Insights for Practicing Engineers from a Formal Verification Study of the Linux Kernel*

Suresh Kothari, Payas Awadhutkar, and Ahmed Tamrawi
Electrical and Computer Engineering Department
Iowa State University
Ames, Iowa 50010
Email: {kothari, payas, atamrawi}@iastate.edu

***Abstract*—Formal verification of large software has been an illusive target, riddled with the problem of scalability [1]–[4]. Even if the obstacle of scale may be cleared, major challenges remain to adopt formal verification in practice.**

This paper presents an empirical study using a top-rated formal verification tool for Linux [5], and draws insights from the study to discuss the intrinsic challenges for adopting formal verification in practice. We discuss challenges that focus on practical needs: (a) facilitate crosschecking of verification results, (b) enable the use of formal verification for certification, and (c) integrate formal methods in a development environment. Leaning on visionary papers [6], [7] by Turing Award recipients, we present novel ideas for advancing formal verification in new directions that would help practicing engineers.

I. Introduction

Formal verification has been the holy grail of software engineering research [8]. Automated software verification methods have led to advances in data and control flow analyses, and applications of techniques such as Binary Decision Diagrams (BDDs) to analyze large software [9]. However, there are two fundamental limitations: (1) a completely automated and accurate analysis encounters NP hard problems [10]–[12], and (2) formal verification methods work as automated black boxes with very little support for cross-checking [5], [13], [14]. This paper is about the second limitation. It elaborates on the practical difficulties emanating from the limitation and how it may be possible to address the limitation.

The second limitation leads to issues that leave several practical needs unaddressed. Although the formal verification works as an automated black box, it requires an inordinate amount of preprocessing effort, involving a transformation from the software to the formal specification that can be checked automatically using a model checker or a SAT solver. This transformation is not automatic, it requires domain knowledge of the particular formal method and a lot of cumbersome human effort.

Besides the preprocessing, another serious issue is the lack of supporting evidence to be able to understand and use the results of formal verification. Without the evidence, it is not possible to use formal verification as a certification apparatus, or to integrate formal methods in a development environment. As we shall exemplify, it is quite hard for the user to know that the verification result is wrong without supporting evidence. We will present an empirical study to elaborate the notion of evidence and its importance in practice.

We did an empirical study to get deep insights into some of the state of the art formal verification methods. We use the empirical study results to motivate the specific needs for generating supporting evidence as a part of formal verification. We used the Linux Driver Verification tool (LDV) [5] which has been the top Linux device driver verification tool in the software verification competition (SVCOMP) [15]. The LDV's developers were generous to help us with the study. The study includes three recent versions of the Linux operating system with altogether 37 MLOC and $66,609$ verification instances. Each instance involves verifying that a `Lock` is followed by `Unlock` on all feasible execution paths. Running LDV on these Linux versions yields the result that pairing is correct for $43,766$ $(65.7)\%$ of `Lock` instances. LDV is inconclusive on $22,843$ instances, i.e. either the tool crashes or it times out. LDV does not find any instance with incorrect pairing. LDV does not provide evidence to support its results except for the instances where the verification reveals a bug.

We dissected some of the instances to gain insights into how hard it is to understand the verification and what evidence could facilitate such understanding. We used the interactive visual query language of Atlas [16], [17] to dissect the instances. The Atlas query language makes it easy to build software analysis, comprehension, validation and transformation tools. The dissected instances point to specific needs for evidence. One of the examples is actually a serious bug that was missed by the formal verification tool.

Our paper is inspired by the following question: "What advances in formal methods would it take to meet the practical needs?" Supported by examples from our study, we will discuss two key ideas to answer this overarching question. These ideas lean on visionary papers [6], [7] by Turing Award recipients.

Overall, this paper makes the following key contributions:

1) A formal verification study of the Linux kernel to show supporting evidence as a critical need for practical adoption of formal methods (Section II).
2) Two key ideas for new directions for advancing formal verification methods to meet the practical needs (Section III).

*This material is based on research sponsored by DARPA under agreement numbers FA8750-12-2-0126 and FA8750-15-2-0080. 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.

II. A Motivational Verification Study

We undertook a verification case study to get deep insights into the state of the art for formal verification. We use results of the study to motivate the need for supporting evidence and its use to address practical needs such as certification and integration of formal methods in a development environment. We did the study using the Linux Driver Verification tool (LDV) [5] which has been the top Linux device driver verification tool in the software verification competition (SVCOMP) [15]. The LDV's developers were generous to help us with the study.

We chose an extensively studied 2-event verification problem. Given two events `E1(X)` and `E2(X)` operating on an object `X`, the goal is to verify that `E1(X)` is followed by `E2(X)` on every feasible execution path. We studied two 2-event problems: one is to verify correct pairing of `allocation` and `deallocation` to avoid memory leaks and the other is to verify correct pairing of `lock` and `unlock` to avoid unsafe synchronization. We used LDV to verify three recent versions (3.17-rc1, 3.18-rc1 and 3.19-rc1) of the Linux kernel along with the device drivers. We enabled all possible `x86` build configurations via `allmodconfig` flag. The results for verifying correct pairing of `lock` and `unlock` are reported in Table I

TABLE I. LDV Linux Verification Results

Kernel	LOC	Type	Locks	Unlocks	LDV			
					$\mathcal{C}1$	$\mathcal{C}2$	$\mathcal{C}3$	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**

Altogether the three Linux versions have 37 MLOC and $66,609$ instances of locks. Table I gives the results for: (i) $\mathcal{C}1$: the automatically verified instances with no violation (correct pairing), (ii) $\mathcal{C}2$: the automatically verified instances with one or more violations (incorrect pairing) shown by missing unlock(s) on feasible path(s), and (iii) $\mathcal{C}3$: the remaining instances where the verification is inconclusive (inconclusive pairing). Column `Type` identifies the synchronization mechanism. Columns `Locks` and `Unlocks` show the number of lock/unlock instances. Note that a lock may be paired with multiple unlocks on different execution paths.

A. Discussion of Results

Note that the LDV tool verifies the pairing of $43,766$ $(65.7)\%$ `Lock` instances. It either times out or crashes on $22,843$ instances. This study points to the need for accompanying evidence. The evidence is critically important to crosscheck formal verification results.

- LDV pronounces that pairing is correct for $43,766$ $(65.7)\%$ `Lock` instances. However, LDV provides no proof that humans can crosscheck. LDV transforms the source code to a representation that is suitable for its verification engine. That representation is not suitable for for humans to crosscheck. Without human-verifiable evidence, it is either trust the verification results blindly or crosscheck them manually without any help from the verification apparatus. As a result, the formal verification is not amenable for use as a certification apparatus.

- LDV does not find any violations. Maybe, the Linux developers have learnt not to make the mistakes that LDV can find. However, it remains open: is LDV mistakenly pronouncing some instances as safe and we simply do not know it?

- The instances are inconclusive if LDV cannot complete the verification. LDV is inconclusive on $22,843$ instances. Looking across the inconclusive instances for the three versions in our study, LDV seems to have reached a saturation point. What is so difficult about these instances that LDV is not able to complete verification? This brings up another reason why human-comprehensible evidence is important. Even when the verification is incomplete, the evidence can be valuable to developers to explore further and it can also be useful to improve the verification tools.

Our dissection of some of the `Lock` instances have revealed bugs that LDV missed. The Linux organization has accepted these bugs and corrected them. We have described one such bug in Section IV.

III. Evidence-Equipped Verification

The evidence is important to address practical needs such as: (a) facilitate crosschecking verification results, (b) enable the use of formal verification for certification, and (c) integrate formal methods in a development environment. This section focuses on what should be the evidence that facilitates human comprehension of the verification to address practical needs.

We propose visual models as verification-critical evidence. These models can serve many practical needs. For example, they can empower interactive analytical reasoning by developers, or they can serve as easy-to-understand documentation for certification. Specifically, we will discuss two types of models: (a) a model for interprocedural verification, and (b) a model for intra-procedural verification.

We show representative examples of Linux verification instances to bring out the benefits of having such models. We show how these models can enable the user to construct modular proofs. We describe an evidence-based modular proof for a verification instance which could not be verified by LDV. The evidence that enables the user to construct proofs without too much effort is especially important because the formal proof is at a low level; it is suitable for the machine but not amenable to human understanding.

In Section IV, we will use the evidence for a complex verification instance to show that the formal verification by LDV is incorrect for that instance.

The interprocedural model presented in this paper is based on our research [18]–[20]. This model is specifically in the context of verification for 2-event matching problems. There is a need to generalize the interprocedural model for broader

applicability. The intra-procedural model is based on our research [21]. The intra-procedural model is not restricted to the 2-event matching problems. Our models have direct correspondence to the source code and thus it is easy for the user to check if the models have any errors before using them to crosscheck the results of formal verification. Thus, the models are easy for the human to understand, and it is also easy for the human to use them to construct modular proofs. The models for all the $66,609$ `Lock` instances are available at [22]. We use these models to teach software verification to undergraduates [20].

We will discuss specific needs for the formal verification tools to be advanced to produce visual models as evidence. A formal verification tool must compute some approximation of these model as the models are intrinsically important for verification. However, the computation remains implicit and the models are not exposed to the user. We present a case to argue that the formal verification tools should make these models explicit and open them to the user.

A. Interprocedural Visual Model for Evidence

The complexity of software is rooted in its own version of the *butterfly effect* [23], [24]. A small change at one point can impact many parts of the software and cause an unforeseen effect at a very distant point of the software. This impact propagation is hard to decipher from software viewed as lines of code; it makes program comprehension and reasoning tedious, error-prone, and almost impossible to scale to large software.

Abstractly, there exists a set $S(X)$ of methods that captures the exact expanse of the software butterfly effect. The set $S(X)$ is necessary and sufficient to verify an instance X. An exact computation of $S(X)$ is not possible because of inexact pointer analysis and other computationally expensive or intractable program analysis problems. During verification, a verifier must compute $\hat{S}(X)$, its version of $S(X)$, for performing the verification. If $\hat{S}(X)$ is smaller than $S(X)$ then the verification is incomplete. If $\hat{S}(X)$ is bigger than $S(X)$ then the verification is unnecessarily complicated.

The LDV does not reveal to the user $\hat{S}(X)$. We have computed our version of set $S(X)$ for each `Lock` instance X. Our computation, based on our research [18]–[20], is specifically in the context of 2-event matching problems such as the pairing of `Lock` and `Unlock` or the pairing of memory `Allocation` and `Deallocation`. We refer to it as the Matching Pair Graph (MPG). As a general notion, we refer to $S(X)$ as the Interprocedural Verification Graph (IVG). The IVG includes the *call* edges between the methods in IVG.

An Example of IVG for a Linux Verification Instance: Now, let us discuss an example of how the IVG can be valuable evidence. Figure 1 shows the IVG for the `Lock` call in function `clk_enable_lock`. This IVG is actually for two `Lock` instances corresponding to the calls `raw_spin_lock` and `raw_spin_trylock`. A single call to `raw_spin_unlock` in `clk_enable_unlock` is used to pair with both the `Lock` instances. The IVG implies that verifying the correct pairing for this example can be narrowed down to the 10 methods shown in the IVG (not counting the `Lock` and `Unlock` methods).

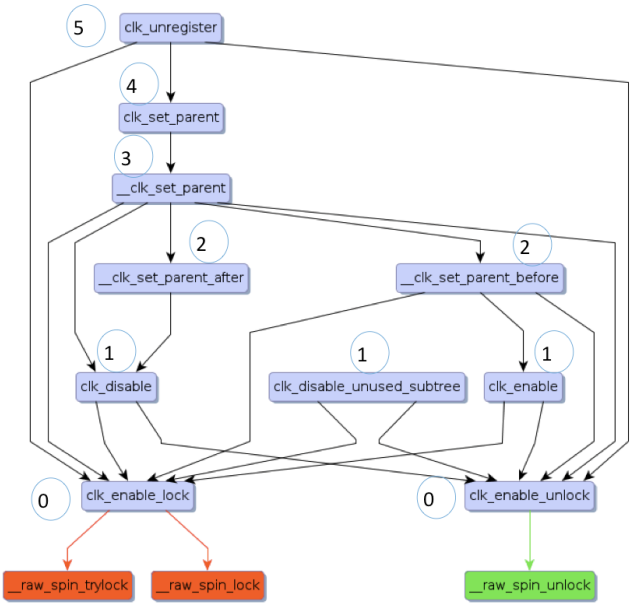

Fig. 1. IVG for the lock in `clk_enable_lock`

Practical Benefits of IVG: Let us point out the major practical benefits of IVG.

1) By going through an IVG produced by a formal verification tool, the user could easily crosscheck if all the relevant functions are correctly covered by the verification. Suppose inside a function `f` in the IVG there is a path without `Unlock` and on that path a `Lock` pointer is passed to another function `g`. Then, `g` must be in the IVG. If not, the user can deduce that there is a potential verification error because the IVG is incomplete.
2) The user can complete verification building on the evidence from IVG. Continuing the above example, suppose `g` is called using a *function pointer*. The formal verification tool could be inconclusive because it cannot handle calls through function pointers. As described in [19], the user can chase the function pointer with the help of an interactive tool and complete the verification. We will revisit this point in Section IV where we present a bug missed by the formal methods tool LDV.
3) The IVG reveals possibilities for human-centric proofs for verification. The formal methods proofs are machine-centric, they use techniques such as constructing a boolean formula and then checking its satisfiability. Such proofs are not amenable to human comprehension. As a contrast, let us consider a human-centric proof facilitated by the example IVG shown in Figure 1.

An Example of Human-centric Proof Enabled by IVG: The proof is for the verification instance in the above example. Figure 1 shows the IVG for the `Lock` call in function `clk_enable_lock`. The key idea for the proof is as follows. The proof proceeds step-wise based on the structure of the IVG. This example requires 5 steps and the number of nodes to be verified at each step are respectively 3, 2, 1, 1, 1 as shown in the Figure 1. Going through the *control flow graph* (CFG) of each of the 3 nodes in step 1, one must verify the correct pairing of `Lock` and `Unlock` on all feasible control flow

paths. The nodes in step 2 call the nodes in step 1, and thus they have *indirect* calls to `Lock` and `Unlock`. The verification in step 1 implies that these calls are verified and we only have to verify the correct pairing of *direct* calls to `Lock` and `Unlock` from the nodes in step 2. The proof proceeds inductively from step i to step $i+1$ until the final step 5. The step-wise inductive proof is much simple for the human to comprehend and thus the IVG provides an alternative of human-friendly proof.

We will later discuss a compact representation of CFG called the Event Flow Graph (EFG). We will discuss how the EFG is used at each of these 5 steps of the proof.

The nodes numbered 0 are the *wrapper* functions that simply call `Lock` and `Unlock` individually. It is a common practice in the Linux kernel to use such wrappers and name them appropriately to indicate the purpose for which the locks are introduced.

B. Intra-procedural Visual Model for Evidence

This model provides evidence for intra-procedural verification. The evidence helps the user with two formidable barriers that make intra-procedural verification difficult. The barriers are: (a) the number of paths in CFG increases exponentially with non-nested branch nodes, and (b) the verification requires checking the feasibility of paths in a CFG. If a path is unsafe, the verifier must check if it is feasible. Without the feasibility check, it could be a false positive. The Linux kernel has some very complex CFGs with a large number of branch nodes, resulting in five hundred thousand paths. Without some compact form of evidence, the human can be lost while trying to understand or cross-check the intra-procedural part of the verification.

An Example: What compact evidence could formal methods generate to circumvent the intra-procedural barriers? Let us look at a small example to motivate the answer. Figure 2 shows an example of a CFG (G) and $\hat{G}$ which is a compact form of the CFG. We call the compact form the *event flow graph* (EFG). The graph G (the CFG) in Figure 2(a) has 5 branch nodes resulting in 8 paths after the `Lock`. Some of these paths go through a complex loop with two exits. The 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 are eliminated in the EFG.

Note that the EFG captures all the information that is relevant to verify correct pairing of `Lock` and `Unlock`. However, the EFG is simpler because it has only one path corresponding to the CFG paths that contain the same sequence of events relevant to the verification.

The EFG also simplifies the path feasibility check. As seen from the EFG in Figure 2(b), there is a path with a missing unlock and the feasibility of that path must be checked. It is a violation only if that path is feasible. The EFG has retained only the condition that is necessary to verify the feasibility of that path, the other 4 conditions from the CFG are not retained in the EFG. The analyst can easily cross-check the verification by observing that if the lock is granted then the particular condition is `false`. So, the `true` path in Figure 2(b) is not feasible and thus it is not a violation.

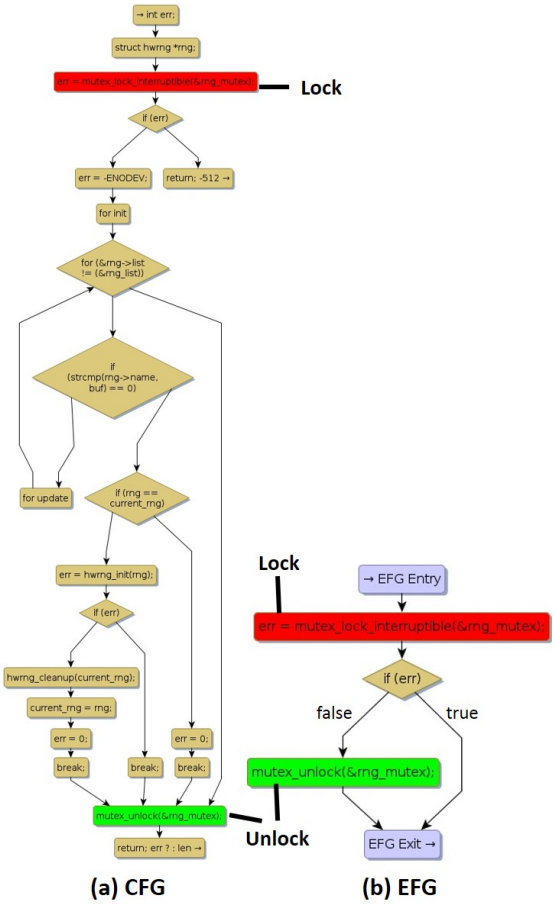

Fig. 2. CFG and EFG for the function `hwrng_attr_current_store`

The general notion of EFG, an algorithm for computing EFG, and a tool for computing EFG are presented in the paper [21]. In essence EFG introduces an *equivalence relation* on the CFG paths to partition them into equivalence classes that capture relevant information for verification.

Practical Benefits of EFG: Let us summarize how EFG is beneficial as intra-procedural evidence for verification.

1) EFG minimizes the effort for checking paths by having one path for each distinct trace of *relevant events* for verification. Events correspond to statements in a CFG. Note that a call to a function is a relevant event if that function calls the `Lock` or the `Unlock`. Other examples of relevant events include aliasing of a pointer `p` to a lock object, passing of `p` to a function, in general a relevant statement is one that is relevant to verification.
2) EFG minimizes the effort for checking path feasibility check by retaining only the subset of branch nodes relevant for verification.
3) The EFG complements the IVG by being useful for understanding the intra-procedural part of verification proof.

Use of the EFG for Intra-procedural Verification: Let us now discuss how the EFG is used at each step of the modular proof that a user can construct using the IVG and EFG. Recall that the proof proceeds in 5 steps for the example. We illustrate the use of EFG for step 3 to verify the correct pairing of `Lock`

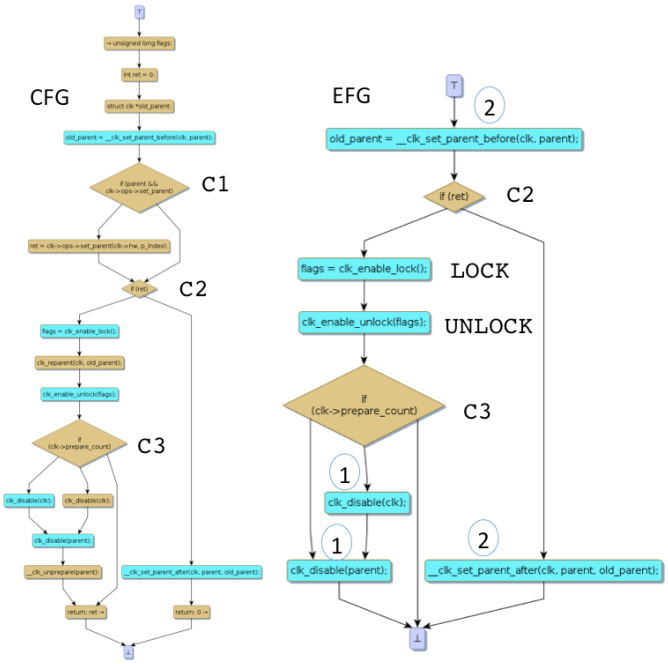

Fig. 3. CFG and EFG for the function `clk_set_parent`

and `Unlock` in the method `_clk_set_parent` (the node labeled 3 in the IVG shown in Figure 1). As seen from the IVG, this method calls both the methods verified at step 2, one method verified at step 1, and directly calls `Lock` and `Unlock`.

The EFG for `_clk_set_parent`, shown in Figure 3, includes the relevant part from the corresponding CFG also shown in the same figure. The relevant part includes the direct and indirect calls to `Lock` and `Unlock` and the branch nodes governing the paths on which these calls occur. One branch node (labeled `C3` in CFG) is not retained in the EFG. This branch node is irrelevant because the paths governed by it are equivalent for the purpose of verification. Note that the indirect calls are through the methods that are verified in previous steps. The calls to these methods are labeled in the EFG shown in Figure 3. The calls labeled 1 are to the same method `clk_disable`.

With the help of EFG, the intra-procedural verification is straight forward. The indirect calls are verified in previous steps and the direct calls to `Lock` and `Unlock`, as labeled in the EFG, are on a single path and thus trivially verified without a need for a path feasibility check.

C. Discussion

The formal verification can be incorrect. The incorrectness can be due to multiple reasons, for example it may be due to an incorrect formal specification given to the verifier. It is critically important to have capabilities to discover if the formal verification is incorrect. In Section IV, we show an example of a verification instance that LDV verifies incorrectly. We actually need systematic techniques to scrutinize correctness of formal verification. Unfortunately, the formal verification tools are not equipped to provide proofs that the human can understand and scrutinize. This incorrectness is an important issue and we elaborate on it further.

Consider a scenario where the formal specification for correct pairing is formulated so that the verification applies only to the direct calls and it ignores the calls to verified methods. The formal verification would pronounce the above verification instance as safe. The formal verification proof is incomplete but its verdict is correct. In this example, it is alright to ignore the calls to verified methods but that is not always the case. We take that up next.

Now, a hypothetical scenario where the EFG for `_clk_set_parent` include a call to a `Lock`, a call to a verified method, and then a call to `Unlock`. The formal verifier would again pronounce it safe. With the human-centric proof enabled by the IVG and the EFG, it is easy to see that the formal verifier is incorrect. It is an unsafe instance because it is a situation of `Lock` followed by `Lock` without an `Unlock` in between. This is because the hypothetical scenario has a direct call to `Lock` in `_clk_set_parent` followed by a call to `Lock` in a verified method.

The idea of evidence needs to be explored further to develop capabilities to enable a scrutiny of correctness of formal verification tools. One possibility is to use the visual models to understand the hardness spectrum and use that knowledge to create a comprehensive test suite to evaluate correctness of formal verification. For instance, the above scenario shows the need to create a test case in which a call to a verified method is in between direct calls to `Lock` and `Unlock`. Since the LDV is inconclusive on the above instance itself, we have not bothered to evaluate it with the more difficult hypothetical scenario.

The notions of IVG and EFG are central to our ongoing research to detect and prove *algorithmic complexity* and *side channel* vulnerabilities [25]. We have developed a generalized notion of EFG. The IVG notion is developed for the matching pair problem and a generalized notion for IVG is an open research problem. We have developed tools to compute the IVG and EFG and these tools have been used to produce automatically the IVG and EFG for lock-unlock pairing and memory allocation-deallocation pairing problems.

IV. A Linux Bug Missed by Formal Verification

We present an example of a complex Linux verification instance where `Lock` is not correctly paired with `Unlock`, but LDV mistakenly verifies it as correct pairing. The EFG of a function shows that `Lock` is not followed by `Unlock`. We expected that LDV verifier would notice it and declare it unsafe. To our surprise, LDV has declared it a safe instance. We did a further investigation and found that it is an unsafe instance but not for the obvious reason. Since the formal verification proof is not revealed, it is not clear why LDV has verified this instance incorrectly.

As a part of the DARPA project we are developing tools to analyze the visual models automatically. The variety of many complexities for verification makes Linux a good test case for us to evaluate and advance our tools. We have an ongoing project to scrutinize all the visual models to discover difficult verification scenarios.

This particular instance attracted our attention because of a peculiarity the EFG exhibited. The EFG 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 lock 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 a tool we have developed using Atlas [19], we find the situation shown in Figure 4. 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.

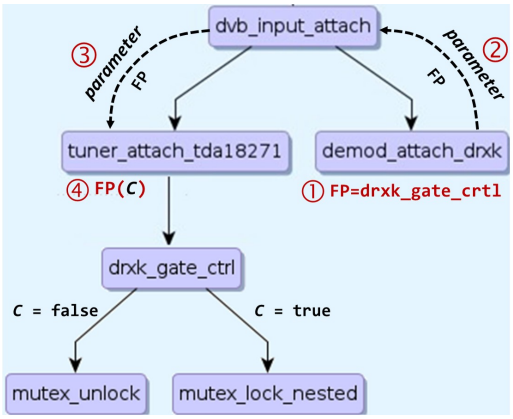

Fig. 4. Search model for drxk_gate_crtl after resolving calls via function pointers

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.

It is a mystery why LDV incorrectly verifies this instance as safe. We are not aware any program analysis techniques that would correctly resolve the difficult-to-resolve function pointer situation encountered in this instance. It is hard to imagine that LDV has resolved the function pointers in this case. An analysis without resolving function pointers is more likely to lead to unsafe verdict because the Lock and Unlock are on disjoint paths and thus it is an unsafe instance. 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.

V. Key Ideas for Future Advances

We discuss two key ideas to address the challenges of making formal verification useful to practising engineers. These ideas have evolved from our Linux verification studies and visionary papers: Social Processes and Proofs of Theorems, by De Millo, Perlis (1st Turing Award, 1966) and Lipton [6], [26], and Computer Scientist as a Toolsmith, by Brooks (Turing Award, 1999) [7].

A. Human-Centric Verification

The first paper [6] makes following important points.

- Mechanisms that make engineering and mathematics really work are obscured in the fruitless search for perfect verifiability. In mathematics, the aim is to increase the human confidence in the correctness of a theorem. Nor does the proof settle the matter, contrary to what its name suggests, a proof is only one step in the direction of confidence. It is a social process that determines whether mathematicians feel confident about a theorem. Verification is nothing but a model of believability. It cannot be a model where proofs are accepted in blind faith. A proof should be amenable to human scrutiny.
- A good proof is one that makes us wiser. With formal verification, we know that our program is formally, correct. We do not know, however, to what extent it is reliable, dependable, safe; We do not know within what limits it will work; we do not know what happens when it exceeds those limits. The verification must provide knowledge that improves our practice.

Perfect verifiability is clearly not possible. The question is what would help to establish more trust in the correctness of formal verification. Because of the low-level at which formal verification operates, the formal proofs are extremely long and not amenable to human understanding. The Linux bug discussed in Section IV shows that either the formal verifier itself or the specification being generated is not working correctly. Except declaring that the instance is SAFE, the verifier produces no other output. The user has no clue what is not working correctly. Had the verifier produced the IVG and the corresponding EFGs resulting from its verification, we could have used those to gain deeper insights about the formal verification as performed by LDV.

As the paper [6] notes, a good proof is one that makes us wiser. The paper makes distinction between formal verification proofs and the proofs used in mathematics and points to the intrinsic problem that formal proofs do not make us wiser about the software.

To summarize, the first key idea is to advance formal methods to produce artifacts that make developers wiser about their software.

B. Human-Machine Collaboration

Frederick Brooks writes [7]: *"If indeed our objective is to build computer systems that solve very challenging problems, my thesis is that IA > AI, that is, that intelligence amplifying systems can, at any given level of available systems technology, beat AI systems. That is, a machine and a mind can beat a mind-imitating machine working by itself."* Both verification instances we have presented illustrate why human-machine collaboration is important.

Software assurance inherently involves learning and reasoning about large code. The overarching question for any software assurance technology is: should this learning and reasoning be machine-centric or human-centric? The traditional vision is machine-centric with formal methods based on low-level formal specifications. While this vision has led to incremental progress over the last fifty years, there has been a quantum jump in escalated safety and security risks emanating from software failures.

Formal verification inherently involves problems with exponential complexity as well as intractable problems that are equivalent to the *halting problem*. Of the two verification instances we have presented here as examples, the formal

verification is inconclusive on one instance discussed in Section III. While the instance is difficult for the formal verifier, we have shown a modular proof to verify the instance. The visual models that serve as the evidence for this proof are produced automatically. It is an example of human-machine collaboration to verify an instance which is beyond the reach of the top-rate, customized formal verifier for Linux. The second instance discussed in Section IV is much harder. Paradoxically, the formal method verifies the second instance which is much harder. However, the formal verification is not correct.

The bug we have shown for the second instance is produced by human-machine collaboration as described in using Atlas [19]. As described in Section IV, the second instance involves a complex resolution of function pointers which would be very time consuming and prone to errors without automation. However, it is hard if not impossible to make it fully automated. Moreover, the crucial starting point is a human hypothesis. The function `f` in that instance has `Lock` and `Unlock` on two disjoint paths governed by some condition `C`. It is human hypothesis that `f` is called twice first with `C=TRUE` for `Lock` and then with `C=FALSE` for `Unlock`. The human hypothesis is then confirmed by using an automated tool to search for this coding pattern.

As described in [19], our technique to resolve the function pointers requires human intelligence, domain-knowledge, and powerful automated querying to mine programs. It requires program mining because one needs to search if there is a point in program where a function pointer is set to point to the given function. It is actually quite complicated because the Linux kernel has multiple functions with the same name and it requires a careful resolution of scope that a function pointer is set to the particular function.

VI. Conclusion

As the paper [6] notes, a good proof is one that makes us wiser. The paper makes distinction between formal verification proofs and the proofs used in mathematics and points to the intrinsic problem that formal proofs do not make us wiser about the software.

Following up on the idea of "proofs to make us wiser," this paper makes a case for producing visual models as supporting evidence for formal verification. Developing visual models as evidence for formal verification is challenging research that requires significant systematic experimentation to determine what should be the evidence for addressing practical needs. As it turns out, much the same is true in mathematics as well. Most mathematicians spend a lot of time thinking about and analyzing particular examples [26]. This motivates future development of theory and gives one a deeper understanding of existing theory. Gauss declared, and his notebooks attest to it, that his way of arriving at mathematical truths was "through systematic experimentation." Our DARPA research focuses on advancing a platform for performing software modeling experiments. The visual models (EFG and IVG) developed through our experiments are used here in the presented empirical study to make a case for producing supporting evidence for formal verification.

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.

References

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

[2] P. Stratis, "Formal verification in large-scaled software: Worth to ponder," 2014. [Online]. Available: https://blog.inf.ed.ac.uk/sapm/2014/02/20/formal-verification-in-large-scaled-software-worth-to-ponder/

[3] D. Beyer and A. K. Petrenko, "Linux driver verification," in *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*. Springer, 2012, pp. 1–6.

[4] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, p. 19, 2009.

[5] "Linux driver verification (LDV) tool," http://linuxtesting.org/project/ldv.

[6] R. A. De Millo, R. J. Lipton, and A. J. Perlis, "Social processes and proofs of theorems and programs," *Communications of the ACM*, vol. 22, no. 5, pp. 271–280, 1979.

[7] F. P. Brooks Jr, "The computer scientist as toolsmith II," *Communications of the ACM*, vol. 39, no. 3, pp. 61–68, 1996.

[8] B. Gates, "Bill Gates Keynote: Microsoft Tech-Ed 2008," 2008. [Online]. Available: http://news.microsoft.com/speeches/bill-gates-keynote-microsoft-tech%E2%80%A2ed-2008-developers/

[9] O. Lhoták, "Program analysis using binary decision diagrams," Ph.D. dissertation, McGill University, 2006.

[10] A. Church, "A note on the entscheidungsproblem," *J. Symb. Log.*, vol. 1, no. 1, pp. 40–41, 1936.

[11] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *J. of Math*, vol. 58, no. 345-363, p. 5, 1936.

[12] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953.

[13] Y. Xie and A. Aiken, "Saturn: A scalable framework for error detection using boolean satisfiability," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 29, no. 3, p. 16, 2007.

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

[15] D. Beyer, "Status report on software verification," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014.

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

[17] "Ensoft corp." http://www.ensoftcorp.com.

[18] K. Gui and S. Kothari, "A 2-phase method for validation of matching pair property with case studies of operating systems," in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*. IEEE, 2010, pp. 151–160.

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

[20] S. Kothari, A. Tamrawi, J. Sauceda, and J. Mathews, "Let's verify linux: accelerated learning of analytical reasoning through automation and collaboration," in *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016, pp. 394–403.

[21] A. Tamrawi and S. Kothari, "Event-flow graphs for efficient path-sensitive analyses," *arXiv preprint arXiv:1404.1279*, 2014.

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

[23] J. Gleick and R. C. Hilborn, "Chaos, making a new science," *American Journal of Physics*, vol. 56, no. 11, pp. 1053–1054, 1988.

[24] E. N. Lorenz, "Deterministic nonperiodic flow," *Journal of the atmospheric sciences*, vol. 20, no. 2, pp. 130–141, 1963.

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

[26] D. Epstein and S. Levy, "Experimentation and proof in mathematics," *Notices of the AMS*, vol. 42, no. 6, pp. 670–674, 1995.