Proceedings of the 2017 Winter Simulation Conference
W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, eds.

MODELING LESSONS FROM VERIFYING LARGE SOFTWARE SYSTEMS FOR SAFETY AND SECURITY

Suresh Kothari
Payas Awadhutkar

Iowa State University
Ames, Iowa 50010, USA

Ahmed Tamrawi

Yarmouk University
Irbid, Jordan 21163

Jon Mathews

EnSoft Corp.
Ames, Iowa 50010, USA

ABSTRACT

Verifying software in mission-critical Cyber-Physical Systems (CPS) is an important but daunting task with challenges of accuracy and scalability. This paper discusses lessons learned from verifying properties of the Linux kernel. These lessons have raised questions about traditional verification approaches, and have led us to a model-based approach for software verification. These models are high-level models of the software, as opposed to the prevalent formal methods with low-level representations of software. We use models to gain insights into software verification challenges and use those insights to improve software verification. We demonstrate significant advantages of models with a Linux kernel study involving verification of $66,609$ `Lock` instances. We use models to: (a) analyze and find flaws in verification results from LDV, a top-rated Linux verification tool, (b) show significant improvement over LDV by improving accuracy, speed, and by verifying 99.3% instances compared to 65.7% instances by LDV.

1 INTRODUCTION

Software verification is especially important in the context of Cyber-Physical Systems (CPS) for critical applications, where failures have had catastrophic consequences from the Ariane 5 launch vehicle explosion (Flight 501 Inquiry Board 1996) to the power system outage that crippled most of the Northeast corridor of the U.S. and parts of Canada (Verton, D. 2013). In this paper, we present an empirical verification study of the Linux operating system kernel, which is fundamentally important to CPS. The Linux kernel alone, which provides the basis for so many devices (web servers, routers, smart phones, desktops), is over 12 MLOC. How can we verify this mountain of code?

Formal verification of large software has been an illusive target, plagued with the challenges of accuracy and scalability (Canal and Idani 2015, Stratis, P. 2014). The challenges of verifying software are daunting in part because of scale, but also due to approach. De Millo, Lipton and Perlis point out in their landmark 1979 paper (De Millo et al. 1979) the need for using high-level concepts (modeling) for scalable and human-comprehensible software verification.

In the next section, we present results from an empirical study using the Linux Driver Verification tool (LDV) (Zakharov, Mandrykin, Mutilin, Novikov, Petrenko, and Khoroshilov 2015) which has been the top-performing tool in the software verification competition (SVCOMP) (Beyer 2014). We use these results to motivate research questions about software verification.

In the subsequent sections, we present novel ideas for advancing model-based software verification, leaning on visionary papers (De Millo et al. 1979, Brooks Jr 1996) by Turing Award recipients. We consider

978-1-5386-3428-8/17/$31.00 ©2017 IEEE

verification of the 2-event matching property that abstracts the characterization of many software safety and security vulnerabilities. In this paper, we use `Lock` and `Unlock` paring as a 2-event property. Each instance involves verifying that a `Lock` is followed by `Unlock` on all feasible execution paths. The paired `Lock` and `Unlock` must be for the same shared object. A `Lock` instance is considered *safe* if the pairing is correct and *unsafe* otherwise. An unsafe instance is a vulnerability; it leads to *starvation* because a shared object remains locked, not available to other processes. The purpose of the verification is to determine for each `Lock` instance whether it is safe or not.

We present an empirical study using high-level models which we have developed based on our experiences verifying properties in the Linux kernel. The study involves an analysis of *verification instances* using software graphs as models to assess the verification complexity of each instance. These graphs are defined specifically to capture the information relevant to facilitate verification of any 2-event matching property. The study uses two types of models: (a) *macromodels* for inter-related modules related to a problem verification instance, (b) *micromodels* for relevant execution behaviors within each module. We have used the interactive visual query language of Atlas (EnSoft Corp. 2017, Deering et al. 2014) to dissect the instances and construct the models. The models for the $66,609$ instances in our empirical study are posted on a website (Tamrawi 2016).

We use the models to define metrics and present a multidimensional spectrum of complexities across the $66,609$ instances in our empirical study. Equipped with the knowledge of models, we present insightful representative examples of how LDV verification is unnecessarily complicated or sometimes inaccurate. Finally, we describe an automated model-based verification tool called $\mathscr{L}$-SAP which we have developed. $\mathscr{L}$-SAP is able to accurately verify $66,151$ (99.3% of the total) instances. Using $\mathscr{L}$-SAP we have discovered 7 unsafe instances which have been reported and fixed by the Linux developers.

The remainder of the paper is organized as follows. We first present our empirical study and the motivational research questions in Section 2. Next, Section 3 describes the software models we have developed for verifying the 2-event matching property. Section 4 presents examples from the Linux kernel showing the case of model-based verification and LDV limitation. Section 5 discusses our verification complexity metrics and presents results from lock/unlock verification on the Linux kernel. In Section 6, we describe $\mathscr{L}$-SAP, an automated model-based verification tool , which we have developed. Finally, we conclude in Section 7.

2 EMPIRICAL STUDY AND RESEARCH QUESTIONS

Our study of LDV includes three versions of the Linux operating system with altogether 37 million lines of code and $66,609$ verification instances. Running LDV on these Linux versions yields the result that $43,766$ (65.7)% of `Lock` instances are safe. LDV is inconclusive on $22,843$ instances, i.e. either LDV crashes or times out. LDV does not find any unsafe instances. LDV developers point out that it is challenging to trace the LDV verification to understand and address its failures (Zakharov et al. 2015).

We used LDV to verify three recent versions (3.17-rc1, 3.18-rc1 and 3.19-rc1) of the Linux kernel. We enabled all possible `x86` build configurations via `allmodconfig` flag. The results for verifying correct pairing of `Lock` and `Unlock` are reported in Table 1. Altogether the three Linux versions have 37 million lines of code and $66,609$ instances of locks. The results are reported in Table 1 as: $\mathscr{C}1$ *Category* of instances verified as *safe*, $\mathscr{C}2$ *Category* of instances verified as *unsafe*, and $\mathscr{C}3$ *Category* of the remaining instances where the verification is inconclusive. Column `Type` identifies the synchronization mechanism. Columns `Locks` and `Unlocks` show the number of lock/unlock instances of each type. Note that a lock may be paired with multiple unlocks on different execution paths.

2.1 Research Questions

LDV does not report any unsafe instances($\mathscr{C}2$ *Category*). Is it that the developers are no longer making mistakes or is it that the remaining unsafe instances are so complex that LDV cannot find them? The

Table 1: LDV Linux verification results.

Kernel	LOC	Type	Locks	Unlocks	LDV			
					$\mathscr{C}1$	$\mathscr{C}2$	$\mathscr{C}3$	Time
3.17-rc1	12.3 M	`spin`	14,180	16,817	8,962 (63.2%)	0	5,218	26h
		`mutex`	7,887	9,497	5,494 (69.7%)	0	2,393	27h
3.18-rc1	12.3 M	`spin`	14,265	16,917	9,152 (64.2%)	0	5,113	30h
		`mutex`	7,893	9,550	5,427 (68.8%)	0	2,466	30h
3.19-rc1	12.4 M	`spin`	14,393	17,026	9,204 (63.9%)	0	5,189	32h
		`mutex`	7,991	9,653	5,527 (69.2%)	0	2,464	29h
All Kernels			**66,609**	**79,460**	**43,766 (65.7%)**	**0**	**22,843**	**173h**

possibility of unsafe instances is worrisome for mission-critical CPS where an unsafe instance can lead to a system crash with dire consequences. This leads to the question:

> **Empirical Study Research Question 1:** Are there unsafe instances that LDV does not find because of its inconclusive verification?

LDV pronounces $43,766$ $(65.7)\%$ `Lock` instances to be safe ($\mathscr{C}1$ *Category*). Are all these instances really safe? The BLAST model checker (Beyer and Petrenko 2012) used in LDV uses the *CounterExample Guided Abstraction Refinement* (CEGAR) method for verification. CEGAR does not produce a proof or other evidence to support its assertion that an instance is *safe*. While LDV uses a model checker which is supposed to be formally correct, it can still have false negatives (e.g. due to incorrect transformation of software into the low-level representation that the model checker requires). This leads to the question:

> **Empirical Study Research Question 2:** Are there cases of erroneous LDV verification where an unsafe instance is verified as safe?

LDV pronounces $22,843$ $(34.3)\%$ `Lock` instances to be inconclusive ($\mathscr{C}3$ *Category*). What is so difficult about these instances that LDV cannot verify them?

> **Empirical Study Research Question 3:** Are there simple verification cases that become difficult because of the way LDV tries to verify them?

> **Overarching Research Question:** How can models help to decipher the complexity of each verification instance and use that knowledge to facilitate manual cross-checking or completion of verification?

In the next section, we present models which we have developed to help verify the Linux kernel, and which can help shed light on our research questions.

3 SOFTWARE MODELS FOR VERIFICATION

The software models are developed with two goals: (a) facilitate efficient and human-comprehensible verification, (b) enable analysis of complexity of each verification instance. The models are for verification of the 2-event matching property that abstracts the characterization of many software safety and security vulnerabilities.

3.1 Software Analysis Challenges to be Addressed

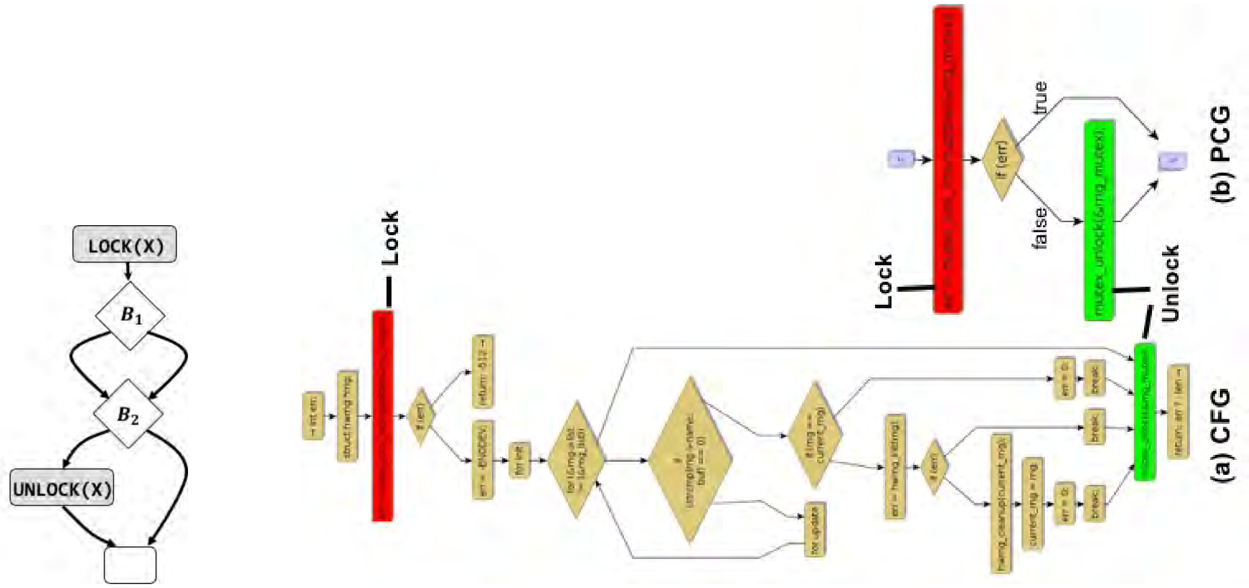
The following fundamental software analysis challenges must be addressed to verify complex instances of 2-event property. The proposed models are designed to address these challenges.

1. *Exponential Path Growth Challenge*: The number of paths is 2^n with n 2-way non-nested branch nodes. This exponential growth makes verification hard because every path must be considered for verifying correct matching.
2. *Path Feasibility Challenge*: A path on which the matching property fails does not always mean an unsafe instance because the conditions governing the path may be such that they all cannot be simultaneously true. Checking whether a set of boolean conditions can be simultaneously true is a hard problem requiring in the worst case 2^n computations for n boolean conditions.
3. *Inter-related Modules Challenge*: A large number of modules (functions) may have to be considered when the two events to be matched are in different functions.

3.2 Micro-model: Projected Control Graph (PCG)

The PCG model was developed as a part of doctoral research by Tamrawi (Tamrawi and Kothari 2016). The PCG provides an abstraction to address the first two challenges described above: *Exponential Path Growth* challenge, and the *Path Feasibility* challenge.

Let us clarify key ideas for the PCG using the control flow graph (CFG) in Figure 1a. Consider the problem of pairing `Lock` and `Unlock`. Suppose the CFG has a `Lock(X)` for object `X`, followed by a branch node `B1`, followed by another branch node `B2`, and each branch node has two paths. For `B2`, one path has `Unlock(X)`. The two `B1` paths do not have any statements relevant for the `Lock`/`Unlock` pairing. In this example, we have 4 paths. The 4 corresponding behaviors relevant to the pairing are: (a) `Lock(X)` followed by `Unlock(X)` on two paths, and (b) `Lock(X)` *not* followed by `Unlock(X)` on two other paths.



(a) Four CFG Paths with Only Two Distinct Behaviors.

(b) CFG and PCG for the function `hwrng_attr_current_store`.

Figure 1: Projected Control Graph (PCG) illustration.

The 4 paths can be partitioned into 2 equivalence classes corresponding to the 2 distinct behaviors. The PCG is obtained by transforming the CFG to retain only the distinct behaviors. In this example, it amounts to removing the `B1` branch node. The resulting PCG (not shown in the figure) has only the `B2` branch node with 2 paths corresponding to 2 distinct behaviors.

In this example, there is a path on which `Lock(X)` is not followed by `Unlock(X)`. However, it is not an unsafe `Lock(X)` if the path is *infeasible*. The path feasibility amounts to checking if the condition `B2` can be `true` (assuming `Unlock(X)` is on the `false` branch). Suppose `B2` is set to `false` before or after `Lock(X)`. Then the path is infeasible and it is a safe instance.

Unlike the above simple case involving just one condition, the Linux has several complex cases where the PCG model is a valuable abstraction for efficient and comprehensible verification. Figure 1b shows a Linux example of a CFG and its corresponding PCG. The CFG has 5 branch nodes resulting in 8 paths after the lock. Some of these paths go through a complex loop with two exits. 4 out of 5 branch nodes are irrelevant to the verification because all the paths branching from them lead to the `Unlock` and are thus equivalent. These 4 branch nodes get eliminated in the PCG and the 7 paths are represented by a single path in the PCG. Thus, the PCG simplifies the verification task by compacting the CFG.

The PCG also simplifies the path feasibility check. As seen from the PCG in Figure 1b, there is a path with missing `Unlock` and the feasibility of that path must be checked. If feasible, it is a bug. Otherwise the particular `Lock` is correctly paired. The PCG has retained only the condition that is necessary to verify the path feasibility. If the lock is granted, then the particular condition is `false`. So, the `true` path in Figure 1b is not feasible and thus it is safe instance.

3.3 Macro-model: Matching Pair Graph (MPG)

The MPG model was developed as a part of doctoral research by Gui (Gui and Kothari 2010). The MPG provides an abstraction to address the *Inter-related Modules* challenge. Let us make key observations about the MPG using an example `Lock` instance from Linux 3.19-rc1. The MPG is shown in Figure 2.

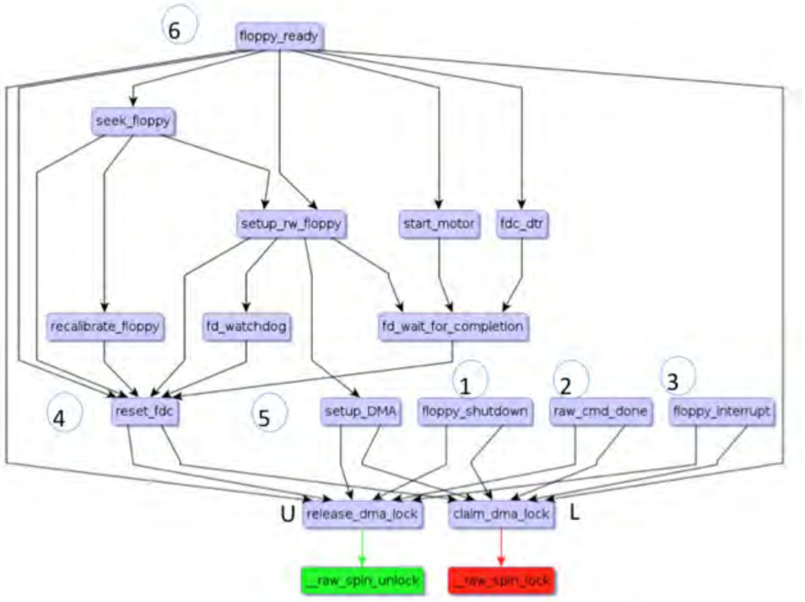


Figure 2: MPG with six calls to a `Lock` instance.

1. Each MPG is for a `Lock` instance which is a single call site for a `Lock` call. In this example, the call site is in the function `claim_dma_lock`, labeled `L`. Each call to `L` requires verification.
2. MPG includes all functions that call `L` directly. This MPG has 6 such functions numbered 1 to 6. Let us denote by D_L the set of direct callers to `L`.
3. MPG includes the corresponding `Unlock` instances. In this example, it is `release_dma_lock`, labeled `U`.
4. MPG includes all functions that call `U` directly. This MPG has 6 such functions numbered 1 to 6. Let us denote by D_U the set of direct callers to `U`.
5. MPG includes all functions in call chains originating from functions in D_L or D_U and reaching `L` or `U`. There are 7 such functions in this MPG.

6. If a function f in MPG is *balanced*, i.e., it directly or indirectly calls both L and U then f is a root unless there is a predecessor g of f that directly calls L or U. The functions numbered 1, 2, 3, and 6 are roots and it implies none of their predecessors call L or U directly. The functions numbered 4, and 5 are balanced but not roots. The function numbered 6 is a predecessor of 4 and 5 and it calls L so 6 is included.
7. A predecessor h of a balanced node is not included in the MPG if h is not a successor of a function that calls L or U directly.
8. Function f is always included in MPG if it is *unbalanced*, i.e., it directly or indirectly calls only L or U but not both.
9. A balanced root can have an unsafe Lock. Suppose a function f calls U followed L and f does not have a predecessor that calls L or U. Then f is a balanced root but it has an unsafe Lock.

It is a common Linux practice to have wrapper functions such as claim_dma_lock and release_dma_lock with logical names indicative of their functionality. The use of these functions, eliminates the need for any complex aliasing analysis to find Unlock corresponding to a given Lock instance.

4 MODEL-BASED VERIFICATION EXAMPLES

It can be shown that, except for a few corner cases, the MPG is necessary and sufficient for verification. Two major corner cases observed in our Linux study are: (a) The MPG is not complete because some functions are missed due to calls using function pointers, and (b) a condition that is necessary for a path feasibility check is set in a function not included in the MPG. We present representative examples show how we use MPG and PCG to dissect verification instances, one of case (a) that was verified as safe by LDV but it is actually unsafe.

4.1 Example 1: Models Simplify Verification

This is an example where LDV fails but the models reveal a simple verification method that avoids unnecessary complexity. The Figure 3a shows the MPG and Figure 3b shows the PCG for function numbered 1. The function nrs_policy_start_locked, labeled L, has the Lock instance for this MPG. The functions numbered 1 and 2 have calls to L. Each of these calls must be verified. The function nrs_policy_stop0, labeled U1 has the corresponding Unlock. The MPG shows that the typical practice in Linux is not strictly followed because functions 1 and 2 have direct calls to the _raw_spin_unlock, labeled U2. Under the typical practice, these calls would be to the wrapper function nrs_policy_stop0.

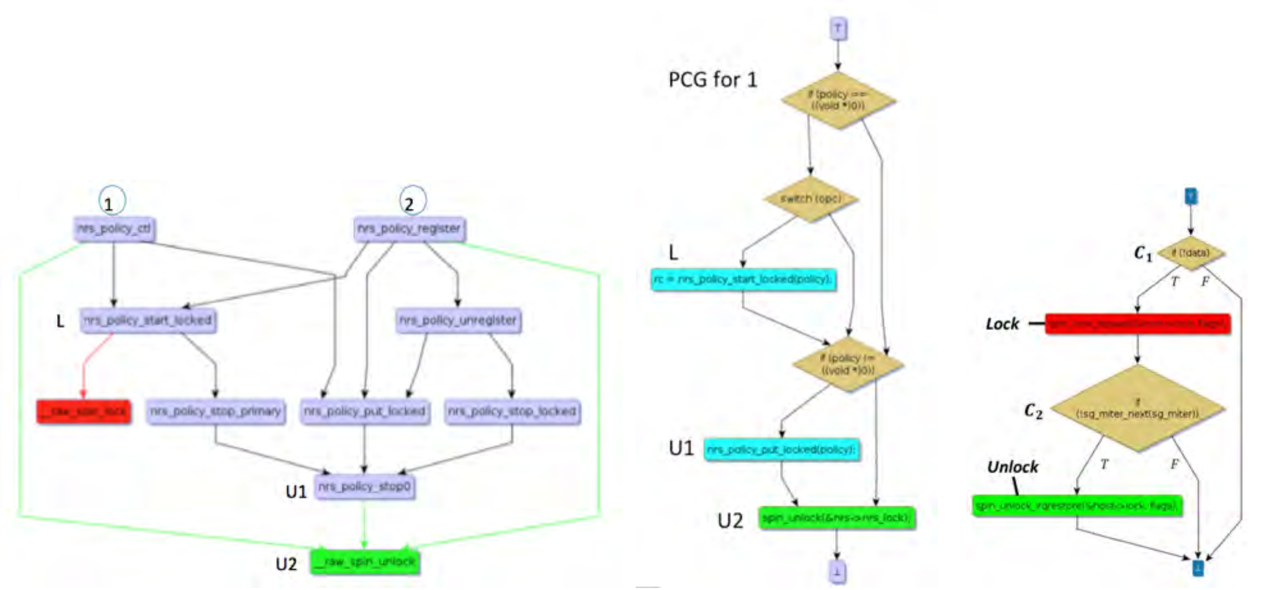
Note that the wrapper function L is balanced but it is not a root. The predecessor functions numbered 1 and 2 are included in the MPG because they have direct calls to U2.

We have not shown the PCG for the function L, but we note here a point about that PCG which is important for model-based verification. The PCG for L shows that U does not follow L on all paths. Thus, the function L by itself has an unsafe Lock instance.

The MPG shows that the function L is called by the functions numbered 1 and 2. The call to U2 in MPG hints that the verification could be completed easily if these callers 1 and 2 have Unlock on all paths. We need the PCG of these callers to proceed with this simple verification method.

Figure 3b, the PCG for the function numbered 1, shows that the unsafe Lock in L is no longer unsafe because of the call to U2 which protects all paths. The function numbered 2 has a similar PCG which completes the verification.

LDV runs out of time on this example. Our plausible explanation is based on observing a recurring LDV failure pattern: a function f has an unsafe Lock call that can be verified because a function g that calls f has Unlock on all paths following after Lock. Without the MPG, the LDV could be performing a computation intensive verification by examining all predecessors of f. Another possibility, LDV could be performing a path feasibility check on paths in f that do not have Unlock operations. We observed that



(a) Example-1: MPG.

(b) Example-1: PCG.

(c) Example-2: Unsafe `Lock` missed by LDV.

Figure 3: Examples 1 and 2 to Illustrate Simple Verification.

the *reverse call graph* of functions `f` with unsafe `Lock` are often very huge with hundreds or thousands of functions and without the help of MPG the state space for verification is extremely large.

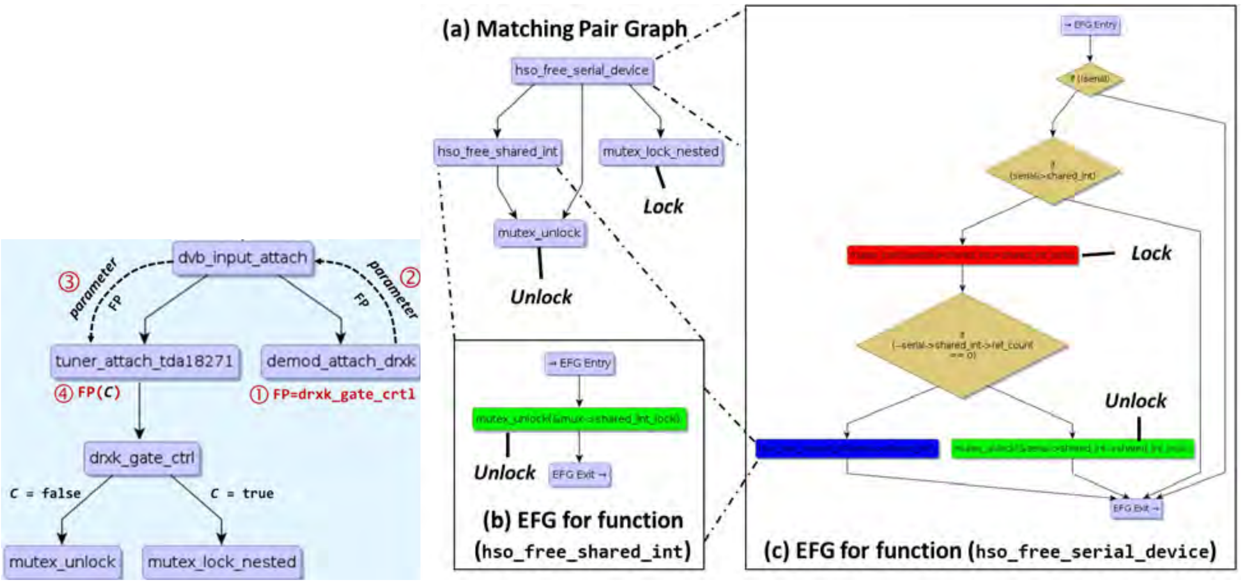
4.2 Example 2: Unsafe Lock Missed by LDV

This is another Linux example for which LDV cannot complete the verification, but for which models reveal a simple verification method. This instance is a bug, and requires a path feasibility check within a single function to detect. However, the CFG for the function is relatively complicated, with 8 branch nodes and multiple loops. The CFG can be viewed at the website (Tamrawi 2016). We suspect that LDV cannot complete this example because it cannot simplify the path feasibility check. Using the PCG, we can at once simplify the CFG and detect the bug. Figure 3c shows the PCG for the function `toshsd_thread_irq` that has calls to `Lock` and `Unlock`. The multiple CFG paths between `Lock` and `Unlock` are all equivalent and are mapped to one PCG path. 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 condition expression $(C_1\overline{C_2})$ is true. The feasibility check has become easy due to the PCG. The path is feasible and thus it is an unsafe `Lock`. This bug was reported to the Linux organization and was fixed.

4.3 Example 3: Unsafe Lock Declared Safe by LDV

This particular instance attracted our attention because of a peculiarity the PCG exhibited. The PCG shows that the `Lock` and `Unlock` are on disjoint paths in the function `drxk_gate_crtl` (`f1`) and if $C =$ `true`, the `Lock` occurs, otherwise, the `Unlock` occurs. We hypothesized that the `Unlock` and `Unlock` can match if `f1` is called twice, first with $C =$ `true` and then with $C =$ `false`. A quick query using Atlas shows that `f1` is not called directly anywhere. Thus, it is either dead code or `f1` is called using a function pointer.

Resolving the function pointers using Atlas (Kothari et al. 2016), we find the situation shown in Figure 4a. The function `tuner_attach_tda18271` (`f2`) calls the function `f1` via function pointer. `demo_attach_drxk` sets the function pointer to `f1`, the pointer is communicated by parameter passing to `dvb_input_attach`, then to `f2`. Recall that `f1` must be called twice. The function `f2` has a path on which there is a return before the second call to `f1` and thus it is a bug.



(a) Example-3: `drxk_gate_crtl` after resolving calls via function pointers.

(b) Example-4: Using Models for Manual Cross-check.

Figure 4: Examples 3 and 4.

Without access to its proof, it is not possible to determine what goes wrong with LDV when it incorrectly verifies this peculiar instance as safe.

4.4 Example 4: Importance of Models for Manual Cross-check

We present this example to underscore that the models have been designed to aid in manual cross-checking. Figure 4b(a) shows the functions that must be examined for the lock in the function `hso_free_serial_device`. Figures 4b(b) and 4b(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 branch node with two paths: (1) one path leads to a matching unlock (intra-procedural), and (2) the other path leads to a call to function `hso_free_shared_int` (blue-colored node). The PCG of the called function `hso_free_shared_int` shows a matching unlock on all paths within that function.

5 VERIFICATION COMPLEXITY METRICS

We studied a subset of Linux kernel instances to understand the distribution of complexity in the Linux kernel. We selected 14,729 instances from Linux kernel 3.19-rc1 for this study. We use the following metrics to evaluate the complexities.

- RCG to MPG reduction: MPG simplifies verification by retaining only a subgraph of the RCG. A function `f` which calls `Lock` or `Unlock` can be called by several other functions. They may or may not be required for analysis of the 2-event problem and can create unnecessary complication. How significant is this graph reduction? We report average and maximum size (nodes and edges) of RCG and MPG of functions in selected instances.
- CFG to PCG reduction: PCG simplifies verification by retaining only the distinct relevant behaviors from the CFG. How significant is the reduction from the CFG to PCG transformation? We analyze the size distribution of CFG and PCG to explore this question.

The second objective of this study is to gain further insights into the kinds of difficulties that LDV faces. Out of the selected instances, LDV declares 8320 instances as safe (referred as LDV-safe instances) and is inconclusive on remaining 6409 instances (referred as LDV-inconclusive instances). The aim is to understand what makes LDV-inconclusive instances hard, and how they differ from the LDV-safe instances.

5.1 RCG and MPG

If a function `f` calls `Lock` but does not call `Unlock`, then the predecessors of `f` need to be analyzed. In general, all predecessors, i.e. RCG of `f` may have to be analyzed. RCG in the Linux kernel tend to be huge. MPG is a model designed to include a subgraph of RCG which is sufficient in all but a few corner cases. We compare MPG with RCG.

Table 2a shows a comparison of RCG and MPG. In the selected instances, average RCG of a function has 33.37 nodes and 39.40 edges. In contrast, the average MPG has 3.29 nodes and 2.49 edges. This implies that a full RCG is generally not needed for analysis. The reason many MPGs have fewer edges than nodes is the prevalence of the simplest possible MPG, having 3 nodes and 2 edges: a function `f`, which calls `Lock` and `Unlock` directly i.e. 3 nodes `f`, `Lock`, `Unlock` and 2 edges `f`$\rightarrow$`Lock` and `f`$\rightarrow$`Unlock`. This is reflected in Figure 5.

Table 2: Comparison between RCG and MPG.

(a) Distribution of sizes of RCG and MPG.

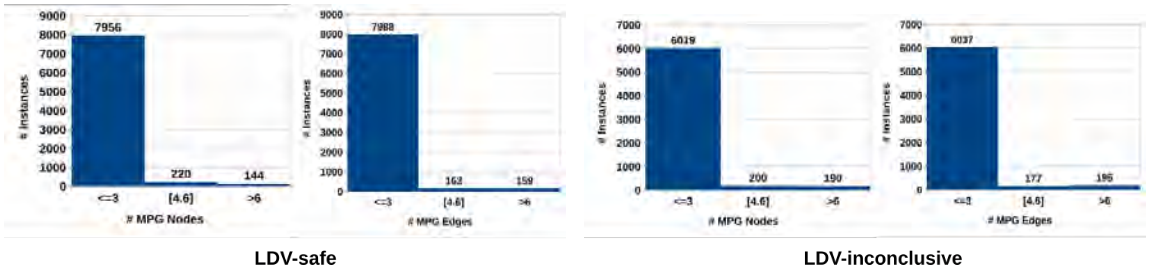
Graph	Artifact	Average	Max
RCG	Nodes	33.37	130028
	Edges	39.40	272146
MPG	Nodes	3.29	89
	Edges	2.49	205

(b) Instances with significant RCG to MPG reduction.

Function	RCG		MPG	
	Nodes	Edges	Nodes	Edges
rpm_resume	20692	28339	3	2
mmc_host_clk_hold	651	1151	3	2
core_scsi3_ua_allocate	184	231	3	2

It also gives an indication of how large an instance can be. A particular function has RCG with 130028 nodes and 272146 edges. If this function is relevant to an instance, it will make analysis of that instance very hard. In contrast, the largest MPG in selected instances has 89 nodes and 205 edges. Table 2b shows some interesting instances which have significantly smaller MPG than the RCG of the function.

Figure 5 shows the distribution of MPGs in LDV-safe and LDV-inconclusive instances. It reveals that majority of the instances have the simplest possible MPG described in previous paragraph. There are 7956 LDV-safe and 6019 LDV-inconclusive instances with this type of MPG. It corresponds to 95.6% of LDV-safe and 93.92% of LDV-inconclusive instances. This implies that even though the MPG of an instance can be simple, LDV may still be unable to finish its analysis and time out.



LDV-safe

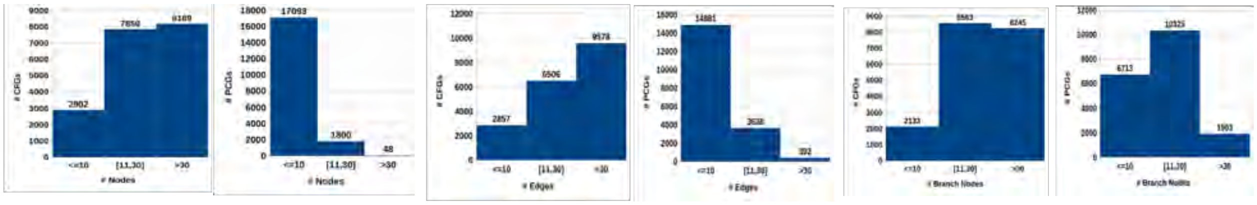
LDV-inconclusive

Figure 5: Comparison of MPGs in LDV-safe and LDV-inconclusive instances.

5.2 CFG to PCG reduction

Figure 6 shows how CFG and PCG of relevant functions in selected instances vary in size. In particular, we present distribution of nodes, edges, and branch nodes in both CFGs and PCGs. Considering a graph with $\leq$ 10 nodes as simple to analyze, there are 17093 simple PCGs compared to 2902 CFGs. Similarly, considering the metric for a simple graph as $\leq$10 edges, there are 14881 simple PCGs compared to 2857 CFGs. Considering the graph with >30 nodes as complex to analyze, there are 8189 complex

CFGs compared to 48 PCGs. These numbers indicate a significant amount of reduction in analysis work. Furthermore, compared to 2133 CFGs with no branch nodes there are 6713 with no branch nodes. Absence of branch nodes implies that path feasibility check is not required. Which means, using PCG we eliminate the need for path feasibility check in 3 times more cases. Branch Nodes cause exponential growth of paths, which makes verification hard. Considering a graph with more than 5 branch nodes as complex, there are 8245 complex CFGs compared to 1903 complex PCGs. It implies that CFG is 4 times more likely to be a complex graph than PCG.



(a) Nodes.

(b) Edges.

(c) Branch Nodes.

Figure 6: CFG to PCG reduction.

6 MODEL-BASED VERIFICATION (MBV) AUTOMATION

We describe a MBV method based on the MPG and PCG models. We present results obtained with $\mathscr{L}$-SAP, a MBV tool we have developed.

6.1 MBV Method

Our automated MBV method works as follows:

1. *Function Summary:* The distinct behaviors of a function in the MPG are summarized by a tuple. The summary tuple (`-`,`L`,`LL`,`LU`,`U`,`UL`) for a function `f` implies that the set of control flow paths can be divided into 6 equivalence classes corresponding to the 6 tuple elements. Each equivalence class is a set of paths that produce the behavior summarized by one tuple element.
2. *Behavior Equivalence Rules:* The behavior along a path is represented by a *regular expression* consisting of symbols `L` and `U` corresponding to calls to `Lock` and `Unlock`. We have the following equivalence rules: (i) the expression $(\texttt{LU})^{+}$ is equivalent to `LU`, (ii) $(\texttt{UL})^{+}$ equivalent to `UL`, (iii) `LUL` equivalent to `L`, (iv) `ULU` and `UU` both equivalent to `U`, and (v) `LLU`, `ULL`, and $\texttt{L}(\texttt{LL})^{+}$ all equivalent to `LL`. These equivalence rules are designed to keep track of minimal summary sufficient for performing the verification. Note that the summary `LL` presents a deadlock situation and it is not equivalent to `L`. The equivalence rules imply that a summary tuple can have at most 6 elements.
3. *Applying Function Summary:* The summary of function `f` is applied at each of its callsites to calculate the summary of the caller function `g`. For a path in `g` with `L` followed by a call to `f`, the summary (`-`,`L`,`LL`,`LU`,`U`,`UL`) of `f` transforms to (`L`,`LL`,`LU`). For a path in `g` with a call to `f` followed by `U`, the summary of `f` transforms to (`U`,`LU`,`LL`). In transforming the summary, we maintain only the distinct tuple elements. The summary of `g`, is obtained by aggregating the summary tuple elements on all control paths. Suppose `g` only has the two paths as discussed, then the summary of `g` is (`L`,`LL`,`LU`,`U`). A function has at least one path with behavior corresponding to each each tuple element of its summary. Ordering the tuple elements is not important, we only need to know all the distinct elements.
4. *Loops and Recursion Summary:* In presence of loops and recursion, function summaries are computed as fixed points using the behavior equivalence rules. For example, a loop with `L` followed by `U` is summarized as `LU`.
5. *Summary Computation Using MPG and PCG:* The summary is computed bottom-up starting with leaf nodes in the MPG. The summary for each function is computed using its PCG.

6. *Verification Verdict:* A vulnerability is reported the first time `LL` is detected. Note that `LL` does not change after its first appearance. A vulnerability is reported if `L` or `UL` are elements of the function summary of a MPG root. If these situations are not encountered, the verification reports that the `Lock` instance is *safe*.

6.2 MBV Proof Examples

We illustrate the use of our MBV method to prove verification for two `Lock` instances for which the MPGs are shown in Figures 2 and 3a. We do not show the PCGs used to compute the summaries for functions belonging to these MPG. The MPGs, PCGs and CFGs are posted on the website (Tamrawi 2016).

Verification Proof Example 1: Let us prove that the `Lock` instance is safe for the MPG shown in Figure 2. The proof steps are as follows.

1. Using the respective PCGs, the summary is (`LU`) for the functions numbered 1 to 5.
2. Besides the roots, there are 7 functions that call the functions numbered 1 to 5. Since these 7 functions do not call `L` or `U` directly, all of them have the same summary as the functions they call, i.e., their summary is also (`LU`).
3. The proof is complete if we verify the root function numbered 6. The PCG for the root function shows that its direct `L` call is followed by its direct `U` call. The calls to other MPG functions are not in between the direct `L` call and the direct `U` call and thus the root function summary is also `LU`. This completes the proof.

Note: If the root function were to have a path with: a direct call to `L`, call to `f` with summary `LU`, followed by a direct call to `U`, then the vulnerable behavior `LL` would occur.

Verification Proof Example 2: Let us prove that the `Lock` instance is safe for the MPG shown in Figure 3a. The proof steps are as follows.

1. The MPG in Figure 3a shows that 4 functions call only `U`. Each of them has the summary `U`.
2. Using its PCG, we can show that the summary of the function labeled `L` is (`LU`, `L`).
3. Using their PCGs, it is observed that the root functions numbered 1 and 2 have call to `Unlock` at the end on all paths. These functions call the function labeled `L`. The summary of the root functions is thus (`LU`) on all paths. This completes the proof.

7 CONCLUSION

The National Science Foundation (NSF) defines Cyber-physical systems (CPS) as ”engineered systems that are built from, and depend upon, the seamless integration of computational algorithms and physical components.” It is important to enable efficient development of high-confidence distributed CPSs whose nodes operate in a provably correct manner in terms of functionality and timing (synchronicity between physical and software components), leading to predictable and reliable behavior of the entire system. To this end, this paper presents an efficient and accurate approach to analyze software with millions of lines of code for a broad spectrum of safety and security vulnerabilities. Using the model-based proof method in this paper, it is possible to manually cross-check the verification. In a course taught by one of the authors, undergraduate students perform such verification using the website (Kothari et al. 2016).

ACKNOWLEDGMENTS

We thank our colleagues from Iowa State University and EnSoft for their help with this paper. Dr. Kothari is the founder President and a financial stakeholder in EnSoft. This material is based on research sponsored by DARPA under agreement number FA8750-15-2-0080. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

REFERENCES

Beyer, D. 2014. "Status Report on Software Verification". In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer.

Beyer, D., and A. K. Petrenko. 2012. "Linux Driver Verification". In *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, 1–6. Springer.

Flight 501 Inquiry Board 1996. "ARIANE 5 Failure - Full Report". http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html.

Brooks Jr, F. P. 1996. "The Computer Scientist as Toolsmith II". *Communications of the ACM* 39 (3).

Canal, C., and A. Idani. 2015. *Software Engineering and Formal Methods: SEFM 2014 Collocated Workshops: HOFM, SAFOME, OpenCert, MoKMaSD, WS-FMDS, France*, Volume 8938. Springer.

De Millo, R. A., R. J. Lipton, and A. J. Perlis. 1979. "Social Processes and Proofs of Theorems and Programs". *Communications of the ACM* 22 (5): 271–280.

Deering, T., S. Kothari, J. Sauceda, and J. Mathews. 2014. "Atlas: A New Way to Explore Software, Build Analysis Tools". In *Companion Proceedings of the 36th International Conference on Software Engineering*, 588–591. ACM.

EnSoft Corp. 2017. "EnSoft Corp.". http://www.ensoftcorp.com.

Gui, K., and S. Kothari. 2010. "A 2-Phase Method for Validation of Matching Pair Property with Case Studies of Operating Systems". In *2010 IEEE 21st International Symposium on Software Reliability Engineering*.

Kothari, S., A. Tamrawi, and J. Mathews. 2016. "Human-Machine Resolution of Invisible Control Flow". In *Proceedings of the twenty-fourth IEEE International Conference on Program Comprehension*. IEEE.

Kothari, S., A. Tamrawi, J. Sauceda, and J. Mathews. 2016. "Let's Verify Linux: Accelerated Learning of Analytical Reasoning Through Automation and Collaboration". In *Proceedings of the 38th International Conference on Software Engineering Companion*, 394–403. ACM.

Stratis, P. 2014. "Formal Verification in Large-Scaled Software: Worth to Ponder". https://blog.inf.ed.ac.uk/sapm/2014/02/20/formal-verification-in-large-scaled-software-worth-to-ponder/.

Tamrawi, A. 2016. "Linux Kernel Verification Results". http://kcsl.ece.iastate.edu/linux-results/.

Tamrawi, A., and S. Kothari. 2016. "Projected Control Graph for Accurate and Efficient Analysis of Safety and Security Vulnerabilities". In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*.

Verton, D. 2013. "Software Failure Cited in August Blackout Investigation". http://www.computerworld.com/article/2573466/disaster-recovery/software-failure-cited-in-august-blackout-investigation.html.

Zakharov, I. S., M. U. Mandrykin, V. S. Mutilin, E. Novikov, A. K. Petrenko, and A. V. Khoroshilov. 2015. "Configurable Toolset for Static Verification of Operating Systems Kernel Modules". *Programming and Computer Software* 41 (1): 49–64.

AUTHOR BIOGRAPHIES

SURESH KOTHARI is the Richardson Endowed Professorship in the Electrical and Computer Engineering (ECE) Department at Iowa State University (ISU) and he is the founding President of EnSoft. He has pioneered graph-based software analysis and verification technology. Dr. Kothari has served as the PI for the DARPA APAC program and currently is the PI for the DARPA STAC program, and as a Co-PI on DARPA SEC program. He has been an ACM Distinguished Lecturer. His email address is kothari@iastate.edu.

PAYAS AWADHUTKAR is a Ph.D. student in the ECE department at ISU. His research focuses on software analysis and its applications. His email address is payas@iastate.edu.

AHMED TAMRAWI is an assistant professor in the Computer Engineering Department at Yarmouk University in Jordan. He did his Ph.D. in the ECE department at ISU. His email address is atamrawi@yu.edu.jo.

JON MATHEWS works at EnSoft; he is the Architect of Atlas, the graph database platform for analyzing large software. His email address is jmathews@ensoftcorp.com.