, it is easy to observe from the PCG of `hso_free_serial_device` that the `Lock` is followed by a condition node with two paths: (1) one path leads to a matching `Unlock` (intra-procedural), and (2) the other path leads to a call to


hso_free_serial_device
hso_free_shared_int
mutex_lock
mutex_unlock
Lock
Unlock
hso_free_serial_device
if (!serial)
false
if (serial->shared_int)
Lock
true
mutex_lock(&serial->shared_int->shared_int_lock);
if (--serial->shared_int->ref_count == 0)
Unlock
true
false
hso_free_shared_int(serial->shared_int);
mutex_unlock(&serial->shared_int->shared_int_lock);
hso_free_shared_int
Unlock
mutex_unlock(&mux->shared_int_lock);

(a) Call graph for relevant functions

(b) PCG for hso_free_shared_int

(c) PCG for hso_free_serial_device

Figure 10: An example of correct lock/unlock pairing

function `hso_free_shared_int` (inter-procedural). The PCG of the called function `hso_free_shared_int` shows a matching `Unlock` on all paths within that function.

Example 2: Inconclusive Verification by BLAST

The second example shows the PCG serving as valuable evidence to manually cross-check an instance for which the BLAST verification is inconclusive, i.e., BLAST cannot determine if the pairing happens correctly or not. Our cross-checking with the help of a PCG revealed that it is an unsafe instance. This instance was reported as a bug and the Linux organization fixed the bug.

Figure 11 shows the PCG for the function `toshsd_thread_irq` that has calls to `Lock` and `Unlock`. The CFG for this function is more complex with 8 condition 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 CFG can be viewed at the website [24].

The PCG for `toshsd_thread_irq` shows a path on which the `Lock` is not followed by an `Unlock`. As seen from the PCG, the path is feasible if its governing conditions $C_1 =$ `false` and $C_2 =$ `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.

Example 3: Verification Involves a Loop

The third example shows the PCG serving as valuable evidence to reason about cases where the `Lock` happens inside a loop. This instance looks like a bug

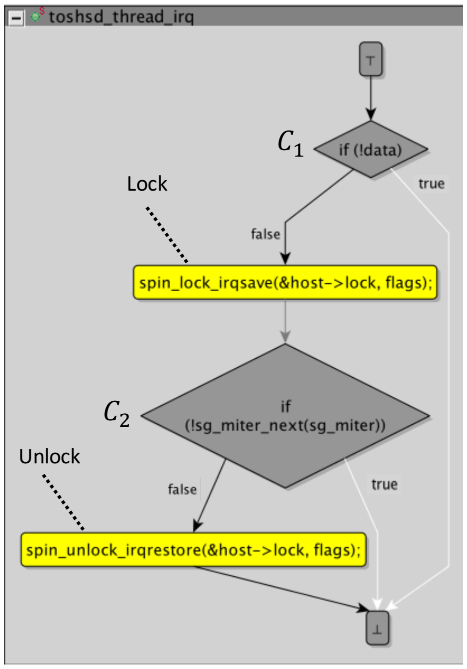
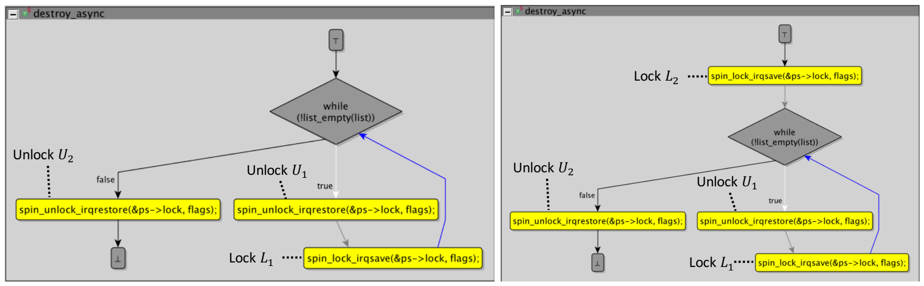

toshsd_thread_irq
⊤
C_1
if (!data)
true
Lock
false
spin_lock_irqsave(&host->lock, flags);
C_2
if (!sg_miter_next(sg_miter))
Unlock
false
true
spin_unlock_irqrestore(&host->lock, flags);
⊥

Figure 11: A Linux bug discovery using a PCG as evidence

at a cursory glance, but a careful review using the PCG shows that it is not a bug. The PCG in this example is an unusual case because our PCG-based verification verifies one lock at a time. The PCG in Figure 12(a) shows that the `Lock` L_1 is matched correctly on two paths with the `Unlock` U_1 and `Unlock` U_2. However, U_1 is dangling upon the entry to the loop. This raises the question of whether there is another `Lock` before the loop, which would be required for a correct pairing. Since a separate PCG is created for each instance of `Lock`, the other `Lock` is not seen in this PCG. Creating a PCG considering all `Locks` and `Unlocks` in this function shows that there is another `Lock` L_2 before the loop as shown in Figure 12(b). Thus, it is not an error in Linux, but this unusual situation does require a careful review of the PCG to confirm that. A possible refinement of the current PCG-based verifier is to include all the `Locks` on a single lock object together as relevant statements.

Example 4: A Verification Instance with a Quirk

The fourth example shows the PCG serving as a valuable evidence that helped us find an erroneous verification by BLAST. BLAST verifies this instance as safe. An unusual programming style revealed by the PCG made us suspect the BLAST verification. This turns out to be a complex scenario for verification, but we were able to show that it is an unsafe instance [25]. This instance was reported as a bug which the Linux organization fixed. The reason why this


destroy_async
while (!list_empty(list))
Unlock U_2
false
Unlock U_1
true
spin_unlock_irqrestore(&ps->lock, flags);
spin_unlock_irqrestore(&ps->lock, flags);
Lock L_1
spin_lock_irqsave(&ps->lock, flags);
destroy_async
Lock L_2
spin_lock_irqsave(&ps->lock, flags);
while (!list_empty(list))
Unlock U_2
false
Unlock U_1
true
spin_unlock_irqrestore(&ps->lock, flags);
spin_unlock_irqrestore(&ps->lock, flags);
Lock L_1
spin_lock_irqsave(&ps->lock, flags);

(a) PCG with respect to: L_1, U_1, and U_2

(b) PCG with respect to: L_1, L_2, U_1, and U_2

Figure 12: A PCG indicating a missing lock preceding a loop

scenario is complex has to do with extra layers of indirection [25], which we describe below.

Figure 13 shows the PCG for the function `drxk_gate_crtl`. The PCG shows that the `Lock` *cannot* be matched by the `Unlock` as the `Lock` and `Unlock` are on two mutually exclusive paths. The mutually exclusive paths are governed by the branch node marked as C. If $C =$ `true`, the `Lock` executes, otherwise the `Unlock` executes.

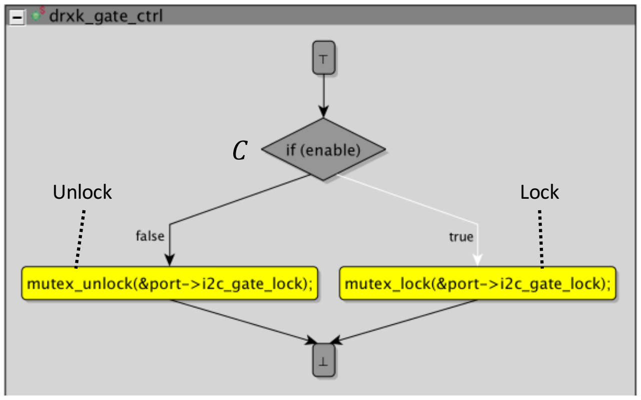

drxk_gate_ctrl
C
if (enable)
Unlock
Lock
false
true
mutex_unlock(&port->i2c_gate_lock);
mutex_lock(&port->i2c_gate_lock);

Figure 13: A PCG Quirk

The `Lock` and `Unlock` on disjoint paths could pair with each other if function `drxk_gate_crtl` is called twice, first with $C =$ `true` then with $C =$ `false`. This amounts to using `drxk_gate_crtl` first as a lock and then as an unlock. A quick query using Atlas shows that `drxk_gate_crtl` is not called directly anywhere. The function could be either dead code or called via function pointers. Our PCG-based automated verification tool currently lacks function pointer analysis. Using advanced Atlas queries, we can see that `drxk_gate_crtl` is called twice using function pointers in function `tuner_attach_tda18271`. However, there is a path on which there is a return before the second call which makes `drxk_gate_crtl` act as unlock and thus a bug.

This example shows an unusual programming pattern which would be intractable for a fully automated verification. A human-in-the-loop evidence-supported approach is crucial to handle such difficult cases.

7. Case Study 2: Detecting a Side-Channel Vulnerability

Detecting sophisticated Side-Channel Vulnerabilities (SCVs) [26] is like searching for a needle in haystack without knowing what the needle looks like. It often requires domain-specific knowledge [27, 28]. Detection involves exploring software to identify vulnerable code, conceiving plausible attack hypotheses, and analyzing software to gather evidence to prove or refute each hypothesis.

The PCG helps the analyst to focus on the space and time changing events and their governing conditions. If executing such events causes observable space/time differences then it creates the possibility of an SCV. The governing conditions need to be user-input controlled for an attacker to force the execution of paths with observable space/time differences. Thus, to detect SCVs, the analyst must understand the program to answer specific questions: (a) What are the space/time changing events present in the program? (b) What are the governing conditions controlling the execution of these events? (c) Can the governing conditions be controlled by user-inputs? Three phases of the interactive analysis are:

- **Phase I:** *Automated Exploration.* The objective is to precompute information that serves as the basis for the analyst to begin the investigation. The precomputed information includes the locations of space/time changing loops and the user-input controlled conditions.
- **Phase II:** *Hypothesis Formulation.* After reviewing the precomputed information, the analyst hypothesizes possibilities for SCVs. By the end of Phase II, the analyst has hypotheses that need to be either validated or refuted.
- **Phase III:** *Validating the Hypotheses.* The objective is to: (1) enable the analyst to gather evidence to refine, refute, or validate each hypothesis formulated in Phase II, and (2) help the analyst compose the overall *modus operandi* of the attack.

Illustrating Example

This example illustrates an interactive use of the PCG to analyze Java bytecode to detect SCVs. Consider a simple password checking app that compares the passwords stored in a server against user-input strings submitted as passwords. The app `Accepts` if the submitted string matches with a stored password. The app `Rejects` if the match fails.

At the end of Phase I, the analyst observes the statement `Thread.sleep(25)` as a time-changing event. The analyst then needs to check whether there are any user-input controlled conditions that govern the time-changing event. With the help of an Atlas-based taint flow analyzer, the analyst can explore the taint flow graph shown in Figure 14. The taint flow graph shows: (1) the program statements in the app tainting the secret (i.e., the passwords stored on the server) highlighted with red color, (2) the program statements in the app tainting the user-input (i.e., the string that can be submitted via the user as a password) highlighted with blue color, and (3) the program statements that mutually taint the secret and the user-input highlighted with yellow color. Based on the taint analysis result, the analyst observes that the taints from the secret and the user-input come together at conditions C_1 and C_2. These conditions are user-input controlled conditions and both compare the user-input against the secret passwords on the server.

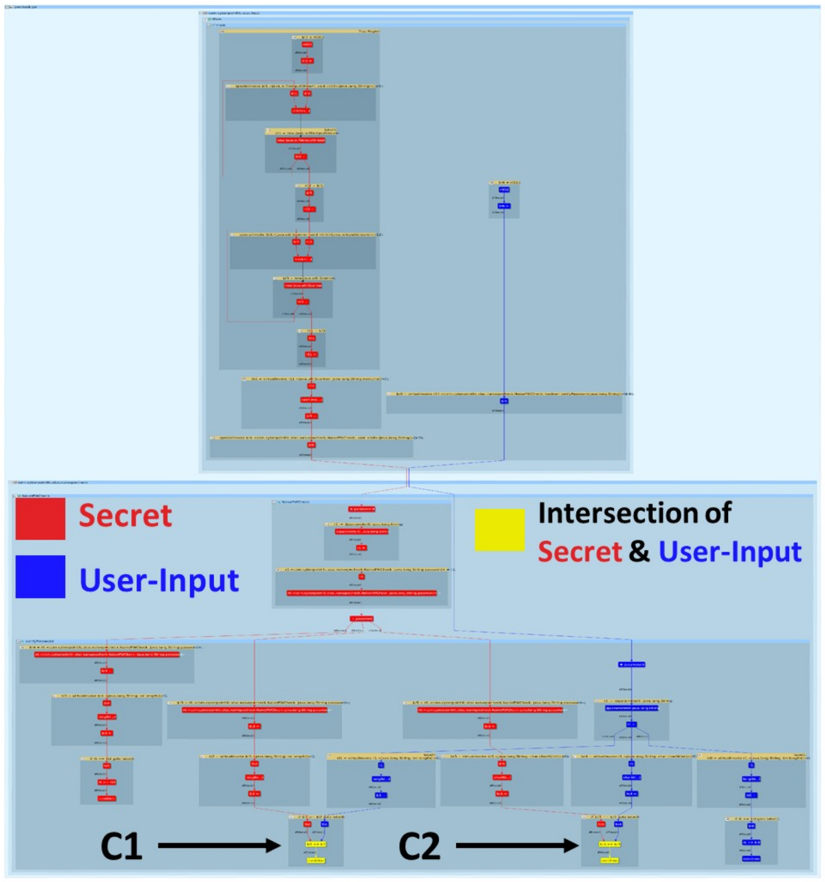
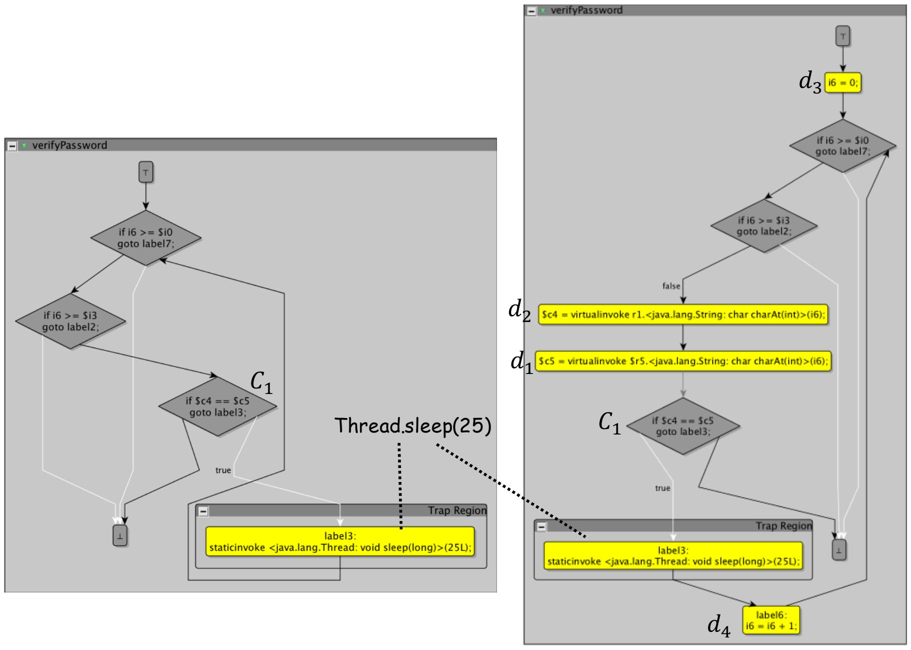


Figure 14: Taint flows from the secret and the user-controlled input

In Phase II, the analyst hypothesizes that, depending on the comparison result of the secret passwords with the submitted password, two paths are created either by the condition C_1 or C_2 such that the time-changing event `Thread.sleep(25)` happens on only one of those two paths. Investigating the conditions C_1 and C_2, the analyst sees that the secret and the user-input password are compared one character at a time. Thus, if time difference can be observed by an attacker, it can reveal to the attacker that there is a character match. By submitting different strings for the password and observing the time differences, the attacker can learn the secret password. To validate the hypothesis, the analyst gathers evidence in Phase III to answer the following questions:

1. Do the conditions C_1 or C_2 determine whether the time-changing event `Thread.sleep(25)` will execute? In other words, are C_1 and C_2 relevant conditions for the time-changing event?
2. Do the conditions depend on character-wise comparison of secret and user-input?

The PCG is useful for answering these questions. To answer the first question, the analyst creates the PCG using `Thread.sleep(25)` as a relevant statement. The resulting PCG in Figure 15(a) shows that the condition C_1 actually governs the relevant statement `Thread.sleep(25)` when C_1 = `true`. Thus, the execution of `Thread.sleep(25)` is dependent on the comparison of the user-input and the secret passwords.


verifyPassword
if i6 >= $i0 goto label7;
if i6 >= $i3 goto label2;
C_1
if $c4 == $c5 goto label3;
true
Thread.sleep(25)
Trap Region
label3: staticinvoke <java.lang.Thread: void sleep(long)>(25L);
verifyPassword
d_3
i6 = 0;
false
d_2
$c4 = virtualinvoke r1.<java.lang.String: char charAt(int)>(i6);
d_1
$c5 = virtualinvoke $r5.<java.lang.String: char charAt(int)>(i6);
d_4
label6: i6 = i6 + 1;

(a) PCG with respect to Thread.sleep(25)

(b) PCG with respect to Thread.sleep(25) and data flow statements for C_1

Figure 15: PCGs with respect to `Thread.sleep(25)` and data flow statements for C_1

To answer the second question, the analyst uses Atlas data flow analyzer to find the statements that belongs to the data flow for condition C_1. The data flow analyzer returns the statements d_1, d_2, d_3 and d_4. Then, the analyst adds these statements to the set of relevant statements with `Thread.sleep(25)` to construct the PCG shown in Figure 15(b). The new PCG reveals the character-wise comparison and the execution of `Thread.sleep(25)` is dependent on the condition C_1 comparing the user-input and the secret passwords. Thus, confirming the existence of the SCV.

8. Related Work

The papers by Choi *et al.* [29] and Ramalingam [30] 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. The fixed-point algorithm to compute the SEG aggregates data flow behaviors on different paths.

The Binary Decision Diagram (BDD) [31] has been used in different contexts of program analysis as a way to reduce the explosion of state space [6, 32]. Unlike the approaches [6, 32], the PCG does not use a heuristic based on loop unrolling and efficiently computes the relevant behaviors without the need to compute all behaviors. The Binary Decision Tree (BDT) to BDD reduction has been also used for path-sensitive analysis [7]. Unlike BDT to BDD reduction, the PCG transformation does not require the input CFG to be acyclic and each path in the acyclic PCG corresponds to a unique distinct relevant base behavior.

Das *et al.* [1] proposed a path-sensitive program verification in polynomial time. Their approach propagates symbolic states relevant to a particular analysis along control flow paths. At a merge point for a given condition node, the approach merges the symbolic states and eliminates a condition node and its branches if the propagated symbolic states are not affected by the statements along the branches.

CFG pruning techniques have been proposed in [33, 34] to overcome the computational complexity of exploring all paths. However, the resultant compact CFG does not achieve the optimality (Definition 21) the PCG achieves. The PCG has evolved from our earlier research on *event view* [35] and *event flow graphs* [36].

9. Conclusion

This paper presents an efficient and accurate approach to analyze software for a broad spectrum of safety and security vulnerabilities. The paper provides a rigorous formulation and a practical method to apply the approach. The approach is illustrated with a challenging analysis problem of lock/unlock pairing on all feasible execution paths. The tool support for the approach is developed using Atlas [13, 14], a platform for developing software analysis and visualization tools.

The approach involves transforming a CFG to a 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 [24].

As a study of interactive analysis using the PCG, 400 `Lock` instances of varying difficulty were chosen for auditing by undergraduate and graduate students. We reported four representative examples of interesting findings from the study. These examples show the PCG serving as a valuable evidence to facilitate manual verification of correct lock/unlock pairing. Our manual verification with the help of PCGs has revealed bugs in the Linux kernel - bugs that were missed by a top rated formal verification tool.

We describe the use of the PCG to detect sophisticated side-channel vulnerabilities (SCVs). Detecting SCVs requires domain-specific knowledge [27, 28]. It involves exploring software to identify vulnerable code, conceiving plausible attack hypotheses, and analyzing software to gather evidence to prove or refute each hypothesis. We present a case study to show that the PCG is useful when the analyst has to gather evidence to prove or refute an SCV.

Acknowledgment

This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0126. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. We thank our colleagues at the Knowledge-Centric Software (KCS) Engineering Lab at Iowa State University and the colleagues at EnSoft for helping us with our research. We are extremely grateful to the reviewers for their thoughtful and constructive reviews that have resulted in significant improvements in the paper. Dr. Kothari is the founding President of EnSoft.

References

[1] M. Das, S. Lerner, M. Seigle, ESP: Path-sensitive Program Verification in Polynomial Time, in: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, PLDI '02, ACM, New York, NY, USA, 2002, pp. 57–68.

[2] L. Carter, J. Ferrante, C. Thomborson, Folklore Confirmed: Reducible Flow Graphs are Exponentially Larger, in: Proc. of the 30 th ACM SIGPLANSIGACT Symposium on Principles of Programming Languages, ACM Pr, 2003, pp. 106–114.

[3] M. N. Ngo, H. B. K. Tan, 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, ESEC-FSE '07, ACM, New York, NY, USA, 2007, pp. 215–224.

[4] V. Vojdani, V. Vene, Goblint: Path-sensitive data race analysis, Annales Univ. Sci. Budapest., Sect. Comp 2009.

[5] A. Navabi, N. Kidd, S. Jagannathan, Path-Sensitive Analysis Using Edge Strings, Purdue University Technical Report 10-006.

[6] T. Ball, S. K. Rajamani, Bebop: A Path-sensitive Interprocedural Dataflow Engine, in: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '01, ACM, New York, NY, USA, 2001, pp. 97–103.

[7] Z. Xu, J. Zhang, Path and Context Sensitive Inter-procedural Memory Leak Detection, in: 2008 The Eighth International Conference on Quality Software, 2008, pp. 412–420.

[8] I. Dillig, T. Dillig, A. Aiken, Sound, Complete and Scalable Path-sensitive Analysis, in: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08, ACM, New York, NY, USA, 2008, pp. 270–280.

[9] S. Kothari, P. Awadhutkar, A. Tamrawi, J. Mathews, Modeling Lessons from Verifying Large Software Systems for Safety and Security, in: Cyber-Physical Systems Special Track at Winter Simulation Conference (WSC), 2017, to appear.

[10] D. Beyer, T. A. Henzinger, R. Jhala, R. Majumdar, The Software Model Checker Blast: Applications to Software Engineering, International Journal on Software Tools Technology Transfer 9 (5) (2007) 505–525.

[11] D. Beyer, Status report on software verification, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2014, pp. 373–388.

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

[13] T. Deering, S. Kothari, J. Sauceda, 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.

[14] Atlas Platform, EnSoft Corp., `http://www.ensoftcorp.com`.

[15] Common weakness enumeration, `http://cwe.mitre.org`.

[16] C. Perrin, The CIA triad, `http://www.techrepublic.com/blog/security/the-cia-triad/488`.

[17] S. A. Cook, The Complexity of Theorem-proving Procedures, in: Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC '71, ACM, New York, NY, USA, 1971, pp. 151–158.

[18] R. Tarjan, Depth-first search and linear graph algorithms, SIAM journal on computing 1 (2) (1972) 146–160.

[19] P. Hell, J. Nesetril, Graphs and homomorphisms, Oxford University Press, 2004.

[20] The Eclipse Foundation open source community website, `http://www.eclipse.org`.

[21] T. Wei, J. Mao, W. Zou, Y. Chen, A New Algorithm for Identifying Loops in Decompilation, in: Proceedings of the 14th International Conference on Static Analysis, SAS'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 170–183.

[22] R. A. De Millo, R. J. Lipton, A. J. Perlis, Social Processes and Proofs of Theorems and Programs, Commun. ACM 22 (5) (1979) 271–280.

[23] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, Correctness witnesses: exchanging verification results between verifiers, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2016, pp. 326–337.

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

[25] S. Kothari, A. Tamrawi, J. Mathews, Human-machine resolution of Invisible Control Flow?, in: Program Comprehension (ICPC), 2016 IEEE 24th International Conference on, IEEE, 2016, pp. 1–4.

[26] DARPA-BAA-14-60: Space/Time Analysis for Cybersecurity (STAC), `https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-14-60/listing.html`.

[27] D. Brumley, D. Boneh, Remote Timing Attacks Are Practical, in: Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12, SSYM'03, USENIX Association, Berkeley, CA, USA, 2003, pp. 1–1.

[28] A. Futoransky, D. Saura, A. Waissbein, Timing attacks for recovering private entries from database engines, BlackHat USA.

[29] J.-D. Choi, R. Cytron, 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.

[30] G. Ramalingam, On Sparse Evaluation Representations, Vol. 277, Elsevier Science Publishers Ltd., Essex, UK, 2002.

[31] S. B. Akers, Binary Decision Diagrams, IEEE Trans. Computers 27 (6) (1978) 509–516.

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

[33] M. K. Ramanathan, A. Grama, S. Jagannathan, Path-Sensitive Inference of Function Precedence Protocols, in: Proceedings of the 29th International Conference on Software Engineering, ICSE '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 240–250.

[34] S. Lafortune, S. Mahlke, Y. Wang, T. Kelly, H. Liao, H. K. Cho, Practical Lock/Unlock Pairing for Concurrent Programs, in: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), CGO '13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 1–12.

[35] S. Neginhal, S. Kothari, Event Views and Graph Reductions for Understanding System Level C Code, in: 2006 22nd IEEE International Conference on Software Maintenance, 2006, pp. 279–288.

[36] A. Tamrawi, K. Gui, S. Kothari, Event-Flow Graphs for Efficient Path-Sensitive Analyses.
URL `http://arxiv.org/abs/1404.1279`