# COMB: Computing Relevant Program Behaviors[*]

Benjamin Holland[1], Payas Awadhutkar[1], Suresh Kothari[1], Ahmed Tamrawi[2], Jon Mathews[2]
[1]Iowa State University, Ames, Iowa, 50010
[2]EnSoft Corp., Ames, Iowa, 50010
{bholland,payas,kothari}@iastate.edu,{ahmedtamrawi,jmathews}@ensoftcorp.com

## ABSTRACT

The paper presents COMB, a tool to improve accuracy and efficiency of software engineering tasks that hinge on computing all relevant program behaviors. Computing all behaviors and selecting the relevant ones is computationally intractable. COMB uses *Projected Control Graph* (PCG) abstraction to derive the relevant behaviors directly and efficiently. The PCG is important as the number of behaviors relevant to a task is often significantly smaller than the totality of behaviors.

COMB provides extensive capabilities for program comprehension, analysis, and verification. We present a basic case study and a Linux verification study to demonstrate various capabilities of COMB and the addressed challenges. COMB is designed to support multiple programming languages. We demonstrate it for C and Java. Video url: https://youtu.be/YoOJ7avBIdk

## CCS CONCEPTS

• **Security and privacy → Software and application security**; • **Software and its engineering → Software verification and validation**;

## KEYWORDS

Program Behaviors, Software Analysis, Software Verification

## 1 INTRODUCTION

Many software engineering tasks require analysis and verification of all program behaviors relevant to the task. For example, all relevant behaviors must be analyzed to verify software for safety or security. We will describe COMB capabilities and underlying ideas with the help of examples. Consider the problem of verifying software for *Division-By-Zero* (DBZ) vulnerabilities. Line 24 of the code in Figure 1 involves division by `d`, which must be checked for a DBZ vulnerability.

As described in [10], each Control Flow (CF) path yields a behavior represented by the corresponding *trace*. Each trace is a regular expression of program statements along a path. In this simple example without loops, each trace is a sequence of statements. There are six behaviors and their corresponding traces are: (B1) 8, 9, 10, 11, 15, 18, 19, 24; (B2) 8, 9, 12, 13, 15, 18, 19, 24; (B3) 8, 9, 10, 11, 15, 16, 17, 24; (B4) 8, 9, 12, 13, 15, 16, 17, 24; (B5) 8, 9, 10, 11, 21, 22, 24; (B6) 8, 9, 12, 13, 21, 22, 24. The first two traces exhibit the vulnerability.

```
 1 public class DBZ {
 2     public static int a1 = 0;
 3     public static int a2 = 1;
 4     public static boolean c1 = true;
 5     public static boolean c2 = false;
 6     public static boolean c3 = true;
 7     public static void main(String[] args){
 8         int x = a1 + a2;
 9         int d = a1;
10         if(c1){
11             x = a1;
12         } else {
13             x = a1;
14         }
15         if(c2){
16             if(c3){
17                 int y = a1;
18             } else {
19                 d = d - a1;
20             }
21         } else {
22             d = d + 1;
23         }
24         int z = x / d;
25     }
26 }
```

**Figure 1: A DBZ vulnerability**

Verifying any vulnerability poses two challenges: (1) computing the vulnerable behaviors out of all behaviors and, (2) path feasibility, i.e., checking feasibility of CF paths for the vulnerable behaviors. The large number of behaviors ($2^n$ behaviors for $n$ non-nested `IF` conditions) make the problem computationally intractable. The path feasibility problem is equivalent to the satisfiability problem [5].

An efficient algorithm must compute the vulnerable behaviors without computing all the behaviors. A closer inspection reveals that multiple behaviors can be grouped so that each group correspond to a unique *relevant behavior* defined by a sub-trace that retains only the relevant statements and conditions. The relevant statements for the DBZ vulnerability are: 9, 19, 22, and 24. The relevant behaviors are: (RB1) 9, 15, 18, 19, 24; (RB2) 9, 15, 16, 24; (RB3) 9, 21, 22, 24 with the grouping: RB1: {B1, B2}, RB2: {B3, B4}, RB3: {B5, B6}. Note `if(C1)` is irrelevant because taking the true and false branches produce the same relevant behaviors.

COMB uses the PCG abstraction [9] to compute the relevant behaviors directly and efficiently. The abstraction amounts to an efficient way to form the equivalence classes of behaviors to yield relevant behaviors. The Control Flow Graph (CFG) fuses together multiple facets of the program logic. Unlike the CFG, the PCG distills the behaviors for a particular facet of the program logic. The CFG and its corresponding PCG for the DBZ example are shown in Figure 2. As borne out by the case studies, the use of the PCG can

[*]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.

*ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden*
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3183476

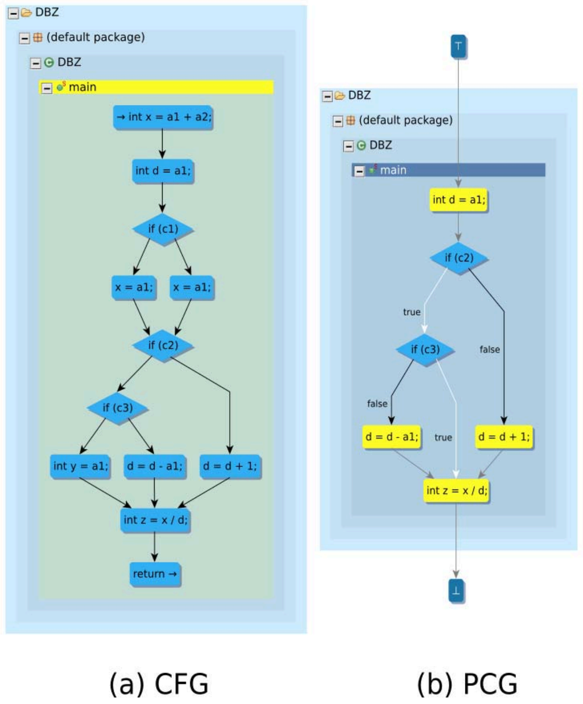


**Figure 2: CFG and PCG for the DBZ vulnerability**

circumvent the challenges of analyzing large number of paths and path feasibility.

## 2 COMB WORKFLOW

Using COMB involves the following steps:

(1) Use COMB to select the relevant statements within a function. The user may do so interactively by clicking on the statements in the source code panel or by clicking on the corresponding CFG nodes from a previously constructed CFG. Alternatively, the user can invoke an automated analyzer (e.g., for the DBZ problem (Section 1), we use an analyzer that computes the Use-Def (UD) chains). The demo shows different ways of selecting relevant statements.

(2) Use COMB to compute the PCG based on the selected relevant statements. This can be done interactively through a visual interface or programmatically by using an API.

(3) The user may use COMB to visualize PCGs for program comprehension. Alternatively, the user may invoke COMB APIs to compute PCGs as a part of an automated analyzer (e.g., an automated verification for the DBZ). The demo includes an automated use of COMB from our safety verification study on the Linux kernel. [9].

## 3 THE PCG ABSTRACTION

The PCG abstraction and an efficient algorithm to compute it are originally described in our 2016 paper [10]. Figure 2 shows the PCG for the DBZ vulnerability discussed in Section 1. User can choose a layout algorithm and apply styling to suit their visualization needs. Nodes for relevant statements are colored yellow and the `True` edges from `IF` are colored white in Figure 2.

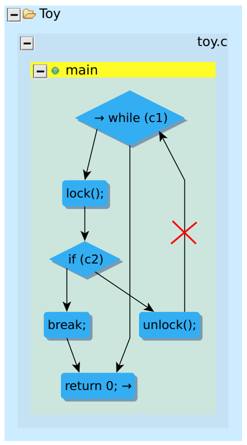


**Figure 3: An illustrative example of loop behavior model. The loop-back edge is marked with a cross.**

With respect to this paper, key points about the PCG are:

- The PCG is obtained by transforming the corresponding CFG. In all our studies, we have found the PCGs are often much smaller than the corresponding CFGs. An extreme example from the Linux verification study [9], the CFG for `client_common_fill_super` has 1,101 nodes, 1,179 edges, and 249 branch nodes. The corresponding PCG has 15 nodes, 28 edges, and 13 branch nodes.
- Filtering out unnecessary details is important to facilitate program comprehension. This is brought out in the demo by an example of a Linux bug that is easily spotted using the PCG.
- Relevant behaviors are derived by traversing the PCG paths. The demo includes a utility to count and compare the number of paths in a CFG and the corresponding PCG.

### 3.1 Loop Behaviors Model

To compute behaviors in presence of loops, a model of loop behaviors is required as loops may iterate several times. Our loop behavior model [10] is intended for a class of problems in which the validity of safety or security properties does not depend on the exact number of times the loop iterates. This model is motivated by an empirical study of loops in open-source C and Java software and the need for a practical approach to account for loop behaviors to verify safety and security properties.

Let us illustrate the model with an example (Figure 3). Consider the problem where a `Lock` must be followed by `Unlock` and it must not be followed by another `Lock`. Consider a loop with `Lock` `Unlock` and a `break` in between. The model yields three relevant behaviors with traces: (1) $(c1, L, \overline{c2}, U)^+$, (2) $(c1, L, \overline{c2}, U)^* c1, L, c2,$ `break`, (3) $\overline{c1}$ with $L$ for `Lock` and $U$ for `Unlock`. $ci$ and $\overline{ci}$ denotes whether the condition $ci$ evaluates to true or false. The second behavior implies that the property is violated. The model requires a unique entry for each loop but allows multiple exits to account for `break` or `return`.

## 4 COMB INFRASTRUCTURE

COMB provides an extensive infrastructure to support program comprehension, analysis, and verification for which analyzing all relevant behaviors is a crucial requirement.

Building this infrastructure from scratch would be a major engineering effort. Instead, we have designed COMB as an Atlas [1, 8] plug-in to leverage its multi-language support, graph database, query language, eXtensible Common Software Graph Schema (XCSG), a variety of program analyzers, and interactive program graph visualization. COMB is written in Java and deployed as an Eclipse plug-in, and so it can also leverage the Eclipse infrastructure. COMB leverages the Eclipse and Atlas infrastructure for the following:

*Visual interactions using program graphs or source code:* This capability is used in COMB 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 COMB in two ways: (a) build a set of relevant statements using queries, and (b) feed the PCG to a query to perform further analysis based on the PCG.

*Write Java programs with COMB 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.

*Analyzers in Atlas:* As discussed earlier, a number of analyzers in Atlas can be used in COMB to select relevant statements. For example, COMB uses a Atlas-based analyzer for detecting loops based on the DLI algorithm [12]. The analyzer identifies all the loop-back edges as shown in Figure 3.

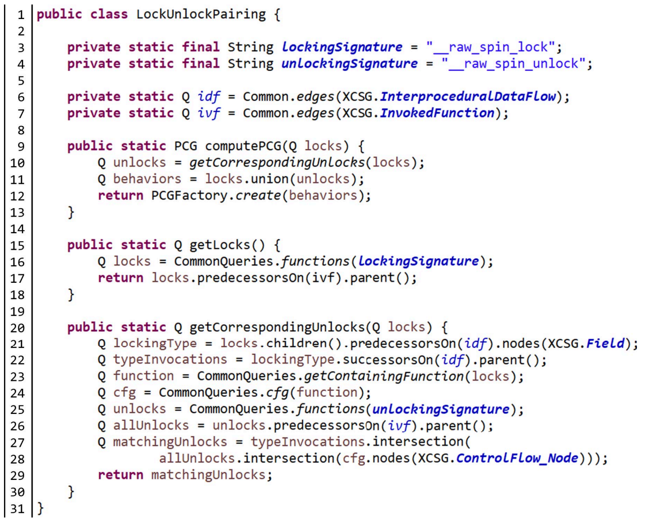
*XCSG Schema:* XCSG provides support for multiple languages, which provides a base for COMB to do the same. The video demonstrates COMB for C and Java.

*Graph Database:* Atlas uses attributed graphs [8] 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 (Figure 3). These loop-back edges are then used by another analyzer to create an acyclic graph to compute the paths corresponding to relevant behaviors.

Figure 4 shows an example of how the COMB infrastructure can be used. The function defined on line 22 is an analyzer written in Java, using COMB APIs and Atlas queries. This analyzer is used to find the matching unlock calls corresponding to a lock call. It is invoked on line 16 and the result is used by the COMB infrastructure on line 18 to compute the PCG. This PCG can then be used to verify whether a lock call has a matching unlock call on all relevant paths.

## 5 COMB CASE STUDIES

We present two case studies to demonstrate COMB: (1) a simple example of DBZ and, (2) the Linux verification study to illustrate its use on real-world software.



```
 1 public class LockUnlockPairing {
 2 
 3     private static final String lockingSignature = "__raw_spin_lock";
 4     private static final String unlockingSignature = "__raw_spin_unlock";
 5 
 6     private static Q idf = Common.edges(XCSG.InterproceduralDataFlow);
 7     private static Q ivf = Common.edges(XCSG.InvokedFunction);
 8 
 9     public static PCG computePCG(Q locks) {
10         Q unlocks = getCorrespondingUnlocks(locks);
11         Q behaviors = locks.union(unlocks);
12         return PCGFactory.create(behaviors);
13     }
14 
15     public static Q getLocks() {
16         Q locks = CommonQueries.functions(lockingSignature);
17         return locks.predecessorsOn(ivf).parent();
18     }
19 
20     public static Q getCorrespondingUnlocks(Q locks) {
21         Q lockingType = locks.children().predecessorsOn(idf).nodes(XCSG.Field);
22         Q typeInvocations = lockingType.successorsOn(idf).parent();
23         Q function = CommonQueries.getContainingFunction(locks);
24         Q cfg = CommonQueries.cfg(function);
25         Q unlocks = CommonQueries.functions(unlockingSignature);
26         Q allUnlocks = unlocks.predecessorsOn(ivf).parent();
27         Q matchingUnlocks = typeInvocations.intersection(
28                 allUnlocks.intersection(cfg.nodes(XCSG.ControlFlow_Node)));
29         return matchingUnlocks;
30     }
31 }
```

**Figure 4: Example use of COMB infrastructure**

### 5.1 DBZ Study

This basic study illustrates a variety of user interactions to: (a) select relevant statements, (b) compute the PCG, (c) use the PCG for further analysis. It brings out the basic concepts of relevant behaviors, traces to represent behaviors, and relevant conditions for behavior feasibility check.

The underlying code (Figure 1) is small and simple so that a user can manually check the results produced by the tool. As shown in Figure 2, the total number of behaviors is six. The number of relevant behaviors is three. One behavior depicts the vulnerability. Two conditions out of three are relevant for the feasibility of the vulnerability.

### 5.2 Linux Verification Study

We designed a COMB-based automated analyzer to verify the lock-/unlock pairing in the Linux kernel [9]. The study compared the COMB-based verifier with the Berkeley Lazy Abstraction Software Verification Tool (BLAST) [4], a top-rated tool in the software verification competition (SV-COMP) [2]. We summarize pertinent results from the study and bring out some uses of COMB for verification. The problem we have chosen is to verify the `Lock`/`Unlock` pairing on all feasible behaviors. The study is based on three versions of the Linux operating system with altogether 37 million lines of code and 66, 609 `Lock` instances to be verified.

BLAST verifies 43, 766 (65.7)% of `Lock` instances as safe, and 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 COMB-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. One of the bugs we found is an instance that BLAST reported as a safe instance. COMB-based verifier required 3 hours and 24 minutes.

In a landmark paper [7] on program verification and proofs in mathematics, De Millo, Lipton and Perlis (the first recipient of the Turing Award) argue that tools for program verification must

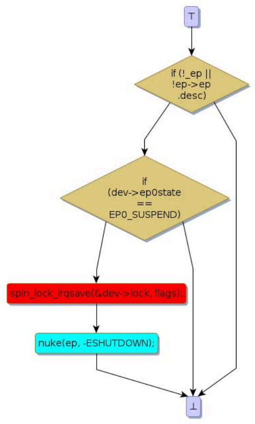

⊤
if (!_ep \|\| !ep->ep .desc)
if (dev->ep0state == EP0_SUSPEND)
spin_lock_irqsave(&dev->lock, flags);
nuke(ep, -ESHUTDOWN);
⊥

**Figure 5: PCG retaining results of inter-procedural analysis**

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 [3]. The bugs we have found in the results of automated verification substantiate the argument for evidence.

The COMB-based verification tool produces supporting evidence. The evidence includes the CFG and the PCG for each analyzed function. Overall, PCGs are smaller than corresponding CFGs. For the Linux version 3.19-rc1, the average CFG to PCG reduction is 73.4% for the number of nodes and 75.5% for the number of edges.

**COMB with inter-procedural analyzer:** Inter-procedural analysis is required to solve many problems and it is important to reflect the results of the analyzer in the PCG-based evidence. Let us briefly discuss how the COMB-based tool brings out results from an inter-procedural analysis. The tool can produce PCGs as shown by the example in Figure 5. As shown, PCG retains the call to the function `nuke` as a relevant call. The call to `Lock` is followed by the call to `nuke`. It suggests that `Unlock` is called inside the function `nuke`. The analyst can then check the PCG for `nuke` to see if it includes `Unlock` calls on all paths.

**A Linux Bug:** This is an example for which BLAST cannot complete the verification. but the PCG reveals a bug. Figure 6 shows the PCG for the function `toshsd_thread_irq` that has calls to `Lock` and `Unlock`. The CFG for this function is complex with 8 branch nodes and multiple loops. PCG simplifies the path feasibility check. The PCG for the function `toshsd_thread_irq` shows a path on which the `Lock` is not followed by an `Unlock`. Using the two relevant conditions shown by the PCG, it can be checked that the vulnerable path is feasible. This bug was reported to the Linux organization and it was fixed.

## 6 RELATED WORK

This demo paper is based on our recent work [9, 10]. For extensive references to the literature on the challenges and approaches for computing all relevant behaviors, we refer the readers to [10]. The efficient algorithm to compute PCG abstraction is based on a graph algorithm by Tarjan [11].

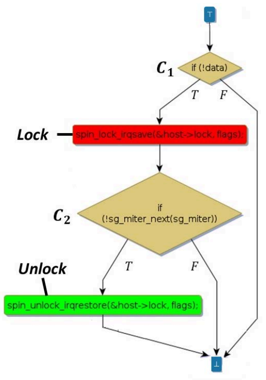

⊤
$C_1$
if (!data)
T
F
Lock
spin_lock_irqsave(&host->lock, flags);
$C_2$
if (!sg_miter_next(sg_miter))
Unlock
T
F
spin_unlock_irqrestore(&host->lock, flags);
⊥

**Figure 6: PCG for an unsafe `Lock`**

## 7 CONCLUSION

Analyzing each relevant behavior individually is important for accuracy, but that is often not done because of the efficiency and scalability hurdles posed. The PCG abstraction is useful to mitigate these hurdles when the the number of relevant behaviors is significantly smaller than the totality of behaviors, which is often true in practice. This is borne out by a Linux verification study [9]. The paper and the accompanying demonstration presents COMB, a tool based on the PCG abstraction. We are using COMB in DARPA Space/Time Analysis for Cybersecurity (STAC) project [6] to detect Algorithmic Complexity and Side Channel vulnerabilities.

## ACKNOWLEDGMENT

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. Dr. Kothari is the founding President of EnSoft.

## REFERENCES

[1] 2002. EnSoft Corp. http://www.ensoftcorp.com. (2002).
[2] Dirk Beyer. 2014. Status report on software verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 373–388.
[3] Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. 2016. Correctness witnesses: Exchanging verification results between verifiers. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 326–337.
[4] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. 2007. The software model checker Blast. *International Journal on Software Tools for Technology Transfer* 9, 5-6 (2007), 505–525.
[5] Stephen A Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*. ACM.
[6] DARPA. 2014. Space/Time Analysis for Cybersecurity. https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-14-60/listing.html. (2014).
[7] Richard A De Millo, Richard J Lipton, and Alan J Perlis. 1979. Social processes and proofs of theorems and programs. *Commun. ACM* 22, 5 (1979), 271–280.
[8] Tom Deering, Suresh Kothari, Jeremias Sauceda, and Jon Mathews. 2014. Atlas: A New Way to Explore Software, Build Analysis Tools. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA.
[9] Suresh Kothari, Payas Awadhutkar, Ahmed Tamrawi, and Jon Mathews. 2017. Modeling Lessons from Verifying Large Software Systems for Safety and Security. In *Proceedings of the 2017 Winter Simulation Conference*. to appear.
[10] Ahmed Tamrawi and Suresh Kothari. 2016. Projected Control Graph for Accurate and Efficient Analysis of Safety and Security Vulnerabilities. In *Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific*. IEEE, 113–120.
[11] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.
[12] Tao Wei, Jian Mao, Wei Zou, and Yu Chen. 2007. A new algorithm for identifying loops in decompilation. In *International Static Analysis Symposium*. Springer.